TU WIEN Informatics

Towards a Generic Framework for Extending System Architectures of Autonomous Vehicles with a Self-Managing Functionality

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Tobias Kain, BSc
Registration Number 01329088

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.-Prof. Mag.rer.nat. Dr.techn. Hans Tompits

The dissertation has been reviewed by:

Dr. Marina De Vos	Prof. Dr. Francesco Ricca

Vienna, 3rd March, 2025

Tobias Kain

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Tobias Kain, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. März 2025

Tobias Kain

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor, Hans Tompits, for his persistent support and guidance throughout my academic journey. Our collaboration began with my Bachelor's thesis, continued through my Master's thesis, and culminated in this dissertation. Over these years, Hans has taught me what it truly means to conduct academic research. I am especially thankful for the many hours we spent together discussing ideas, solving problems, and refining my work. Beyond his role as a mentor, I am profoundly grateful for the personal bond we developed.

Next, I would like to thank my fellow students and colleagues who accompanied me throughout my academic journey. In particular, I am grateful to Toni and Max for their support during our Bachelor's and Master's studies. Working with them has always been a pleasure, and I truly value the deep friendship we have built over the years. I would also like to thank everyone who contributed to my publications, especially Timo, Johannes, Fabian, Julian, Marcel, and Max. Collaborating with you not only enriched my work but also taught me a lot about academic research. Finally, I would like to thank Yasin for the mutual encouragement as we pushed each other toward completing our dissertations.

Furthermore, I would like to express my heartfelt gratitude to my family and friends for their unwavering support. I am deeply thankful to my parents, Uschi and Hans, and my sister Sonja, for standing by me, especially during the early years of my life, and for laying the foundation upon which I could build my career. I am particularly grateful to my grandparents, Oma and Opa, for always believing in me, offering me comfort and warmth, and for their unfailing pride in my achievements. Their financial support also gave me the invaluable opportunity to pursue my studies. I am also very appreciative to my grandmother, Wiener Oma, whose generosity in providing her apartment in Vienna made it possible for me to study here. Furthermore, I would like to thank my beloved girlfriend, Eli, for always standing by my side and being my source of motivation when I needed it most. I am as proud of you as you are of me for being so close to finishing your medical degree. Soon, we will celebrate earning our doctorates together. Finally, I would like to express my heartfelt gratitude to my cat, Pezi, who has been by my side throughout my entire academic journey, faithfully keeping me company during countless hours of studying and writing.

Kurzfassung

Die Automobilbranche erfährt derzeit aufgrund des rasanten technologischen Fortschrittes eine zunehmende Digitalisierung. Ein zentrales Ziel in dieser Branche ist die Einführung von *autonomen Fahrzeugen*, die gemäß der *Society of Automotive Engineers* (SAE) als Fahrzeuge definiert werden, die mit einem automatisierten Fahrsystem (ADS) ausgestattet sind, welches die gesamte Fahraufgabe übernimmt. Folglich benötigen solche Fahrzeuge keinen menschlichen Fahrer.

Die Entwicklung autonomer Fahrzeuge steht jedoch vor einer Vielzahl von Herausforderungen, wobei eine maßgebliche darin besteht, einen sicheren Betrieb in unterschiedlichen Kontexten zu gewährleisten. Autonome Fahrzeuge müssen nämlich in der Lage sein, Kontextänderungen, welche zum Beispiel durch Software- und Hardwarefehler sowie wechselnde Wetterbedingungen hervorgerufen werden können, ohne menschliche Unterstützung zu bewältigen. Um dies bewerkstelligen zu können müssen autonome Fahrzeuge *selbstverwaltend* (engl. *self-managing*) sein. Die Umsetzung von Ansätzen zur Selbstverwaltung von autonomer Fahrzeuge führt jedoch zu zusätzlicher Komplexität in der Entwicklung autonomer Fahrzeuge. Eine Gegenmaßnahme um diese steigende Komplexität zu reduzieren ist die Integration selbstverwaltender Fähigkeiten in die Architektur autonomer Fahrzeuge, wofür jedoch gegenwärtig nur wenige Ansätze existieren.

In dieser Arbeit adressieren wir dieses Problem und stellen eine *generische selbstverwaltende Rahmenstruktur für autonome Fahrzeuge* namens APTUS vor. Unsere vorgeschlagene Rahmenstruktur erfordert keine spezifische zugrundeliegende Fahrzeugarchitektur und ist in diesem Sinne generisch. Insbesondere ist ein Ziel von APTUS, beliebige Systemarchitekturen autonomer Fahrzeuge um eine selbstverwaltende Funktionalität zu erweitern.

APTUS wurde in Anlehnung an die allgemeine dreischichtige Architektur, die Ende der 1990er Jahre von Gat eingeführt wurde und die im Bereich der Robotik weit verbreitet ist, entworfen. APTUS ist unterteilt in drei logische Schichten, nämlich (i) der *Kontextschicht*, (ii) der *Rekonfigurationsschicht* und (iii) der *Komponentenschicht*. Jede dieser Schichten implementiert verschiedene selbstverwaltende Eigenschaften wobei die von APTUS implementierten Schlüsseleigenschaften *Kontextsensitivität*, *Selbstkonfiguration* und *Selbstheilung* sind.

Die Kontextschicht ist für die Umsetzung der Kontextsensitivität verantwortlich. Konkret leitet diese Schicht aus Kontextbeobachtungen eine Menge sogenannter *Software-Architekturanforderungen* ab. Diese Anforderungen umfassen beispielsweise die Menge der auszuführenden Anwendungen, deren Leistungsanforderungen sowie die Redundanzanforderungen. Zur Ermittlung der Software-Architekturanforderungen stellen wir in dieser Arbeit C-sar vor. Bei der Umsetzung von C-sar verwenden wir großteils *Answer-Set Programming* (ASP), ein bekanntes deklaratives Programmierparadigma aus dem Bereich der logikbasierten künstlichen Intelligenz.

Die von der Kontextschicht ermittelten Software-Architekturanforderungen dienen als Eingabe für die Rekonfigurationsschicht. Diese Schicht ist dafür verantwortlich, Zuordnungen zwischen Rechenknoten und Anwendungen, sogenannte *Konfigurationen*, zu berechnen. Zudem wird die Rekonfigurationsschicht auch von der Komponentenschicht angewiesen, das Fahrzeug als Reaktion auf Hardware- oder Softwarefehler neu zu konfigurieren. Im Kern enthält die Rekonfigurationsschicht zwei sogenannte *Application-Placement-Problem-Solver*: logAP^{2}S und linAP^{2}S. Diese Solver sind dafür verantwortlich, gültige Konfigurationen zu finden, wobei logAP^{2}S durch den Einsatz von ASP die Anwendung komplexer nichtlinearer Optimierungsfunktionen ermöglicht. Hingegen ermöglicht der Einsatz von Integer Linear Programming in linAP^{2}S schnelle Lösungszeiten.

Die Komponentenschicht ist für die Selbstheilung des Systems verantwortlich. Daher integrieren wir in diese Schicht unseren Fehlertoleranzansatz Fdiro, der meiner Masterarbeit entspringt. Fdiro definiert vier aufeinanderfolgende Schritte, um das System nach einem Software- oder Hardwarefehler in einen sicheren und optimierten Zustand zu überführen, nämlich: *Fehlererkennung*, *Fehlerisolierung*, *Wiederherstellung* und *Optimierung*. Da Fdiro in seiner ursprünglichen Form nicht in der Lage ist, ausgefallene Hardwarekomponenten wie zum Beispiel Rechenknoten oder Sensoren wiederherzustellen, führen wir in dieser Arbeit Hrr ein. Hrr ist eine Erweiterung von Fdiro und ermöglicht die Wiederherstellung ausgefallener Hardware.

Die Evaluierung von Aptus erfolgt anhand eines Anwendungsfalls, der ein autonomes Fahrzeug umfasst, welches unterschiedlichen Kontextänderungen ausgesetzt wird. Der vorgestellte Anwendungsfall veranschaulicht, wie Aptus das autonome Fahrzeug dynamisch an die sich ändernden Bedingungen anpasst. Die Evaluierung unterstreicht die Effektivität von Aptus und seinen Teilkomponenten, insbesondere C-sar, logAP^{2}S, linAP^{2}S, Fdiro und Hrr, bei der Realisierung der Selbstverwaltung des autonomen Fahrzeugs.

Abstract

The automotive field is currently experiencing increasing digitalization due to rapid technological advancements. A central goal in this field is the introduction of *autonomous vehicles*, which, according to the *Society of Automotive Engineers* (SAE), are vehicles that are equipped with an automated driving system (ADS) that performs the entire driving task, i.e., such vehicles do not rely on a human driver.

However, the development of autonomous vehicles faces a multitude of challenges. A significant challenge is to ensure a safe operation in diverse contexts. Autonomous vehicles must be capable of handling context changes, such as software and hardware faults, as well as changing weather conditions, without human support, i.e., autonomous vehicles have to be *self-managing*. Due to numerous context changes that need to be handled, self-managing approaches introduce additional complexity in the development of autonomous vehicles. A countermeasure against the rising complexity is the incorporation of self-managing capabilities into a systematic design architecture of autonomous vehicles. However, currently, there is a lack of approaches in this direction for autonomous vehicles.

In this dissertation, we address this issue and introduce Aptus, a *generic self-managing framework for autonomous vehicles*. Our proposed framework is generic in the sense that it does not require a specific underlying vehicle architecture. In particular, the aim of Aptus is to extend arbitrary system architectures of autonomous vehicles with a self-managing functionality. Following the general three-layered architecture as introduced by Gat in the late 1990s, which is widely applied in the field of robotics, Aptus likewise consists of three logical layers, namely, (i) the *context layer*, (ii) the *reconfiguration layer*, and (iii) the *component layer*. The layers of Aptus implement various properties that are required for a system to be considered self-managing, whereby the key properties implemented by Aptus are *context-awareness*, *self-configuration*, and *self-healing*.

The context layer is responsible for implementing the context-awareness property. Specifically, the context layer derives a set of so-called *software-architecture requirements* from contextual observations. These requirements encompass, for instance, the set of applications that need to be executed, as well as their performance demands and their required redundancy. To implement the task of deriving software-architecture requirements from contextual observations, we introduce the tool C-sar, which is based on *answer-set programming* (ASP). The latter is a well-known declarative programming paradigm from the area of logic-based artificial intelligence.

The software-architecture requirements determined by the context layer constitute the input of the reconfiguration layer. This layer is required to compute so-called *configurations*, i.e., assignments between computing nodes and applications, that fulfill the needs expressed by the software-architecture requirements. Furthermore, the reconfiguration layer is also instructed by the component layer to reconfigure the vehicle in response to hardware and software faults. At its core, the reconfiguration layer contains two so-called *application-placement problem solvers*, referred to as `logAP`^{2}`S` and `linAP`^{2}`S`, respectively. These solvers are responsible for finding valid configurations, whereby `logAP`^{2}`S` enables applying complex non-linear optimization functions by employing ASP while `linAP`^{2}`S` achieves fast solving times by employing integer linear programming.

Finally, the component layer implements the self-healing property. To this end, APTUS incorporates an adaption of FDIRO, a fault-tolerance approach which we introduced in previous work that defines four successive steps, viz., *fault detection*, *fault isolation*, *recovery*, and *optimization*, to bring the system to a safe and optimized state after a software or hardware fault. FDIRO, in its original design, cannot recover failed hardware components like computing nodes or sensors. To address this limitation, we introduce HRR, which extends FDIRO and enables the recovery of failed hardware.

To evaluate APTUS, we present a use-case scenario involving an autonomous vehicle undergoing various context changes. This use case demonstrates how APTUS dynamically adapts the autonomous vehicle to changing conditions. The evaluation highlights the effectiveness of APTUS, particularly the role of C-SAR, `logAP`^{2}`S`, `linAP`^{2}`S`, FDIRO, and HRR in achieving self-management.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The automotive industry is undergoing a rapid digital transformation driven by technological advancements. A key objective in this domain is the development of *autonomous vehicles*, defined by the *Society of Automotive Engineers* (SAE) [189] as vehicles equipped with an automated driving system (ADS) that is capable of performing the entire driving task, and thus eliminating the need for a human driver. Fully autonomous vehicles like these are classified as ones having *SAE level 4* or *SAE level 5*.

The development towards autonomous vehicles is to be welcomed since these can lead to safer roads [163] and an improved traffic flow [64]. Autonomous vehicles can also cause a social impact by, for example, providing mobility to elderly or disabled people [62]. Furthermore, vehicle autonomy can lead to a reduced energy consumption and fewer emissions [169, 203] through mechanisms such as route optimization [22], congestion mitigation [146], and platooning solutions [209].

However, the development of autonomous vehicles brings various challenges. Many of these result from the fact that autonomous vehicles are complex systems [151, 132], where multiple advanced hard and software components are required to operate a vehicle autonomously. The high complexity of automated driving systems is partly due to the fact that various environmental contexts, e.g., changing illumination and weather circumstances, have to be handled [161]. Furthermore, autonomous vehicles might be operated in several operation modes, like an autonomous mode, a manual mode, or a test mode. These operation modes require, for instance, a different set of applications to be executed by the vehicle [124].

The complexity of automated driving systems is further increased as unpredictable system context changes like ones triggered by hardware or software faults must be handled. This is vital in order to counteract safety-critical situations [13]. For instance, if an application is essential for operating the autonomous vehicle, e.g., the application responsible for detecting objects or planning the vehicle's trajectory, immediate mitigation measures must be taken. Conceivable mitigation measures involve, for instance, activating redundant components that take over the responsibilities of faulty components or performing an emergency stop.

As autonomous vehicles are not meant to be operated by humans, the vehicle itself has to be equipped with means to perform adaptations if these are required due to occurring context changes. Even if a human operator could intervene and perform adaptations, due to the system's complexity, extensive knowledge about the system is required to perform appropriate adaptations. This problem is not only limited to autonomous vehicles but to complex systems in general [69, 102]. To this end, IBM introduced the concept of *autonomic computing* [127] in order to handle the increasing complexity of computer systems that operate in different contexts. The idea of autonomic computing is that a system is *self-managing*, i.e., the system can manage itself without the help of an interventing administrator.

Incorporating self-managing capabilities into autonomous vehicles is of growing interest [232, 219]. As well, such an incorporation can enhance the dependability attributes of autonomous vehicles such as safety, reliability, and availability [201]. However, the introduction of self-managing capabilities induces additional complexities, which can cause negative effects [207]. Therefore, methods are required to reduce the complexity [152]. Moreover, embedding the implementation of self-managing properties into a systematic design architecture can reduce the complexity of self-managing systems [231]. Consequently, *prospective self-managing approaches for autonomous vehicles should be integrated into the system architecture.* Furthermore, since there are no uniform system architectures of autonomous vehicles and, in the upcoming years, the software and hardware components of autonomous vehicles will further evolve, *general concepts for integrating self-managing capabilities which can be adapted to arbitrary ADS implementations are required.*

Currently, only a few self-managing approaches for autonomous vehicles have been introduced [227, 228, 43, 50]. These approaches are, however, either vague theoretical concepts or cover only certain self-managing properties. Hence, a lack of comprehensive self-managing approaches for autonomous vehicles can be identified.

Various aspects have to be considered in order to implement a self-managing framework for autonomous vehicles:

(i) the framework has to be aware of its context to determine the required functionality;

(ii) it must be able to perform adaptations so that the required functionality can be achieved; and

(iii) it must be aware of the current state of the system and react to unexpected events.

An architecture that is well-suited to implement these required functions is the three-layered architecture introduced by Gat [70], which is widely applied in the field of robotics. Gat's three-layered architecture has already been adapted, e.g., by Kramer and Magee [135] to implement a three-layered model for a self-managed system. However, for the use case of introducing self-managing capabilities in autonomous vehicles, adaptations to this architecture are required.

In this dissertation, we address this issue by introducing the framework Aptus, an approach specifically designed for autonomous vehicles which allows extending a given system architecture with a self-managing functionality (the name "Aptus" refers to the Latin adjective "aptus", standing for "adapted" or "suitable", among other meanings).

Following the approach of Gat [70], Aptus is likewise designed in a three-layered fashion comprising

(i) a *context layer*,

(ii) a *reconfiguration layer*, and

(iii) a *component layer*.

The individual layers implement interconnected approaches, whereby each layer implements a different set of self-managing properties. To the best of our knowledge, no concept has been published so far in the domain of autonomous driving that implements self-managing properties to a comparable degree as Aptus.

Our framework is generic in the sense that it is not based on a specific ADS architecture. The approaches which are implemented by Aptus to introduce self-managing capabilities are independent of the hardware and software architecture of the autonomous vehicles. A concrete implementation of Aptus in a specific vehicle would require defining and realizing the interface as provided by Aptus.

In the following sections, we first provide some background information about self-managing systems. Afterwards, in Section 1.3, we outline the main contributions of our work. Section 1.4, provides an overview of the structure of the dissertation. Finally, Section 1.5 lists the publications achieved so far associated with our contributions.

1.2 Self-Managing Systems

Autonomic computing was inspired by the human nervous system [127], which is, for instance, capable of controlling the body temperature, blood sugar level, and digestive system autonomously. Likewise, systems which implement the autonomic computing paradigm are, for example, able to efficiently manage the available resources, adapt to changing circumstances, and handle occurring faults. Such systems are referred to as *self-managing systems* or *self-adaptive systems* [191]. Note that the terms "autonomous computing”, “self-managing systems”, and “self-adaptive systems” are frequently used interchanging [102].

IBM defined the following properties that characterize a self-managing system [100, 179]:

(i) *Self-awareness*: A self-aware system “knows itself”. For instance, a self-aware system knows its hardware and software components, current state, and resource capacity.

(ii) *Context-awareness*: A context-aware system can detect context changes and adapt its behavior accordingly.

(iii) *Self-configuration*: A self-configuring system is able to adapt its configuration dynamically. The performed adaptation actions can, e.g., include starting and stopping hardware and software components.

(iv) *Self-healing*: A self-healing system can detect hardware and software faults. Furthermore, a system, which is considered self-healing, is able to recover from such faults.

(v) *Self-optimization*: A self-optimizing system is able to improve its operation. Self-optimizing actions can, for example, include increasing the throughput and minimizing resource usage.

(vi) *Self-protection*: A self-protective system is able to detect and prevent internal and external malicious attacks.

(vii) *Genericity*: A self-managing system shall be portable to other hardware and software architectures.

(viii) *Anticipativeness*: A self-managing system shall anticipate the needed resources and behavior to handle upcoming situations.

Often only the properties of self-configuration, self-healing, self-optimization, and self-protection are considered when referring to self-managing systems [127, 69]. However, here, we consider all eight properties, to which we refer to as the *self-managing properties*.

A similar concept to autonomic computing is the vision of *organic computing* [195], which aims to extend the concept of autonomic computing toward distributed embedded systems.

Besides the domain of autonomous driving, self-managing systems are required in many diverse fields [160, 220, 55], like the field of robotics [230], cloud computing [47], aerospace [229], healthcare [80], and manufacturing [181].

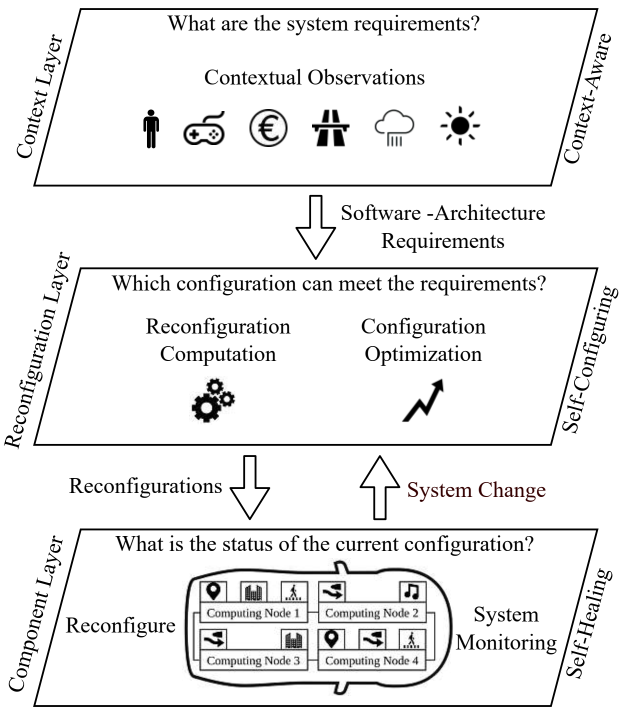


Figure 1.1: The three logical layers of Aptus and their relationships.

1.3 Contributions

As mentioned above, in this dissertation, we present the framework Aptus, comprising the *context layer*, the *reconfiguration layer*, and the *component layer*. Figure 1.1 illustrates these three interconnected logical layers.

Following the definition of self-managing systems, as discussed in the previous section, Aptus has to ensure that all eight self-managing properties are fulfilled.

The context layer achieves the context-awareness property. In particular, the context layer of Aptus is responsible for determining the so-called *software-architecture requirements*, including, e.g., the set of the required applications, their performance demands, and their required level of redundancy, from various contextual observations. For achieving this task, we realized the tool C-sar, standing for “context-based software-architecture requirements”, which employs answer-set programming (ASP) [78, 19, 73, 77] to handle the required reasoning problems and which is implemented in Python. ASP is a well-known declarative problem-solving approach which has been, for instance, successfully applied in the domain of robotics [60], natural-language processing [20], and semantic-web reasoning [59]. ASP is a suitable choice for our purposes since it allows to conveniently formulate the relationship between the software-architecture requirements and the current context using a rule-based encoding that can be efficiently solved by sophisticated ASP solvers—in particular, we use the ASP solver clasp [75].

Furthermore, the context layer employs an approach to dynamically degrade configurations, called D-DEG, by means of receiving data from nearby vehicles. D-DEG then determines, based on the information received, whether applications can be degraded.

The output of the context layer is a set of software-architecture requirements that are used as input for the reconfiguration layer. The focus of the reconfiguration layer is to implement the self-configuration property. Furthermore, this layer also realizes the self-optimization property and anticipativeness.

To implement the self-configuration property, we introduce a novel *application-placement problem solver*, called logAP^{2}S, which, like C-SAR, employs answer-set programming. By the *application-placement problem* we understand the task of determining a new *configuration*, i.e., an assignment between applications and computing nodes, in accordance with various constraints and preferences. Employing ASP for solving the application-placement problem allows for optimizing configurations according to different complex optimization functions. However, the issue with applying complex optimization functions is the risk of extended solving times. Fast reconfiguration times are especially required if a software or hardware fault triggers a reconfiguration. This problem is addressed in Aptus by incorporating an additional application-placement problem solver, called linAP^{2}S [116], that is based on *integer linear programming* [196]. This solver employs a static linear optimization function and can therefore guarantee fast reconfiguration times. Aptus uses linAP^{2}S in case a new configuration must be computed quickly. On the other hand, logAP^{2}S is used in case a configuration has to be optimized according to complex, possibly non-linear, optimization functions.

Furthermore, the reconfiguration layer realizes the self-optimization property. To implement this, we introduce C-PO$_{\text{Aptus}}$, which optimizes the current configuration by selecting optimization functions for logAP^{2}S. C-PO$_{\text{Aptus}}$ is an adaption of our general context-based optimization approach C-PO [121].

Besides the self-configuration and self-optimization properties, the reconfiguration layer also implements anticipativeness in order to save resources. To implement this property, we introduce the configuration-graph manager CGM, which manages precomputed configurations that are likely to be requested to support short configuration-provision times. Furthermore, precomputed configurations will be uploaded to a central cloud-based service in order to share configurations among vehicles. Before computing a new configuration, CGM examines whether the configuration has been precomputed. If this is the case, the precomputed configuration is applied. Thus, the resources required to compute this configuration can be saved. An early idea of CGM was outlined in preliminary work [122].

Finally, the component layer implements the self-healing property. To fulfill this property, we incorporate Fdiro [116, 122], a fault-tolerance approach originally developed for autonomous vehicles. In this work, we adapt Fdiro to suit our framework, enabling the system to transition into a safe and optimized state after a software or hardware fault by performing four successive steps: *fault detection*, *fault isolation*, *recovery*, and *optimization step*.

In its initial implementation, FDIRO is not capable of recovering failed hardware components like computing nodes or sensors. Consequently, only a limited number of hardware faults can be tolerated before the mission has to be terminated by an emergency stop. Therefore, we introduce the hardware redundancy-recovery approach HRR, which is an extension to FDIRO.

For evaluating APTUS, we present a use-case scenario in which an autonomous vehicle is considered, based on a widely adopted generic functional architecture [13], that performs commercial missions in an urban environment. During these missions, various context changes occur, such as software and hardware faults, as well as changing weather and illumination conditions, that are handled by APTUS.

This example allows us to demonstrate the interactions between the different layers of APTUS. Furthermore, to evaluate the reliability of a computed configuration, we implemented a simulation environment called `AT-CARS`, which uses a Monte Carlo simulation [166]. The basic idea of `AT-CARS` is to evaluate the vehicle's system reliability over time by continuously injecting faults that are subsequently handled by APTUS. Our reliability analysis shows that FDIRO and HRR positively impact the reliability of the autonomous vehicles assumed in the use case.

Besides implementing the self-healing property, the component layer also provides the option to include self-protective measures. However, to keep the scope of APTUS manageable, we do not consider the self-protection property in this dissertation. Nevertheless, we illustrate how APTUS can be extended to become self-protective.

Note that some self-managing properties are not implemented by an individual layer but are instead cross-layer properties. In particular, our approach is generically designed and can be, therefore, transferred to different hardware and software architectures. Furthermore, self-awareness is another cross-layer property. We assume that the underlying vehicle platform implements the self-awareness property. Consequently, the vehicle platform can inform APTUS, for instance, regarding the currently available hardware and software components and the current resource usage.

To summarize, the main contributions of our work are as follows:

- We introduce APTUS, a generic self-managing framework for extending system architectures of autonomous vehicles with self-managing functionality.
- We present C-SAR, which is a tool for determining software-architecture requirements that reflect the demands in the current context.
- We sketch D-DEG, an approach for dynamically degrading configurations.
- We present an application-placement problem solver called logAP^{2}S that is based on answer-set programming for determining configurations that fulfill the currently required software-architecture requirements.
- We introduce C-PO$_{\text{APTUS}}$, for selecting optimization functions for logAP^{2}S.

- We integrate linAP^{2}S, an application-placement problem solver that is based on integer linear programming, into Aptus.
- We present CGM, which manages precomputed configuration.
- We incorporate the fault-tolerance approach Fdiro into Aptus.
- We present Hrr, which is an extension of Fdiro to recover faulty hardware components.
- We introduce AT-CARS, which is a tool to analyze the reliability of configurations.
- We present a use-case scenario that illustrates the workflow of Aptus.

1.4 Overview

The rest of this dissertation is structured as follows. In Chapter 2, we introduce the general idea of Aptus and its layers. Furthermore, in this chapter, we discuss related approaches. Chapter 3 introduces the context layer, which includes C-sar and D-deg while Chapter 4 contains the reconfiguration layer. Afterwards, in Chapter 5, we introduce the component layer, including our fault-tolerance approach Fdiro and its extension Hrr, as well as the simulation environment AT-CARS. Additionally, we also provide an overview of approaches related to Fdiro. In Chapter 6, we present our use-case scenario that illustrates the workflow of Aptus. Finally, in Chapter 7, we conclude the dissertation and discuss possible future work.

1.5 Publications

The foundation for the development of Aptus was laid down in my Master's thesis [116], where I introduced linAP^{2}S [123] as well as Fdiro [122]. As mentioned before, these approaches are incorporated into Aptus.

The first concept of Aptus was published at the *Second Workshop on Autonomous Systems Design* (ASD 2020) [119]. A literature study of approaches similar to Fdiro that are used in other industries was presented at the *35. VDI-Fachtagung Fahrerassistenzsysteme und Automatisiertes Fahren* (FAS 2022) [99]. Furthermore, our hardware redundancy recovery approach Hrr was introduced at the *30. VDI-Fachtagung Technische Zuverlässigkeit* (TTZ 2021) [117]. The testing environment AT-CARS that we use for evaluating Fdiro and Hrr was introduced at the *Third International Conference on Connected and Autonomous Driving* (MetroCAD 2020) [97]. C-sar and logAP^{2}S were presented at the International Conference of the *22nd Italian Association for Artificial Intelligence* (AIxIA 2023) [120]. In turn, an outline of D-deg, our approach for dynamically determining configuration degradation actions, was presented at the *31st European Safety and Reliability Conference* (ESREL 2021) [118]. Finally, C-po$_{\text{Aptus}}$ and the use-case scenario will be submitted to a special issue on "Vehicle Safety and Automated Driving" of the *Automation* journal.

CHAPTER 2

The Basic Architecture of the Framework

In this chapter, we introduce the general architecture of APTUS. As our framework is designed specifically for autonomous vehicles, we first give background information about autonomous driving and the system architecture of autonomous vehicles. Afterwards, we discuss the general layout of APTUS and related approaches.

2.1 Autonomous Driving

In recent years, car manufacturers gradually computerized their vehicles. Moreover, rapid technological advancements have made it possible to (partly) automate driving tasks. To categorize the level of automation, the Society of Automotive Engineers (SAE) introduced a classification, depicted in Table 2.1, ranging from no automation to full automation [189].

The era of autonomous cars already began in the late 1970s when engineers at the Tsukuba Mechanical Engineering Laboratory (Tsukuba, Japan) built the first prototype that was able to autonomously maneuver a car between street markings [210]. Nowadays, many automotive manufacturers, including, for example, Tesla, Mercedes-Benz, Volkswagen, and Toyota, as well as technology companies such as Google, Amazon, Nvidia, and Intel, push the research on autonomous driving forward [103]. Some companies, such as Waymo[1] and Baidu[2], are already offering autonomous ride services in selected cities. However, according to the Victoria Transport Policy Institute [149], it will likely take until the 2040s to 2060s for autonomous vehicles to become commonplace.

[1]Waymo: `https://waymo.com/`
[2]Baidu: `https://www.apollo.auto/apollo-self-driving`

Table 2.1: SAE level of driving automation.

Level 0	No Automation	The driver performs the longitudinal and lateral control.
Level 1	Driver Assistance	The system can either perform the longitudinal control using adaptive cruise control or the lateral control using a lane-centering system.
Level 2	Partial Automation	The system can perform both longitudinal and lateral control. However, the driver has to monitor the actions performed by the system.
Level 3	Conditional Automation	The system can perform both longitudinal as well as lateral control, and the driver is not required to monitor the actions performed by the system. However, the driver has to take over control, if required.
Level 4	High Automation	The system performs the driving task autonomously in some defined domain.
Level 5	Full Automation	The system performs the driving task autonomously. No driver is required.

As the development of autonomous vehicles is still in an early phase, none of the proposed software and hardware architecture concepts for autonomous vehicles is widely accepted yet. However, the high-level functional architecture, which consists of a *sensing component*, a *planning component*, and a *control component* is similar in most autonomous vehicles [216, 114, 25, 233, 214, 125, 56, 170].

In the course of this dissertation, we use the generic functional architecture proposed by APTIV, Audi, Baidu, BMW, Continental, Daimler, FCA, HERE, Infineon, Intel, and Volkswagen [13]. It consists of the following functions:

- a *localization function*,
- a *sensor-fusion function*,
- an *ADS-mode-manager function*,
- an *interpretation-prediction function*,
- a *drive-planning function*,
- a *motion-control function*, and
- a *body-control function*.

The localization function is responsible for determining the current localization of the vehicle with respect to a map. Precisely locating the vehicle is essential since the vehicle

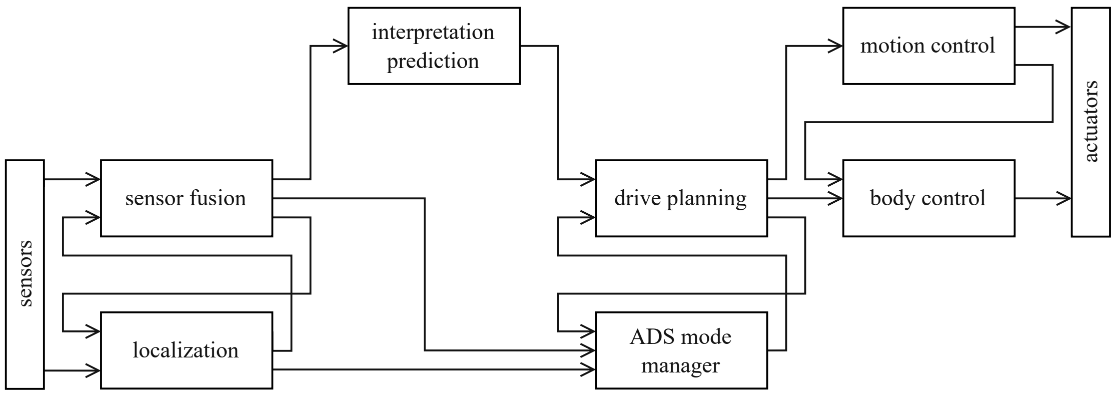


Figure 2.1: Functional system architecture of autonomous vehicles.

needs to know its position on the road and whether it is within the lane. Since some use cases require accuracy in the range of centimeters [16], using only the position data provided by a GPS sensor is not sufficient as the accuracy of GPS ranges above 20 meters [204]. However, the location accuracy can be increased by considering additional information, provided, for instance, by cameras, the internal measurement unit (which measures the linear accelerations and the vehicle angular), radar sensors, and lidar (light detection and ranging) sensors [137]. Furthermore, cooperative approaches which use, e.g., vehicle-to-vehicle (V2V) or vehicle-to-infrastructure (V2I) communication can further increase the accuracy and reliability of the localization.

The sensor-fusion function gets as input the information perceived by several different types of sensors the vehicle is equipped with, including e.g., cameras, radar sensors, lidar sensors, microphones, and ultrasonic sensors. Furthermore, the sensor-fusion function receives map and localization information from the localization function. The sensor-fusion function uses the received input to generate a *world model* which reflects the environment of the vehicle. This model includes, for instance, the detected objects and their classification. Fusing the information of different sensors allows for generating a world model that is more enhanced and more reliable than a world model that is only based on one sensor source [63].

The interpretation-prediction function gets the world model computed by the sensor-fusion function as input. Based on the current world model and the currently applicable traffic rules, the interpretation and prediction function forecasts the future state of the detected objects. For instance, this function determines the future state of other vehicles, cyclists, and pedestrians surrounding the autonomous vehicle.

The prediction of the future states of surrounding objects is required by the drive-planning function in order to determine lawful and safe trajectories that the autonomous vehicle can follow. Besides the output of the interpretation and prediction function, the drive planning function considers the current traffic rules and the motion of the vehicle.

The determined trajectory is then forwarded to the motion control function, which translates the trajectory into physical motion commands. These motion commands are then executed by the longitudinal and lateral actuators of the vehicle, which are referred to as *primary actuators*.

The determined motion control signals, as well as the trajectory determined by the planning function, are used as input for the body-control function. This function is responsible for controlling the so-called *secondary actuators* of the vehicle, including, for instance, the indicators, the headlights, the windscreen wiper, and the horn. An essential task of this function is to communicate the planned motion to other road users using visual and acoustic indications.

The ADS-mode-manager function is responsible for determining whether the vehicle can operate normally or must switch to a degraded mode. Switching to a degraded operation is, for instance, required in case a mechanical or electrical failure of the vehicle is detected, a passenger has not fastened the seat belt, or the vehicle has exceeded its *operational design domain*, which is also referred to as ODD. To determine whether a degraded operation is required, the ADS-mode-manager function considers the reported state of several monitors installed in the vehicle. The determined mode is used as an input for the drive-planning function.

2.2 General Overview of our Framework

As already stated earlier, Aptus follows a three-layered architecture design that was introduced by Gat et al. [70]. Each layer of Aptus, i.e., the context layer, the reconfiguration layer, and the component layer, includes several components that jointly perform the required task of that layer. Figure 2.2 illustrates the three logical layers of Aptus, including the interplay of their components.

In what follows, we discuss the individual layers of Aptus and their components.

2.2.1 Context Layer

The top layer of Aptus is the context layer. This layer, which is responsible for implementing the context-awareness property, extracts, based on the current context, software-architecture requirements. The extracted requirements define, for instance, the list of applications that shall be executed, their resource demands, as well as their required level of redundancy, diversity, and separation.

Figure 2.3 illustrates two use cases that show that distinct sets of contextual observations imply distinct sets of software-architecture requirements.

In the first use case, depicted in Figure 2.3a, 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 contextual observations, a set of required applications can be implied. This set may, for example, include an interpretation and prediction application that performs well in wintry weather conditions, a drive planning

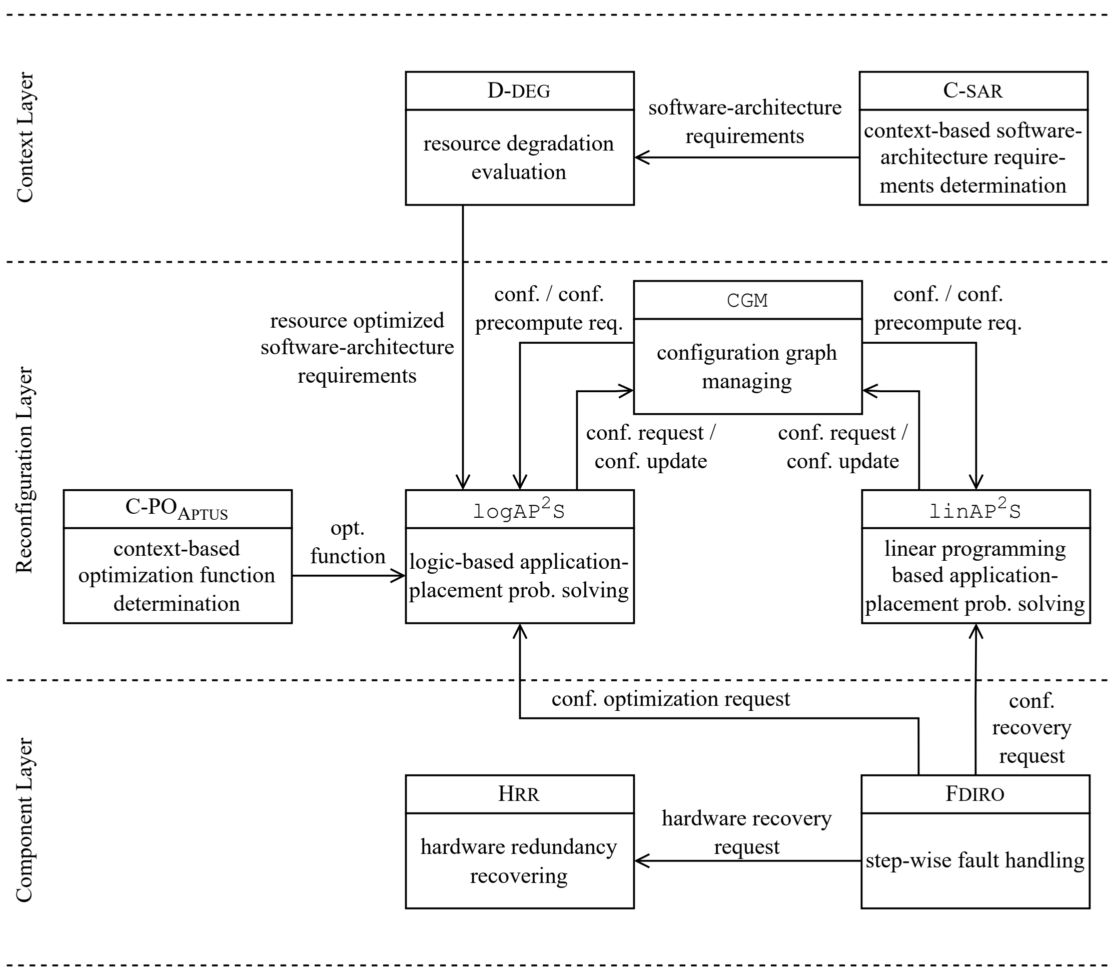
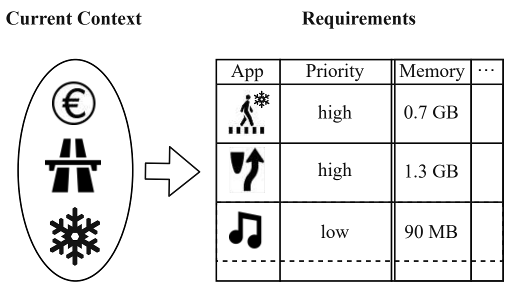

Context Layer
D-DEG
resource degradation evaluation
software-architecture requirements
C-SAR
context-based software-architecture requirements determination
Reconfiguration Layer
resource optimized software-architecture requirements
conf. / conf. precompute req.
CGM
configuration graph managing
conf. / conf. precompute req.
conf. request / conf. update
conf. request / conf. update
C-PO$_{\text{APTUS}}$
context-based optimization function determination
opt. function
logAP^{2}S
logic-based application-placement prob. solving
linAP^{2}S
linear programming based application-placement prob. solving
Component Layer
conf. optimization request
conf. recovery request
HRR
hardware redundancy recovering
hardware recovery request
FDIRO
step-wise fault handling

Figure 2.2: Overview of the three logical layers of APTUS and their components.

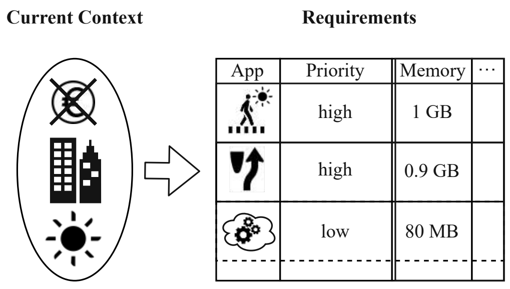
application, and entertainment applications that are included in the ride due to the booked premium package. Moreover, from the contextual observations, we can imply the safety-criticality of the respective applications and, thus, the priority, as well as other performance parameters.

The use case illustrated in Figure 2.3b, on the other hand, assumes a low-cost ride in an urban environment under good weather conditions. Consequently, the set of required applications includes, for instance, an interpretation and prediction that is optimized for good weather conditions and a drive planning application. Furthermore, we assume that a low-cost ride does not include onboard entertainment. Thus, the vehicle can provide its free resources, for instance, to a distributed cloud-based traffic optimization service.

The discussed use cases illustrate two challenges: (i) acquiring context information and (ii) implying software-architecture requirements.



(a) A premium ride on a highway in wintry conditions.



(b) A low-cost ride in an urban environment under good weather conditions.

Figure 2.3: Two use cases showing the correlation between contextual observations and software-architecture requirements. The two distinct scenarios imply a distinct set of requirements.

Acquiring contextual 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. We assume that the context acquisition is performed by the vehicle platform as various vehicle services may require context information. For instance, the current weather information may be provided to the passengers and may be recorded in log files.

The second task, implying software-architecture requirements from the current context, requires methods for specifying implication rules. In Aptus, this task is performed by C-sar and D-deg.

C-sar determines software architecture-requirements that reflect the needs of the current context. The computed software-architecture requirements are then sent to D-deg, which optimizes those requirements concerning the resource utilization. In particular, D-deg analyzes the data received by other vehicles and checks whether the redundancy requirements or the resource requirements of applications can be degraded. If this is the case, D-deg incorporates the degradation measures into the received software-architecture requirements. Finally, D-deg sends the adjusted software-architecture requirements to logAP^{2}S.

2.2.2 Reconfiguration Layer

The reconfiguration layer of Aptus is responsible for implementing the self-configuration and self-optimization property, as well as anticipativeness. To implement those properties, the reconfiguration layer is equipped with several components, namely: C-po$_{\text{Aptus}}$, logAP^{2}S, linAP^{2}S, and CGM.

The reconfiguration layer receives requests from the context layer and the component layer to compute new configurations. These requests are handled by logAP^{2}S and linAP^{2}S, which are application-placement problem solvers that support different properties. The former application-placement problem solver supports complex non-linear optimization functions at the cost of longer solving times. On the other hand, linAP^{2}S supports only one linear optimization function, which, however, leads to fast solving times.

The reconfiguration requests sent by the context layer are handled by logAP^{2}S. As described in the previous subsection, D-DEG transmits the determined software-architecture requirements to logAP^{2}S. Based on this input, logAP^{2}S computes a configuration that suits the current context. In order to achieve this, logAP^{2}S incorporates an optimization function that is determined by C-PO. This component analyzes the current context and specifies an optimization function that aims to enhance the vehicle's safety and reliability.

As stated before, the reconfiguration layer also receives requests from the component layer. In particular, FDIRO contacts linAP^{2}S to perform the recovery step and logAP^{2}S to execute the optimization step. The recovery step is handled by linAP^{2}S since this step requires that a configuration is provided fast. On the other hand, logAP^{2}S performs the optimization step, as this step requires applying complex optimization functions.

Note that before logAP^{2}S and linAP^{2}S start computing new configuration, they check whether CGM, which stores previously computed configurations, can provide the requested configuration. Furthermore, once logAP^{2}S and linAP^{2}S compute a new configuration, they transmit it to CGM, which persists the configuration.

The configurations that are determined by the reconfiguration layer are finally applied to the vehicle platform. Note that the mechanism for applying configurations strongly depends on the vehicle platform. Consequently, individual configuration-applying mechanisms need to be implemented for the individual vehicle platforms.

2.2.3 Component Layer

The component layer, which is the bottom layer of APTUS, is responsible for implementing the self-healing property. Generally speaking, this layer handles system-context changes that are caused by failing hardware and software components. To handle such system-context changes, the component layer incorporates FDIRO and HRR.

FDIRO [116, 119] implements a step-wise procedure to handle hardware and software faults. These steps include detecting and isolating the fault, recovering the software-architecture requirements, and optimizing the configuration. As mentioned before, FDIRO instructs logAP^{2}S and linAP^{2}S to execute the latter two steps.

Furthermore, FDIRO notifies HRR in case a hardware component experiences a failure. Subsequently, HRR initiates a recovery procedure for the malfunctioning hardware component, and if this recovery operation proves successful, the restored hardware component is subsequently again integrated into the system.

2.3 Related Approaches

The topic of self-managing systems in the context of autonomous vehicles is currently evolving, with only limited related work available so far. Nevertheless, some results have already been published, as discussed in what follows.

We first present general approaches for self-managing systems and afterwards we elaborate on self-managing approaches used in the automotive domain and other domains.

2.3.1 General Self-Managing Approaches

Three-Layered Reference Model for Self-Managed Systems

Generally speaking, the idea of a layered architecture to implement a self-managed system is not a new one. Kramer and Magee [135] proposed a general three-layered reference model for self-managed systems that is based on the three-layer architecture introduced by Gat [70] which is widely applied in the field of robotics. The proposed three-layered reference model consists of the so-called *component-control layer*, *change-management layer*, and *goal-management layer*.

At the component-control layer, which is the bottom layer, interconnected hardware and software components perform the desired function of the system. The component-control layer provides an interface to allow components to report their state to the upper layers. Furthermore, it implements interfaces to create new components, delete components, and adjust the behavior of components.

The change-management layer adapts the components of the bottom layer based on the reported component states and the plans received from the top layer. The actions that have to be executed to react to changing conditions are defined in predefined plans.

At the goal-management layer, which is the top layer, the general objectives and plans of the system to achieve these goals are computed. As an input, the goal-management layer takes, for instance, the current state of the system as well as a predefined description of high-level goals. If a new plan is computed, the changes are propagated to the change management layer.

The work of Kramer and Magee supports our approach to base Aptus on a layered architecture. However, their framework is rather general and not adapted to autonomous vehicles. With Aptus, we aim to provide a self-managing system solution specifically designed for autonomous vehicles.

Autonomic Managers

Another widely adopted architectural concept for self-managing systems was introduced by IBM [104]. The proposed concept introduces so-called *autonomic managers* to control resources, e.g., hardware and software components. A system can contain several autonomic managers. Additional autonomic managers may be introduced to orchestrate the individual autonomic managers.

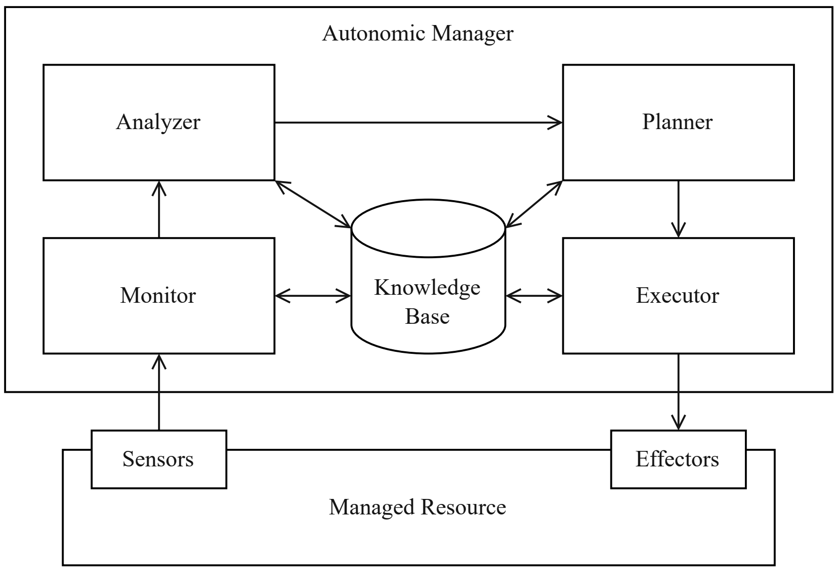


Figure 2.4: IBM's Autonomic manager consisting of a monitor, an analyzer, a planner, an executor, and a shared knowledge base. The resource that is controlled by the autonomic manager contains sensors and effectors.

An autonomic manager, as illustrated in Figure 2.4, implements a control loop consisting of a *monitor*, an *analyzer*, a *planner*, an *executor*, and a shared *knowledge base*. Furthermore, the resource that the autonomic manager controls implements a *sensor interface* and an *effector interface*. The sensor interface of the managed resource is used by the monitor to collect information from the managed resource, e.g., the current status, the configuration, and the available capacity. The information extracted by the monitoring function is in the next step processed by the analyzer, which determines whether further actions are required. For instance, if policies are violated, the analyzer can initiate further measures such as a reconfiguration. If further measures are required, the planner determines a procedure to adapt the managed resource to achieve the desired behavior. Finally, the executor schedules and carries out the determined plan using the effector interface of the managed resource.

The concept of autonomic managers aligns with the architecture of Aptus. Like autonomic managers, Aptus integrates a control loop to manage a resource. In particular, Aptus manages the software configuration of autonomous vehicles, enabling dynamic adjustments to changing conditions.

2.3.2 Self-Managing Approaches in the Automotive Domain

HAFLoop

Zavala et al. [226] introduced an extension of IBM's autonomic manager, referred to as HAFLoop ("highly adaptive feedback control loop"). A HAFLoop is characterised by a control loop consisting of a monitor, an analyzer, a planner, an executor, and a knowledge base that can be adapted at runtime. Their approach allows, for instance, the addition, removal, and substitution of both the control loop's elements and managed resources during runtime. Furthermore, HAFLoops support the adaptations of policies. Zavala et al. outline an example in which HAFLoops are used to improve the self-managing capabilities of autonomous vehicles. Moreover, in subsequent work [227], they illustrated in detail how HAFLoops can be used for implementing an adaptive monitoring behavior in autonomous vehicles. In their approach, Zavala et al. introduce two logical layers which contain HAFLoops: the lower-layer HAFLoops are responsible for adapting the hardware and software components of the autonomous vehicle, while the upper-layer HAFLoops are in charge of adapting the monitors of the lower-layer HAFLoops.

While the work of Zavala et al. is more general as Aptus, it does not provide a concrete implementation for the individual challenges of a self-managing system. Furthermore, Zavala et al. focus on adapting the monitor only, whereas Aptus considers adaptations of various involved elements. For instance, C-PO$_{\text{Aptus}}$ adapts the optimization function of `logAP`^{2}`S`, which correlates to the planning element of the HAFLoop.

Multi-Layered Control Architecture Approach

An alternative approach that builds on autonomic managers was introduced by Zeller et al. [228]. They presented a multi-layered control architecture approach that enables self-management in adaptive automotive systems. The idea of their approach is to group software components, system objectives, and requirements into so-called *clusters*, whereby an autonomic manager controls each cluster. For each cluster, an assignment, which is referred to as *cluster state*, between software components and computing nodes is generated. A cluster state is considered valid if all system objectives and requirements of the cluster are fulfilled. Each cluster continuously monitors if all system objectives and requirements are satisfied. A new cluster state must be computed if a system objective or requirement is not fulfilled.

The clusters are organized in a hierarchical structure that consists of multiple layers, i.e., a cluster can be a member of another cluster. If a cluster cannot find a cluster state which fulfills all requirements, the problem is propagated to a higher cluster. A cluster that is located in a higher layer inherits all software components, system objectives, and requirements of its affiliated subjacent clusters. Thus, the solution space for finding a valid cluster state is larger. Therefore, the probability of finding a valid cluster state increases as the layer number of clusters increases. However, a larger solution space also causes the problem complexity to rise, which, in return, leads to an increased computational effort. Nevertheless, Zeller et al. presented a simulation showing that the introduced cluster-based organization of software components, system objectives, and requirements allows reacting quickly to system changes. Furthermore, they argue that their approach can increase the dependability of adaptive automotive systems.

However, their presented simulation shows that in a few cases, the time for finding a valid cluster state is substantially longer than the solving time of the used reference implementation. Although the authors do not mention the reason for the outliers, they presumably result from propagations to higher layers. Especially in safety-critical situations, however, such a behavior is undesirable. In APTUS we counteract the issue of long solving times for the application-placement problem in safety-critical situations by using `linAP`^{2}`S`, which is specifically designed to compute placements quickly. Furthermore, `linAP`^{2}`S` employs a parallel solving heuristic that computes backup assignments which result from application-placement problems that relax some requirements. In case `linAP`^{2}`S` cannot find a valid assignment for the initial application-placement problem, one of the backup assignments is used.

A Self-Managing Framework for Safety

Some of the introduced self-managing approaches for autonomous vehicles are designed to improve a particular dependability parameter. An example of a self-managing framework for improving the safety of autonomous vehicles was introduced by Carré, Exposito, and Ibañez-Guzmán [43]. The presented framework consists of three components: the *safety orchestrator*, the *safety assessment processes*, and the *safety-oriented knowledge base*. The knowledge base contains, for instance, context information, safety constraints, and adaptation rules. Safety constraints are adapted at runtime according to the current context by the safety orchestrator. Each safety constraint that the safety orchestrator defines is monitored by safety assessment processes. For instance, a safety constraint while driving 30 km/h in an urban area might specify that the minimum distance between the vehicle and pedestrians must not fall below 5 meters. The safety assessment processes not only monitor the safety constraints but also adapt the behavior of the automated driving applications in accordance with the constraints.

In contrast to the safety-preserving mechanisms implemented in APTUS, the work of Carré, Exposito, and Ibañez-Guzmán focuses on adapting the functional properties of applications, e.g., the minimum required distance to specific objects. On the other hand, APTUS focuses on adapting software-architecture requirements, e.g., the level of redundancy of applications. These diverging approaches are, however, not contradictory but complement each other. In our work, we decided to focus on adapting the software-architecture requirements as such adaptations are less dependent on the concrete implementation of the applications executed by the vehicle. However, in future work, when APTUS is embedded in a concrete vehicle, it is conceivable that our framework is extended by an approach similar to the one presented by Carré, Exposito, and Ibañez-Guzmán.

An Approach for Restricting the Operational Design Domain

Colwell et al. [50] introduced a concept that is based on self-awareness and fault tolerance to increase the safety and reliability of autonomous vehicles. Their approach is based on the idea of restricting the ODD of an autonomous vehicle at runtime. The restricted operational design domain (ROD) is a superset of the ODD. Unlike the ODD, which is static, the ROD is updated during the runtime of the vehicle. The ROD of a vehicle is, for instance, adapted if a hardware or software component fails. Such an adjustment of the ROD entails that subsystems of the automated driving system have to adapt to the new restrictions. For example, the planning subsystem can adapt its planning procedure to avoid high-speed roads if the maximum speed is restricted because of a sensor fault. As a consequence of the performed degradation procedures, the vehicle remains operational and safe. Colwell et al. introduce a so-called *ROD monitor* to ensure that subsystems of the automated driving system adhere to the ODD restrictions. If the ROD monitor detects a violation of the ROD, a procedure that transfers the vehicle to a minimal-risk condition is performed.

Compared to Aptus, a limitation of the concept introduced by Colwell et al. is that no mechanisms are implemented to recover the full functionality of the vehicle after the ROD is extended. Aptus, on the other hand, includes Fdiro and Hrr, which both aim to recover the initial functionality of the vehicle in case of software and hardware faults. Nevertheless, restricting the ODD of the vehicle in case of faults is desirable. In the current version of Aptus, we decided not to integrate such an approach since deep knowledge about the applications executed by the vehicle is required. However, in future extensions of Aptus, it is conceivable that such functionality as introduced by Colwell et al. is added.

An Approach to Adapt the Vehicle Detection

Hemmati et al. [89, 90] argue that in different lighting conditions, different features are used to detect other vehicles. To address this, they presented an approach that uses three implementations to detect vehicles during day, dusk, and night. Based on the current ambient light, a reconfiguration unit decides which vehicle-detection implementation shall be executed. For detecting vehicles during the day and dusk, the same model is used. However, two different training sets were used to train the models. On the other hand, the detection that is used during the night uses another detection algorithm. Their results show that the reconfiguration can be completed in 20 msec.

While the approach introduced by Hemmati et al. focuses on adapting vehicle detection, Aptus introduces a generic reconfiguration method that takes into account the current context.

A Context-dependent Approach to Adapt Application Parameters

Horcas et al. [96] presented an approach to dynamically adjust the parameters of a car-following model at runtime based on different weather conditions. A car-following model defines how the lateral behavior of the vehicle is adapted in reference to a leading vehicle. According to Horcas et al., the distance to the lead vehicle and the maximum allowed acceleration are the most crucial parameters in a car-following model that need to be adapted in case the weather conditions change. In their work, they showed that their parameter reconfiguration approach improves the safety as well as the traffic flow.

Similar to the previously discussed approach by Hemmati et al. [89, 90], this approach focuses on reconfiguring a specific application. Aptus, in contrast, provides a generic reconfiguration approach.

Context-Aware Reconfiguration Approaches for Non-Autonomous Vehicles

Self-managing systems are, of course, not limited to autonomous vehicles. Also non-autonomous vehicles can profit from self-managing capabilities. Indeed, an approach in this regard is put forth by Weiss and Struss [218], who discuss a context-aware reconfiguration approach that activates and deactivates advanced driver assistance systems, e.g., adaptive cruise control, traffic sign recognition, and parking assistance, according to current context information. A generic context model based on ontology and object-oriented modeling is introduced to infer which driver assistance systems are required in which situation. The authors illustrate in a simulation that their approach can significantly reduce the number of active driver assistance systems. In the conducted experiment, the average number of active functions was reduced by about one quarter compared to a system that continuously executes all driver assistant systems.

In Aptus, C-sar determines which functions shall be activated in a given context. Unlike the approach of Weiss and Struss, C-sar is specially designed for use in autonomous vehicles. Furthermore, the context model applied by Weiss and Struss is less exhaustive than the one used in C-sar. For instance, the context model applied by C-sar allows modeling relationships between functions and operational contexts as well as user contexts. Weiss and Struss, on the other hand, only consider environmental-context information and the speed of the vehicle.

Another context-aware reconfiguration approach for non-autonomous vehicles was presented by Panagiotopoulos and Dimitrakopoulos [178]. In their work, they focus on the adaption of the driving style based on the current context. For selecting the most suitable driving style, their approach takes into account several parameters, including, the age of the driver, the preferred comfort, the current road type, and the current traffic situation. Finally, the selection of the driving style leads to an adjustment of, for instance, the speed, the suspension, and the gear ratios.

Adapting the driving style based on contextual observations is possible in Aptus by introducing applications that implement different driving styles. To then instruct `logAP`^{2}`S` to select a certain application in a given context, the relation between the application and the context needs to be modeled.

2.3.3 Self-Managing Approaches in other Domains

Besides the automotive field, also other domains, including, for instance, the aviation field, require self-managing systems. For example, in the EU-funded project SCARLETT[3] ("scalable and reconfigurable electronics platforms and tools"), an approach for adding

[3]`https://cordis.europa.eu/project/id/211439`.

reconfiguration capabilities to the next generation of the *Integrated Modular Avionics* platform (IMA-1G) was developed [33]. The proposed reconfiguration approach aims, inter alia, to improve reliability and reduce unscheduled maintenance work. It focuses on reallocating applications to spare nodes if their original nodes fail. The authors introduced a so-called *reconfiguration supervisor* that is responsible for planning and executing the reconfiguration plan. A reconfiguration is triggered by the so-called *monitoring and fault detection function*, which continuously monitors the system and reports faults to the reconfiguration supervisor. The reconfiguration is performed according to a policy that is defined at design time. For instance, the policy can specify that safety-critical aviation applications are prioritized over non-safety relevant applications, e.g., applications of the in-flight entertainment system, in the event of a configuration.

López-Jaquero et al. [154] introduced a similar approach for dynamic reconfiguration in avionic architectures, which are based on the ARINC 653 standard [4]. They introduce a separate intermediary layer called *redundancy and reconfiguration management layer* (RRML), which is located between the real-time operating system and the application execution layer. Furthermore, López-Jaquero et al. present an error detection approach that is based on trust values. The trust value of a component decreases if the implemented voting mechanism detects that the output deviates from the output provided by the redundant components. If the trust value of a component falls below a specified threshold, the component will be isolated.

To conclude, with Aptus we aim to introduce a comprehensive self-managing approach for autonomous vehicles. Aptus contains interconnected approaches that implement the individual self-managing properties. To the best of our knowledge, no concept has been published so far in the domain of autonomous driving that implements self-managing properties to a comparable degree as Aptus.

CHAPTER 3

The Context Layer

In this chapter, we describe the context layer of Aptus. This layer is responsible for implementing the context-awareness property and consists of two components, viz. C-sar and D-deg. C-sar is responsible for determining software-architecture requirements based on the current context, while D-deg optimizes the software-architecture requirements by analyzing data received by other vehicles.

As the design of the context layer is based on the general architecture of context-aware systems, we present in the first part of this chapter general information about context-aware computing. Furthermore, we also introduce our definition of context that is used throughout this dissertation. Next, we recapitulate some basic elements of answer-set programming (ASP) [78, 79] since it is a key element of C-sar. Afterwards, we present C-sar and D-deg. The chapter concludes with a brief summary.

3.1 Context-Aware Computing

Context-aware computing was first introduced by Schilit, Adams, and Want [193] in the field of mobile distributed computing in the 1990s. According to their definition, *context-aware systems* can recognize context changes and can perform adaptation actions that aim to respect the present context. Furthermore, Schilit, Adams, and Want define that context information includes, for instance, the user's location, other nearby people, the available devices, the network connectivity, the noise level, and the lightning conditions. This information can be grouped into computing, user, and physical context [44]. However, several other works have introduced new definitions of the term "context" [190, 44, 41].

In the first part of this section, we list different definitions of the concept of a context. Furthermore, we introduce our definition of a context. Next, we provide an overview of the general architecture of context-aware systems.

3.1.1 Definition of Context

Abowd et al. [2] introduced a rather general definition. They specify context as any information that characterizes the situation of a person, place, or object, which influences the interaction between a user and an application.

In the domain of driver assistance systems, context is often defined as the information that characterizes a driving situation, including, e.g., the road conditions, traffic objects, and traffic rules [68, 212]. However, in the scope of our work, this definition is unsuitable as we consider additional information apart from the driving situation.

Therefore, we define context as any information relevant to the vehicle's system architecture, whereby the context information are categorized into:

- *environmental context*,
- *operational context*,
- *user context*, and
- *system context*.

The environmental context contains information about the external conditions of the vehicle. This includes, for instance, the roadway type, the weather conditions, information about the infrastructure, speed ranges, and the country in which the vehicle operates. These attributes are also considered when specifying the ODD [131, 189, 211]. The ODD defines the external circumstances under which an autonomous vehicle can operate safely.

The operational context describes how the vehicle is operated. We refer to the manner in which the vehicle is operated as the *vehicle-operation mode*. Conceivable vehicle-operation modes are, for instance, fully autonomous operation, manual operation, test operation, and parking. Furthermore, the operation state can be supplemented by properties which describe, for example, whether the vehicle is operated commercially or whether passengers are in the vehicle.

Parameters that describe the requirements of the vehicle users are contained in the user context. Considering an autonomous taxi, the user context, for instance, contains information about whether the user booked a premium ride or a low-budget ride. Likewise, in the case of a personally owned vehicle, the user context can hold information about, e.g., the type of the ordered entertainment package. Furthermore, if the vehicle supports manual operation, the user context includes whether the driver requested assistance services such as lane-keeping, parking, and emergency-brake assistance.

The system context includes information about the vehicle system, i.e., the hardware and software of the vehicle. Details about the currently executed applications, the available hardware resources, and occurring hardware and software errors are, for instance, contained in the system context.

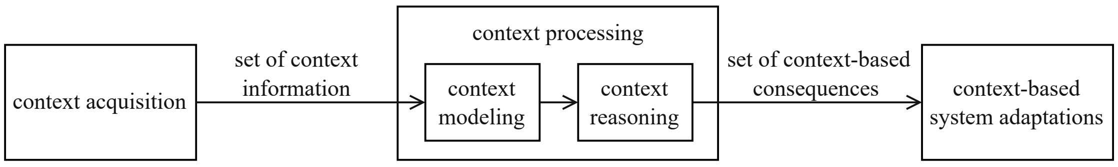

context acquisition
set of context information
context processing
context modeling
context reasoning
set of context-based consequences
context-based system adaptations

Figure 3.1: The three main tasks implemented by a context-aware system, i.e., context acquisition, context processing, and context-based system adaptations.

3.1.2 The General Architecture of Context-Aware Systems

In general, as illustrated in Figure 3.1, context-aware systems implement the following three tasks sequentially [222, 212, 199, 180]:

- *context acquisition*,
- *context processing*, and
- *context-based system adaptations*.

In the context acquisition phase, data provided by sensors or software services is collected. For instance, a rain sensor can be used to determine whether it is raining. On the other hand, a human-machine interface (HMI) service can provide information about the user's requirements.

In the next phase, the collected context information is processed, whereby the context processing is subdivided into

- the *context modeling phase*, and
- the *context reasoning phase*.

Before consequences can be deduced, the context has to be represented in a suitable context model. According to Bettini et al. [29], object-role based, spatial, and ontology-based models are the most prominent approaches for modeling context information.

Object-role modelling (ORM) [87] is a graphical fact-based modeling technique. Facts are represented as nodes, and edges correspond to relationships between the nodes. Models like the *context modelling language* (CML) [92] support, for example, SQL-like queries and reasoning over uncertain information.

Spatial models, like the *Nexus augmented world model* [173], are especially suited for modeling location-based context information. Furthermore, such models allow reasoning about spatial relations.

Ontology-based models employ formal ontology language to express context information. They support formal context reasoning methods by specifying axioms and constraints [223].

Many ontology-based models, including the *standard ontology for ubiquitous and pervasive applications* (SOUPA) [45] and the *CONtext ONtology* (CONON) [84], are based on the *web ontology language* (OWL) [101].

Once the context information is processed, system adaptations are performed in the last phase. Consequences of this phase can be, for instance, an adaptation of the user function, the HMI, and the system architecture.

3.2 Answer-Set Programming

Answer-set programming (ASP) is a declarative programming paradigm used to solve search problems [19, 73, 77]. ASP has been, for instance, used in the domain of robotics [60], natural-language processing [20], and semantic-web reasoning [59].

3.2.1 Syntax of Answer-Set Programs

The basic syntax of answer-set programs is defined over a first-order language, containing *predicate symbols*, *variables*, and *constants*, and constituting the following elements: A *term* is either a variable or a constant, whereby variables are denoted by capital characters and constants by lower-case letters. An *atom* is an expression of the form

$$\texttt{p(t}_1\texttt{,...,t}_n\texttt{)},$$

where $\texttt{p}$ is a predicate symbol and $\texttt{t}_1,\ldots,\texttt{t}_n$ are terms. A *literal* is an atom that is possibly preceded by the *strong negation* symbol "$-$", whereby an atom is a factual statement that is either true or false. A *rule* is an ordered pair of the form

$$\texttt{a}_1 \vee \cdots \vee \texttt{a}_m \texttt{ :- b}_1\texttt{,...,b}_k\texttt{,not b}_{k+1}\texttt{,...,not b}_n\texttt{.}, \tag{3.1}$$

where $\texttt{a}_1,\ldots,\texttt{a}_m,\texttt{b}_1,\ldots,\texttt{b}_n$ are literals, "$\vee$" denotes *disjunction*, and "$\texttt{not}$" stands for *default negation*. We refer to $\texttt{a}_1\vee\cdots\vee\texttt{a}_m$ as the *head* and $\texttt{b}_1,\ldots,\texttt{b}_k,\texttt{not b}_{k+1},\ldots,\texttt{not b}_n$ as the *body* of the rule. The body in turn is subdivided into the *positive* and the *negative* body, given by $\texttt{b}_1,\ldots,\texttt{b}_k$ and $\texttt{not b}_{k+1},\ldots,\texttt{not b}_n$, respectively. The intuitive meaning of rule (3.1) is that if $\texttt{b}_1,\ldots,\texttt{b}_k$ are all derivable but none of $\texttt{b}_{k+1},\ldots,\texttt{b}_n$ is derivable, then at least one literal from the head is asserted. Finally, an *answer-set program*, or *program* for short (if no confusion will arise), is a finite set of rules.

Both the head as well as the body of a rule might be empty. In the former case, the resulting rule is called a *fact*, whilst in the latter case, the rule is referred to as *constraint*.

A literal is called *ground* if it does not contain any variables. Otherwise, the literal is referred to as *non-ground*. Likewise, rules, as well as programs that only contain ground literals, are ground and otherwise non-ground.

A non-ground program can be transferred into a ground program by replacing all variables of a rule in a uniform manner with the constants appearing in the program (or a fresh constant if the program does not contain one). This process is called *grounding*. For a program P, the grounding of P is denoted by $grnd(P)$.

3.2.2 Semantics of Answer-Set Programs

A set I of ground literals that does not contain any atom $\mathtt{a}$ such that $\{\mathtt{a}, \mathtt{-a}\} \subseteq I$ is referred to as *interpretation*. A ground atom $\mathtt{a}$ is (i) *true* (*under* I) in case $\mathtt{a} \in I$, (ii) *false* (under I) if $\mathtt{-a} \in I$, and (iii) *undefined* otherwise. A ground literal $\mathtt{-a}$ is (i) true (under I) if $\mathtt{a}$ is false (under I), (ii) false (under I) if $\mathtt{a}$ is true (under I), and (iii) undefined otherwise. Moreover, a default negated ground literal $\mathtt{not\ l}$ is true (under I) if $\mathtt{l} \notin I$ and otherwise false (under I).

An interpretation I is called a *model* of a ground rule of the form (3.1) if whenever $\{\mathtt{b}_1, \dots, \mathtt{b}_\mathtt{k}\} \subseteq I$ and $\{\mathtt{b}_{\mathtt{k+1}}, \dots, \mathtt{b}_\mathtt{n}\} \cap I = \emptyset$ holds, then also $\{\mathtt{a}_1, \dots, \mathtt{a}_\mathtt{m}\} \cap I \neq \emptyset$ holds, i.e, if the body of the rule is true under I, then also the head is true under I. In case an interpretation I is a model of all rules of a ground program P, then I is called a *model* of P.

The semantics of answer-set programs is given in terms of *answer sets*, which are defined as minimal models of the so-called *Gelfond-Lifschitz reduct* [78, 79], which is determined as follows: Given a ground program P and an interpretation I, the Gelfond-Lifschitz reduct, or simply *reduct*, P^I, of P relative to I is the "not"-free program

$$\{\mathtt{a}_1 \vee \dots \vee \mathtt{a}_\mathtt{m} \mathtt{\ :-\ } \mathtt{b}_1\mathtt{,\ \dots,} \mathtt{b}_\mathtt{k} \mid \mathtt{a}_1 \vee \dots \vee \mathtt{a}_\mathtt{m} \mathtt{\ :-\ } \mathtt{b}_1\mathtt{,\ \dots,} \mathtt{b}_\mathtt{k}\mathtt{,}$$
$$\mathtt{not\ b}_{\mathtt{k+1}}\mathtt{,\ \dots,not\ b}_\mathtt{n} \in P, \mathtt{b}_{\mathtt{k+1}}, \dots, \mathtt{b}_\mathtt{n}\} \cap I = \emptyset\}.$$

Then, I is said to be an answer set of P if I is a minimal model of P^I, i.e., if no other interpretation $J \subset I$ is also a model of P^I. An interpretation I is a minimal model of P^I if no other interpretation $J \subset I$ is also a model of P^I. For an arbitrary, not necessarily ground program P, I is an answer set of P if it is an answer set of $grnd(P)^I$.

Prominent solvers for computing answer sets are DLV [141] and clasp [75]. In the context of this dissertation, we employ the ASP system clingo [72], which uses GrinGo [76] as a grounder and clasp as a solver. We will employ the latter for our subsequent purposes.

The solver clasp supports, like most answer-set solvers, so-called *aggregates* [8], which allow defining properties of a set of atoms. An aggregate in clasp has the following general syntax:

$$\mathtt{s}_1 \prec \alpha\{\mathrm{t}_1\mathtt{:}\mathrm{l}_1\mathtt{;\ \dots;}\mathrm{t}_n\mathtt{:}\mathrm{l}_n\} \prec' \mathtt{s}_2. \tag{3.2}$$

Here, $\mathrm{t}_1, \dots, \mathrm{t}_n$ are list of terms, $\mathrm{l}_1, \dots, \mathrm{l}_n$ are list of literals, α is an *aggregate function*, $\prec$ and $\prec'$ are arithmetic comparison relations (<, <=, =, !=, >=, and >), and $\mathtt{s}_1$ as well as $\mathtt{s}_2$ are terms. Note that one of "$\mathtt{s}_1 \prec$" and "$\prec' \mathtt{s}_2$" may be omitted for expressing just a single relation, e.g., an upper bound or a lower bound.

The intuitive meaning of (3.2) is that the aggregate function α is applied to the first elements in the multiset consisting of term tuples which correspond to those lists of terms for which the corresponding list of literals hold. The result is then compared to $\mathtt{s}_1$ and $\mathtt{s}_2$ using the arithmetic operations $\prec$ and $\prec'$, respectively.

The aggregate functions used in this dissertation are #count, #sum, #max, and #min. The function #count returns the cardinality of a multiset, while #sum returns the sum all elements in it. The functions `#max` and `#min` return the numerical maximum and minimum of a multiset, respectively.

The grounding of rules with aggregates is based on the notions of *global* and *local variables*. While the former kind of variables are those occurring in at least one literal not involved in any aggregation, the other variables are local to the aggregate element they occur in. A ground instance of a rule is then obtained by replacing global variables with ground terms, given a fixed set, and then expanding the local variables in each aggregate. For more details on programs with aggregates, including their precise semantics, we refer to the clingo documentation [72] and the paper by Alviano and Faber [8].

We also make use of *optimization statements*, as supported by clasp [74], which enable the definition of desired properties of an optimal answer set. Syntactically, an optimization statements is an expression of the form

$$\omega\{\mathtt{w}_1@\mathtt{p}_1,\mathrm{t}_1:\mathrm{l}_1;\ldots;\mathtt{w}_n@\mathtt{p}_n,\mathrm{t}_n:\mathrm{l}_n\}.,$$

where $\mathrm{t}_1,\ldots,\mathrm{t}_n$ and $\mathrm{l}_1,\ldots,\mathrm{l}_n$ are as in (3.2), $\mathtt{w}_1,\ldots,\mathtt{w}_n$ are integer terms called *weights*, $\mathtt{p}_1,\ldots,\mathtt{p}_n$ are integer terms called *priorities*, and ω is an *optimization statement*.

In particular, clasp admits the optimization statements `minimize` and `maximize`. These statements aim to find the answer set that either minimizes or maximizes the weights, whereby only the weights and tuples of terms for which the corresponding literal tuples hold are considered. If multiple optimization statements are needed, priorities can be introduced, where higher priority levels take precedence over lower ones.

Finally, another language variant we employ are *cardinality constraints* [200], which are aggregate atoms using counting as their operation. More specifically, a cardinality constraint is an expression of the form

$$\mathtt{l} \prec \{\mathtt{a}_1:\mathrm{l}_1;\ldots;\mathtt{a}_n:\mathrm{l}_n\} \prec' \mathtt{u},$$

where $\mathtt{a}_1,\ldots,\mathtt{a}_n$ are atoms, $\mathrm{l}_1,\ldots,\mathrm{l}_n$ are lists of literals, $\prec$ and $\prec'$ are arithmetic comparison operators (<, <=, =, !=, >=, and >), and `l` as well as `u` are integer values representing the lower bound and upper bound, respectively.

The intuitive meaning of a cardinality constraint is that a subset S of the atoms, for which the corresponding tuple of literals hold, is selected. Furthermore, the cardinality of S must be between the lower bound `l` and the upper bound `u` while respecting the arithmetic comparators $\prec_1$ and $\prec_2$.

3.3 C-sar: A Tool to Determine Context-Based Software-Architecture Requirements

We now describe C-sar, our tool for determining which requirements the software architecture of an autonomous vehicle has to fulfill so that it suits the current context.

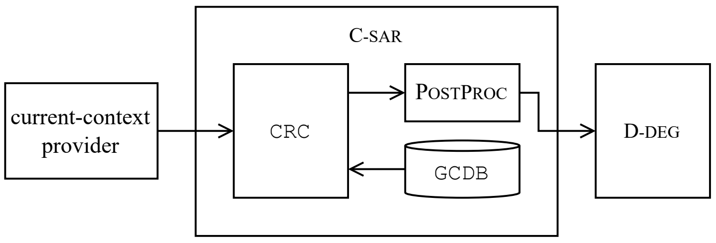


Figure 3.2: The inputs and and the output of C-SAR. Based on the current environment, operational and user-context information, as well as, the context attributes, C-SAR determines requirements for the software architecture.

As stated already, C-SAR uses ASP to encode and calculate the underlying reasoning tasks while the overall program itself is written in Python.

C-SAR follows the general architecture of context-aware systems as described in Subsection 3.1.2. In particular, as illustrated in Figure 3.2, C-SAR consist of the *general context-database*, GCDB, the *context-reasoning component*, CRC, and the *post-processing component*, PostProc.

These components serve the following purposes:

- GCDB specifies the interconnection of context information the vehicle can generally experience. This information is provided at design time by the vehicle engineers and might be updated, e.g., in case software updates are performed, the ODD is changed, and the functional scope of the vehicle is adjusted.
- CRC determines, on the basis of the general context model as given by GCDB and the current context as provided by the *current-context provider*, functions which are requested by the current context as well as a collection of applications which can feasibly execute these functions.
- The postprocessor PostProc adds applications that shall be executed as redundant instances and provides the output of C-SAR to D-DEG which is responsible for optimizing the received software-architecture requirements in terms of its resource utilization.

CRC and GCDB are implemented in terms of two answer-set programs, $P_{\mathtt{CRC}}$ and $P_{\mathtt{GCDB}}$. As well, the current context is also represented by an answer-set program, viz. P_{CC}. To embed these answer-set programs into C-SAR, we use clyngor[1], a Python wrapper for the ASP system clingo.

Unlike the other subcomponents, PostProc is implemented in Python. The reason for this is that PostProc requires the implementation of a multilevel sort algorithm which can be realized more efficiently in Python than in ASP.

[1]clyngor: `https://github.com/Aluriak/clyngor`

The output of CRC is the first computed answer set, A_{SAR}, of the program

$$P_{CC} \cup P_{\texttt{GCDB}} \cup P_{\texttt{CRC}},$$

containing the software-architecture requirements that need to be fulfilled in the current context. In order to deterministically select an answer set, CRC selects the first one. This deterministic selection allows reducing the size of the configuration graph introduced in Section 4.2, as otherwise a non-deterministic selection would result in multiple configurations for the same contextual observations. In case no answer set is found, the vehicle is transferred into a safe state. Then, A_{SAR} is taken by PostProc as input and extends it to A'_{SAR} by adding applications that are executed as redundant instances, as mentioned before. Afterwards, A'_{SAR} is handed over to the application-placement problem solver logAP^{2}S.

In what follows, we first give an overview of the different context providers. Next, we give details on the components of C-sar—in particular, on the answer-set programs $P_{\texttt{GCDB}}$ and $P_{\texttt{CRC}}$ of the context database GCDB and the context reasoning component CRC.

3.3.1 The General Context-Database GCDB

To structure the context information stored by GCDB, we define a context model which specifies the following entities:

- *function*,
- *application*,
- *supporting software*,
- *optimization function*,
- *vehicle-operation mode*,
- *operation property*,
- *user context*,
- *environmental-context category*,
- *environmental-context value*, and
- *environmental-context set*.

These entities, their attributes, as well as their relations are illustrated in Figure 3.3 and Figure 3.4. Each entity is uniquely identified by its name.

The entities were selected to describe the relevant information of the environmental context, the operational context, the user context, and the system context that is required by C-SAR in order to determine software-architecture requirements. In particular, functions, applications, and supporting software are part of the system context as they define the software architecture. Optimization functions are also part of the system context, as these functions are used by the reconfiguration layer to optimize configurations according to the needs of the current context. Furthermore, vehicle-operation modes and operation properties comprise the operational context, while the user context is represented by the entity of the same name. Lastly, environmental-context categories, environmental-context values, and environmental-context sets define the environmental context.

GCDB is defined by the answer-set program P_{GCDB}. This program, in turn, is given by $DB_{sys} \cup DB_{op} \cup DB_{usr} \cup DB_{env}$, where its subprograms, detailed below, correspond to the introduced four context categories, containing the context information in terms of facts. Note that we decided to split P_{GCDB} into these four subprograms in order to improve the readability of the subsequent subsections.

The Database DB_{sys}

Functions, applications, supporting software, and optimization functions hold system-context information.

We define a *function* as a task, e.g., localization, sensor fusion, or motion control, that is implemented by an application. In C-SAR, functions are defined using the predicate `function/1`, where its argument defines the task's unique name:

```
function(localization).
function(sensor_fusion).
function(ads_mode_manager).
function(interpretation_prediction).
function(drive_planning).
function(motion_control).
function(body_control).
function(traffic_optimization).
function(ride_management).
function(update_management).
function(shared_event_recording).
function(logging).
function(ride_visualization).
function(entertainment).
```

Note that the first seven facts define the functions which are, as discussed in Section 2.1 (cf. page 10), required for autonomous driving. On the other hand, the latter seven example functions, viz.,

- `traffic_optimization`,

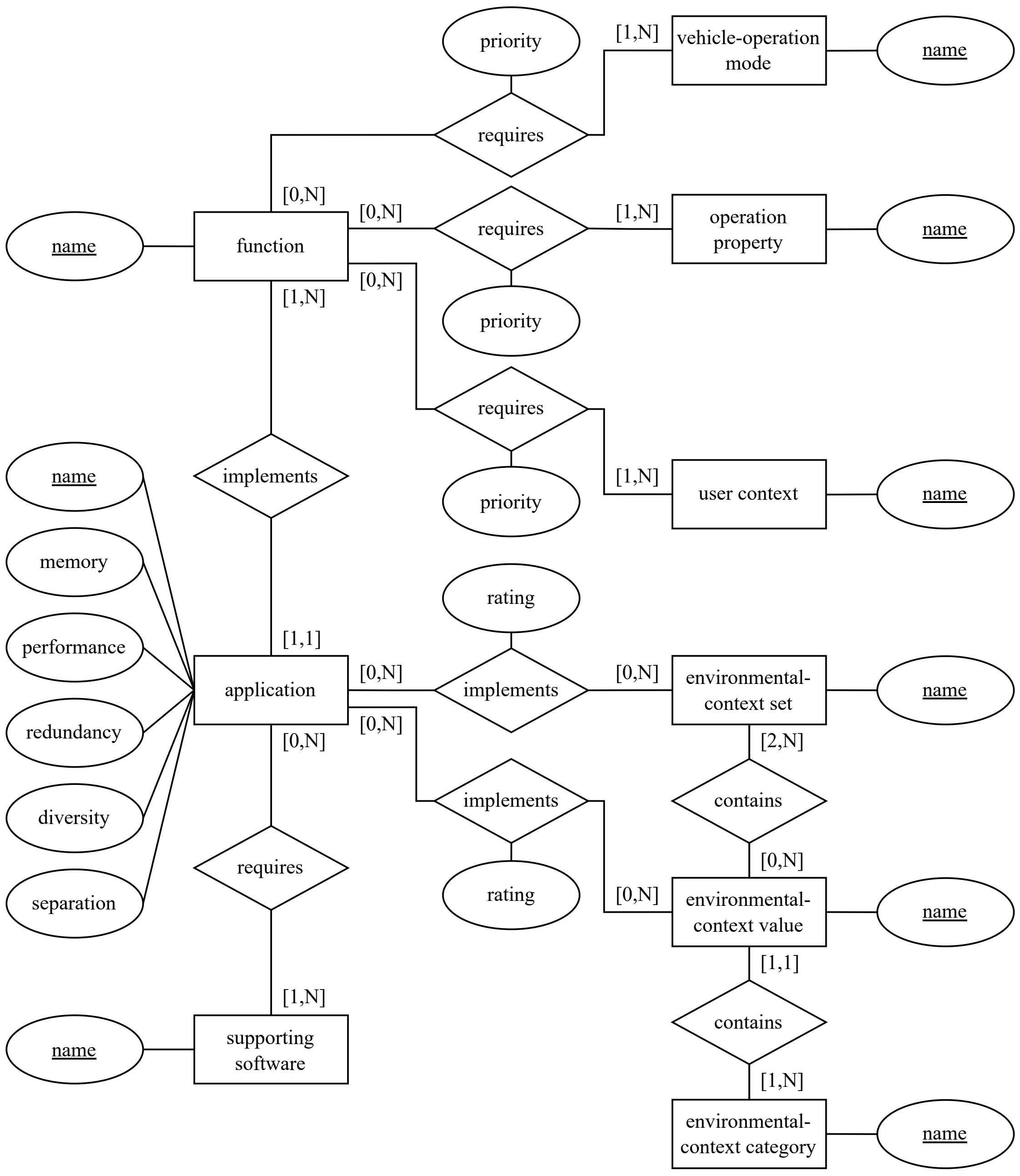


Figure 3.3: The entity-relationship diagram of the context model employed by C-sar. The diagram applies the Chen notation [46], whereby the cardinalities are expressed using the Min/Max notation proposed by Abrial [3].

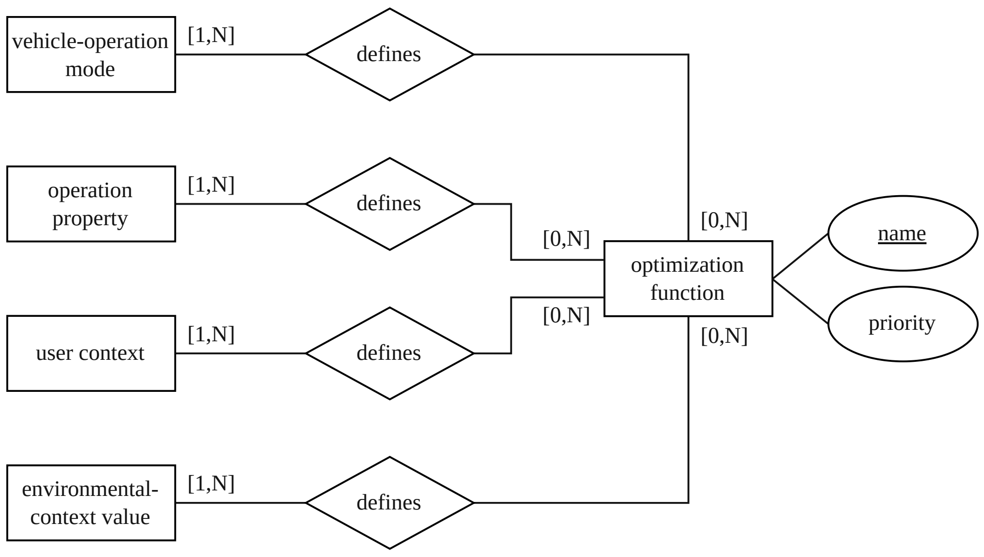


Figure 3.4: Continuation of the entity-relationship diagram illustrated in Figure 3.3.

- `ride_management`,
- `update_management`,
- `shared_event_recording`,
- `logging`,
- `ride_visualization`, and
- `entertainment`,

which are described in Table 3.1, do not interfere with the driving task.

As specified before, functions are implemented by *applications*, i.e., an application is defined as a concrete implementation of a function. Furthermore, we define that multiple diverse applications can implement the same function. However, we assume that one application implements only one function.

Applications and their relation to a specific function are modeled using the predicate `application/2`. Its first argument specifies the name of the application, while the second argument defines the function the application implements. For instance, three applications that implement the sensor-fusion function are specified as follows:

```
application(fus1, sensor_fusion).
application(fus2, sensor_fusion).
application(fus3, sensor_fusion).
```

Table 3.1: Description of the traffic optimization, ride management, update management, shared event-recording, logging, ride visualization, and entertainment function.

Function	**Description**
`traffic_optimization`	The traffic-optimization function computes tasks, which are assigned by a global traffic-management system [174]. The traffic-management system utilizes the computing resources of multiple vehicles to optimize the traffic flow.
`ride_management`	In case of an autonomous taxi, the ride-management function is, for instance, responsible to process incoming ride requests from customers, and to communication the current location and status of the vehicle to the taxi fleet's command center.
`update_management`	The update-management function is responsible to plan, execute, and monitor updates of applications.
`shared_event_recording`	The shared event-recording function, as, e.g., introduced by Guo, Meamari, and Shen [86], records accidents in a blockchain.
`logging`	The logging function records all events in the system. The resulting logs are used by developers to improve the system.
`ride_visualization`	The ride-visualization function, e.g., as proposed by Lungaro, Tollmar, and Beelen [157], aims to improve the user's trust by illustrating the perception and the planned actions in the vehicle cabin.
`entertainment`	The entertainment function allows passengers of the vehicle, e.g., to play games, stream movies, and listen to music.

Applications have different demands regarding their execution. The most relevant application parameters for Aptus are:

- the memory demand,
- the performance demand,
- the level of redundancy,

- the level of diversity,
- the level of separation, and
- a list of required supporting software.

The memory demand, which is given in Megabytes, defines the minimum amount of memory the application requires in order to execute without limitations. Likewise, the performance demand specifies the minimum TFLOPS (i.e., 10^{12} floating point operations per second) an application requests. The level of redundancy specifies the minimum number of redundant applications which implement the same function. Furthermore, the level of diversity defines the number of redundant applications that have to be diverse, and the level of separation defines on how many different computing nodes the applications have to be executed.

The memory demand, the performance demand, the level of redundancy, the level of diversity, and the level of separation are specified by the predicates `memory/2`, `performance/2`, `redundancy/2`, `diversity/2`, and `separation/2`, respectively. The first argument of these attributes indicates the application, and the second argument defines the parameter value. For example, the application demands of the sensor-fusion application `fus1` are given as follows:

```
memory(fus1, 4000).
performance(fus1, 130).
redundancy(fus1, 2).
diversity(fus1, 1).
separation(fus1, 2).
```

Besides the before-mentioned application parameters, a set of required supporting software can be defined for each application. This allows, for instance, to express that a particular application requires a specific runtime environment like Python or Java, and particular libraries such as CUDA[2].

Software that supports applications is modeled as a separate entity by the predicate `supporting_software/1`, as multiple applications can share the same requirements. The argument of this predicate defines the name of the supporting software.

To indicate whether an application requires a particular supporting software, we use the predicate `req_supporting_software/2`. The first argument of this predicate defines the application, and the second argument specifies the supporting software that this application demands in order to be executed. The required supporting software of the sensor fusion application `fus1` is, for example, specified as follows:

```
supporting_software(cuda).
req_supporting_software(fus1, cuda).
```

[2]NVIDIA CUDA Toolkit: `https://developer.nvidia.com/cuda-toolkit`

As mentioned before, DB_{sys} also defines optimization functions besides functions, applications, and supporting software. The application-placement problem solver `logAP`^{2}`S`, which receives the software-architecture requirements determined by C-sar, has potentially multiple options to implement these requirements. Consequently, in order to indicate which implementation of the determined software-architecture requirements is desired most, optimization functions are required.

We introduce the predicate `optimization_function`/1 to specify optimization functions, whereby the argument defines the name of the optimization function. Conceivable optimization functions are, for instance,

- to minimize the used computing nodes to potentially reduce the system's energy consumption,
- minimize the displacement of already running applications, and
- maximize the separation of applications that implement the same function.

These optimization functions are defined as follows:

```
optimization_function(min_comp_nodes).
optimization_function(min_displ_act).
optimization_function(max_sep).
```

The optimization function used for the application-placement problem is not static but has to be adapted according to the current context the vehicle is experiencing, i.e., optimization functions depend on the current context. This relation is modeled using the predicate `def_optimization_function`/3. The first argument specifies the unique name of the optimization function while the second one identifies either a vehicle-operation mode, a vehicle property, a user context, or an environmental-context value. The third argument specifies the optimization function, and the last defines a priority value. This value is used by `logAP`^{2}`S` in case multiple optimization functions are defined for a given context to rank them. Note that the specific priority value is of no meaning.

The linkage between context information and optimization functions is defined by the system architect. Assume, for instance, that in case the vehicle is on low power, the system architect defines that the system architecture shall be optimized so that less power is consumed. Furthermore, the system architect might also define that in case the vehicle is operated autonomously, the goal is to minimize the displacement of applications. In addition to this optimization function, the system architect might specify that the separation of applications that implement the same function shall be maximized if the vehicle is operated autonomously. To define which optimization function, i.e., minimize the displacement or maximize the separate, is of more importance, priorities can be used.

The illustrated relations between optimization functions and context information are expressed as follows:

```
def_optimization_function(low_power, min_comp_nodes, 60).
def_optimization_function(parked, min_comp_nodes, 60).
def_optimization_function(autonomous, min_displ, 80).
def_optimization_function(autonomous, max_sep, 40).
```

The Database DB_{op}

Operational-context information is modeled by the vehicle-operation mode and the operation property entities. Each vehicle-operation mode and operation property requires that certain functions are executed by the vehicle. Consequently, these entities are linked to functions.

The vehicle-operation modes describe, as stated in Section 3.1.2, the possible operational states of a vehicle, e.g., autonomous operation, manual operation, and parked.

Vehicle-operation modes are introduced by the predicate `vehicle_operation_mode`/1, whereby its single argument defines the name of the mode. For instance, the vehicle operation modes for autonomous driving, and parking are defined as follows:

```
vehicle_operation_mode(autonomous).
vehicle_operation_mode(parked).
```

The vehicle-operation mode can be further refined by operation properties. These properties are defined via the predicate `operation_property`/1, whereby its argument defines the name of the operation property. A refinement of the operational context is, for instance, information on whether the vehicle is operated commercially, in a test environment, or in a low power mode. These operation properties are defined as follows:

```
operation_property(commercial).
operation_property(test).
operation_property(low_power).
```

To express which functions the individual vehicle-operation modes and operation properties demand, the predicate `req_function`/3 is introduced. The first argument defines the vehicle-operation mode or the operation property. The second argument of the predicate `req_function` specifies the function which has to be executed under the given vehicle-operation mode or the operation property. The last argument defines the priority of the function in the specified vehicle-operation mode or the operation property, whereby we define three priority classes: *HIGH*, *MEDIUM*, and *LOW*. In C-SAR, the priority class *HIGH* is denoted by 2, *MEDIUM* is indicated by 1, and 0 corresponds to *LOW*. The priority classes specify which applications are prioritized in the event of a resource shortage in the vehicle. For instance, in case a hardware fault causes computing nodes to fail, which in return leads to a reduced computing capacity of the vehicle, the resource demands of applications of the priority *HIGH* are preferred.

The following shows a conceivable mapping between the functions defined above and the introduced vehicle-operation modes and operation properties:

```
req_function(autonomous, localization, 2).
req_function(autonomous, sensor_fusion, 2).
req_function(autonomous, ads_mode_manager, 2).
req_function(autonomous, interpretation_prediction, 2).
req_function(autonomous, drive_planning, 2).
req_function(autonomous, motion_control, 2).
req_function(autonomous, body_control, 2).
req_function(autonomous, shared_event_recording, 1).
req_function(autonomous, ride_visualization, 0).

req_function(parked, update_management, 2).
req_function(parked, traffic_optimization, 0).

req_function(commercial, ride_management, 1).
req_function(test, logging, 0).
```

In the autonomous vehicle-operation mode, all functions which perform the autonomous-driving task are required. These functions are of priority *HIGH* as they are safety-critical. Besides these functions also, the shared event-recording, as well as the ride-visualization function, are required in the autonomous mode. The former function is of priority *MEDIUM* as event-recordings are useful for accident investigation. However, this function is unlike the functions which perform the autonomous driving task, not safety-critical. Ride visualization is of priority *LOW* as this function only visualizes the perception and the planned actions of the vehicle for the passengers and is therefore not safety-critical.

The functions required for operating the vehicle do not have to be executed if the vehicle is in a parking position, as the vehicle is not moving. Instead, the vehicle resources are used to update the system and contribute to optimizing the traffic flow. The update-management function is of priority *HIGH* as updates might fix safety-critical bugs. On the other hand, the traffic-optimization function is of priority *LOW* as other vehicles can take over the optimization task.

If the vehicle is operated commercially, the ride-management function is required. This function is of priority *MEDIUM* as it is not safety-critical but essential for the economic efficiency of the vehicle. In case the vehicle is operated in a test environment, an additional logging function, which is of priority *LOW*, is required.

The Database DB_{usr}

The user context entity holds information about the user and the user's desires for the ride. Specific user contexts require the execution of that certain functions. Therefore, these user contexts are linked to functions.

User context information is defined via the predicate `user_context/1`, whereby the argument defines the name of the user context. For instance, in the case of an autonomous taxi, the taxi operator might offer its customer to choose between a premium and a low-cost ride option. These two options are defined as follows:

```
user_context(premium_ride).
user_context(low_cost_ride).
```

To define which functions the individual user contexts require, the above introduced predicate `req_function`/3 is used. The first and the second argument specifies the user context and the function, while the third argument defines the priority of the function.

The user contexts `premium_ride` and `low_cost_ride` can, for instance, demand the execution of the entertainment function and the traffic-optimization function:

```
req_function(premium_ride, entertainment, 0).
req_function(low_cost_ride, traffic_optimization, 0).
```

This example defines that if the customer ordered a premium ride, the entertainment function is provided on this ride. On the other hand, if the user orders a low-cost ride, the vehicle executes the traffic-optimization function instead of the entertainment function. Both these functions are of priority LOW.

The Database DB_{env}

The environmental-context information is expressed by the environmental-context category, environmental-context value, and the environmental-context set entity. Environmental-context categories structure environmental-context values and environmental-context sets. Environmental-context values and the environmental-context sets concretely define the environment, including, for instance, the weather condition, the scene in which the vehicle operates, and the lighting conditions. As applications can be designed for certain environmental contexts, they are linked to the environmental-context values and the environmental-context sets.

The environmental-context categories group the environmental-context value into several classes, e.g., weather, scene, and illumination. In C-SAR, the environmental-context type is defined by the predicate `environmental_context_category`/1, whereby the argument defines the name of the category:

```
environmental_context_category(weather).
environmental_context_category(scene).
environmental_context_category(illumination).
```

For each environmental-context category, several context values can be defined, which are specified via the predicate `environmental_context_value`/2. The first argument of this predicate defines the value and the second argument specifies the environmental-context category. For the categories weather, scene, and illumination, e.g., the following values can be defined:

```
environmental_context_value(clear, weather).
environmental_context_value(rainy, weather).
environmental_context_value(city, scene).
environmental_context_value(highway, scene).
environmental_context_value(daylight, illumination).
environmental_context_value(dark, illumination).
```

The relation between environmental-context values and application is defined via the predicate `env_cx_implementation/3`. The first argument defines the application, the second argument the environmental-context value, and the last argument specifies the so-called *environmental-context rating*. The environmental-context rating defines the suitability of an application in a given environment. CRC uses this rating to determine the best-suited applications for the current environmental context. The range of the environmental-context rating is between 0 and 100, whereby a higher rating indicates that an application is better suited in the given environmental context than an application with a low rating.

For instance, assuming that the before defined sensor fusion application `fus1` performs good under clear weather conditions, however bad if it is rainy. On the other hand, sensor fusion application `fus2` performs well in rainy weather. These properties are expressed as follows:

```
env_cx_implementation(fus1, clear, 70).
env_cx_implementation(fus1, rainy, 42).
env_cx_implementation(fus2, rainy, 84).
```

Data from test fleets and simulator runs can be analyzed to determine whether an application performs well or poorly in a given environmental context. Furthermore, datasets like the BDD100K [225] dataset, which contains 100,000 videos comprising different scenes, weather, and lighting conditions, can be used to evaluate the performance of an application under certain conditions. For instance, to determine the performance of a the camera-based detection of a sensor fusion application in the rain, all videos of the BDD100K dataset captured in rainy weather can be used.

The environmental-context rating of an application can comprise several different metrics. However, applications that implement the same function should be rated using the same metrics for reasons of comparability.

The performance of an application can also be rated for a more specific environmental context. A sensor fusion application can be rated, e.g., for urban driving in clear weather or for nightly journeys on highways while it is raining.

Therefore, the definition of environmental-context set is supported. Several environmental-context values from distinct categories can be grouped into an environmental-context set. These sets are defined by the predicate `environmental_context_set/1`, whereby the argument defines the name of the set. The predicate `env_cx_set_member/2` allows adding environmental-context values to an environmental-context set. The first argument of the predicate specifies the environmental-context value, and the second one defines the environmental-context set to which the value is added. For instance, the environmental-contexts described before can be introduced as follows:

```
environmental_context_set(rainy_highway_dark).

env_cx_set_member(rainy, rainy_highway_dark).
env_cx_set_member(highway, rainy_highway_dark).
env_cx_set_member(dark, rainy_highway_dark).
```

The relation between environmental-context sets and applications is defined like the association between environmental-context values and applications via the predicate `env_cx_implementation`. For instance, using the before defined term `rainy_high-way_dark` we can specify that `fus2` performs badly under those environmental contexts. In C-SAR, this is expressed as follows:

```
env_cx_implementation(fus2, rainy_highway_dark, 24).
```

Recall that before, we defined that `fus2` performs well when it is raining. This generic rating is refined by the before-specified rule. It states that although `fus2` performs in general good in rainy weather, the sensor fusion does not perform well in rainy weather when the vehicle is operated in the dark on a highway. Such an elaboration can be, for instance, required due to incidences caused by incorrect sensor fusion that occurred in the fleet.

Note that it is not required that an application is rated for all defined environmental-context values and environmental-context sets.

3.3.2 Current-Context Provider

The current-context provider is responsible for acquiring the current context and providing the obtained information to CRC. Current context information can be acquired by using data that is provided by sensors or other software services. Note that in this dissertation, we do not cover context-acquisition techniques. Nevertheless, we provide in what follows an overview of potential context-acquisition techniques.

Environmental-context information such as the weather and road surface conditions can be, for instance, provided by infrastructure services [39]. However, also the vehicle itself can determine weather and road surface conditions, e.g., by analyzing lidar data [187].

Furthermore, the extracted weather and road surface information can be shared with other vehicles [54].

Operational-context information, e.g., the number of passengers the vehicle is transporting, can be, for example, determined by analyzing the video streams provided by cameras mounted in the vehicle cabin [129]. Moreover, driver interference detection mechanisms can be used to determine whether the vehicle is operated autonomously or manually [194].

User-context information, such as the level of desired entertainment, the user's age, and the user category, which allows identifying whether the user is, e.g., a customer, part of the maintenance personnel, or a developer, can be provided by the HMI or backend services.

As mentioned before, the current context is given by the answer-set program P_{CC}. The current context is composed of one vehicle-operation mode and a set of operation properties, user contexts, and environmental-context values, whereby this set can be empty. To express the current context, we introduce the predicate `current_context`/1, which holds one argument that can be the name of a vehicle-operation mode, an operation property, a user context, or an environmental-context value.

For instance, if, in the current context, a vehicle is commercially driving in autonomy mode and chauffeuring passengers who booked a premium ride in a city scene while it is raining, the following holds:

```
current_context(autonomous).
current_context(commercially).
current_context(premium_ride).
current_context(city).
current_context(rainy).
```

By definition, the current context comprises exactly one operation mode. This is ensured by the following constraint:

```
:- #count{M: vehicle_operation_mode(M),
             current_context(M)} != 1.
```

Furthermore, the following constraint is introduced to ensure that only one environmental-context value of a specific environmental-context category is part of the current context:

```
:- environmental_context_category(CAT),
   #count{V: environmental_context_value(V,CAT),
             current_context(V)} > 1.
```

3.3.3 The Context-Reasoning Component CRC

The component CRC is implemented by the answer-set program

$$P_{\mathtt{CRC}} = P_{\mathtt{CRC\_1}} \cup P_{\mathtt{CRC\_2}} \cup P_{\mathtt{CRC\_3}} \cup P_{\mathtt{CRC\_4}},$$

where

- $P_{\mathtt{CRC\_1}}$ is responsible for choosing the optimization function that is requested by the current context,
- $P_{\mathtt{CRC\_2}}$ is responsible for choosing the tasks that are requested by the current context,
- $P_{\mathtt{CRC\_3}}$ takes care of selecting for each selected task one active application, and
- $P_{\mathtt{CRC\_4}}$ identifies the feasible diverse applications for each task.

In what follows, we illustrate the implementation of these programs. The complete program $P_{\mathtt{CRC}}$ is presented in Appendix A.1.

$P_{\mathtt{CRC\_1}}$: Optimization Function Selection

To indicate which optimization functions are desired in the current context, the predicate `selected_optimization_function/2` is introduced. The first argument of this predicate identifies the optimization function, and the second argument defines the priority of that optimization function.

To determine the optimization functions that are required in the current context, the following rule is introduced:

```
selected_optimization_function(OPT_FUNC, PRIO) :-
  optimization_function(OPT_FUNC), current_context(CUR_CX),
  def_optimization_function(CUR_CX, OPT_FUNC, PRIO).
```

$P_{\mathtt{CRC\_2}}$: Functions Selection

Based on the currently defined vehicle-operation mode, operation properties, and user context, a set of functions is selected. To indicate that a function is selected for the current context, the predicate `selected_function/2` is introduced. The first argument of this predicate holds the function's name, and the second argument defines its priority. If a function is required by the current context multiple times, the maximum priority is used.

The rule for selecting functions for the current context is defined as follows:

```
selected_function(FUNC, PRIO) :-
    function(FUNC), current_context(CUR_CX),
    req_function(CUR_CX,FUNC,PRIO),
    PRIO = #max{P: req_function(C, FUNC, P),
                   current_context(C)}.
```

This rule employs the aggregate function `#max` to determine the variable `PRIO` which specifies the maximum priority of a given function in the current context.

$P_{\text{CRC\_3}}$: Active Application Selection

For each selected function, one so-called *active application* has to be selected. We define an active application as an application that is executing in *active operation mode*. The active operation mode, in turn, indicates that an application is an instance of the *primary software architecture* of the vehicle. The latter architecture comprises those applications that imminently generate customer value. For instance, an active sensor-fusion application effectively contributes to the driving task and is, therefore, part of the primary software architecture. The output of the active sensor-fusion application is further processed by the other applications that operate the vehicle.

Besides the active operation mode, we also define the *active-hot operation mode*. Applications that are executed in this operation mode are referred to as *active-hot applications*. These applications do not imminently generate customer value but are redundancies of the active applications. Active-hot applications are part of the so-called *redundant software architecture*.

Program $P_{\text{CRC\_2}}$ distinguishes between the selection of applications for which environmental-context ratings exist or not exist. Applications of the former category are referred to as *rated applications*, while *non-rated applications* denote applications that do not depend on the environmental context. Generally, the selection of rated applications over non-rated ones is preferred. In particular, $P_{\text{CRC\_2}}$ selects applications with the most specific and highest rating to become active applications. Therefore, the so-called *level of specialization* has to be determined, which corresponds to the number of environmental-context values that are members of this set. Moreover, the level of specialization of environmental-context values is 1.

The level of specialization is defined by the predicate `specialization_level`/3, whereby the first argument specifies the application, the second the environmental-context set or environmental-context value, and the last argument specifies the level of specialization.

For instance, the level of specialization of the previously defined sets `clear_city` and `rainy_highway_dark` is 2 and 3, respectively. However, the level of specialization is only determined if all environmental-context values of an environmental-context set are part of the current context.

The rule for determining the level of specialization of environmental-context sets is defined as follows:

```
specialization_level(APP, SET, SPEC_L) :-
    env_cx_implementation(APP, SET, _),
    SPEC_L = #count{V: env_cx_set_member(V, SET)},
    MATCHES = #count{C: environmental_context_set(SET),
                        env_cx_set_member(C, SET),
                        current_context(C)},
    SPEC_L = MATCHES.
```

Note that the variable SPEC_L expresses the number of environmental-context values that are in a specific environmental-context set. On the other hand, the variable MATCHES counts the number of environmental-context values that are part of the environmental-context set and part of the current context. The level of specialization is only determined if all environmental-context values of an environmental-context set are part of the current context, i.e., if SPEC_L = MATCHES holds.

The rule defined above does not determine the level of specialization if the environmental-context rating of an application is specified for an individual environmental-context value, e.g., clear, city, or dark. In this case, as mentioned before, the level of specialization is 1. To define the specialization level, if the rating of an application is defined for an environmental-context value, we add the following rule:

```
specialization_level(APP, CUR_CX, 1) :-
    current_context(CUR_CX),
    env_cx_implementation(APP, CUR_CX, _),
    environmental_context_value(CUR_CX, _).
```

Next, the most specific environmental-context value and environmental-context set, which are fully covered by the current context, are determined.

The predicate max_special_env_cx/2 identifies for each application for which specialization levels are defined the most specific environmental-context value or environmental-context set, whereby the first argument defines the name of the application and the second argument the most specific environmental-context value or set. We introduce the following rule to determine this predicate:

```
max_special_env_cx(APP, ENV_CX) :-
    specialization_level(APP, ENV_CX, SPEC_L),
    SPEC_L = #max{SV:specialization_level(APP, _, SV)}.
```

This rule employs the aggregate *#max* to determine the maximum specialization value for an application APP. The specialization level, SPEC_L, of an environmental-context set or environmental-context value, ENV_CX, has to be equal to this maximum value in order that max_special_env_cx(APP,ENV_CX) holds.

For an application, multiple environmental-context values and environmental-context sets can have the same level of specialization. For instance, consider that for the sensor-fusion application fus1, in addition to the set clear_city, a rating is defined for the context-environment set clear_dark, which comprises the context-environment values clear and dark. If the current context comprises the context-environment values clear, city, and dark, both max_special_env_cx(fus1, clear_city) and max_special_env_cx(fus1, clear_dark) holds. Furthermore, we assume that the application fus1 is rated at 42 for the set clear_city and at 23 for the set clear_dark. In such a case, CRC acts conservatively and chooses the lower rating, i.e., 23.

We define the predicate cx_application_rating/2 to specify the rating of an application in the current context, whereby the first argument defines the application name and the second argument the determined rating. Based on these ratings CRC decides which application shall be selected for a function. For instance, in the example discussed before, cx_application_rating(det1,23) holds. The following rule illustrates how the predicate cx_application_rating is determined:

```
cx_application_rating(APP, RATING) :-
    application(APP, FUNC),
    max_special_env_cx(APP, ENV_CX),
    env_cx_implementation(APP, ENV_CX, RATING),
    RATING = #min{R: max_special_env_cx(APP, E_C),
                     env_cx_implementation(APP, E_C, R)}.
```

As multiple rated applications can implement a function, those with the highest level of specialization have to be identified. To indicate which applications are a feasible choice for a function, we define the predicate feasible_rated_application/2, whereby the first argument defines the application and the second the function. This predicate is determined by the following rule:

```
feasible_rated_application(APP, FUNC) :-
    application(APP, FUNC),
    max_special_env_cx(APP, ENV_CX),
    specialization_level(APP, ENV_CX, SPEC_L),
    SPEC_L = #max{S_L: application(A, FUNC),
                       max_special_env_cx(A,E_C),
                       specialization_level(A, E_C, S_L)}.
```

Furthermore, the predicate feasible_non_rated_application/1 identifies all non-rated applications implementing a selected function for which no rated application exists in the current context. The argument of this predicate identifies the non-rated application and is determined by the following rule:

```
feasible_non_rated_application(APP) :-
    application(APP, FUNC), selected_function(FUNC, _),
    not env_cx_implementation(APP, _, _),
    not feasible_rated_application(_, FUNC).
```

As multiple rated applications that implement the same function can have the same level of specialization, the application with the highest rating is selected using the predicate `max_rated_application`/1, where its argument holds the name of the application. The predicate `max_rated_application` is determined by the following rule:

```
max_rated_application(APP) :-
    selected_function(FUNC, _),
    application(APP,FUNC),
    feasible_rated_application(APP, FUNC),
    cx_application_rating(APP, RATING),
    RATING = #max{R: feasible_rated_application(A, FUNC),
                     cx_application_rating(A, R)}.
```

Finally, for each selected function, the application with the highest rating, which is the first in alphabetically ascending order regarding its name, is selected as active application.

To indicate that an application is selected as active application, we introduce the predicate `active_application`/2. The first argument of this predicate specifies the name of the application, and the second term holds the so-called *redundancy-instance number*. We associate the application name and the redundancy-instance number to identify active and active-hot applications uniquely. Note that the redundancy-instance number of the active applications is 0. The active applications are determined using the following rule:

```
active_application(APP,0) :-
    max_rated_application(APP),
    application(APP,FUNC),
    APP = #min{A: max_rated_application(A),
                  application(A, FUNC)}.
```

In case multiple non-rated applications exist for a function for which no rated application has been selected, `CRC` selects the first application in the alphabetically ascending order using the following rule:

```
active_application(APP,0) :-
    selected_function(FUNC, _),
        application(APP,FUNC),
        feasible_non_rated_application(APP),
        APP = #min{A: feasible_non_rated_application(A),
                      application(A, FUNC)}.
```

To ensure that for each function that is required in the current context, exactly one application, which implements this function, is selected, we introduce a constraint. Note that in case the constraint does not hold, the program is unsatisfying, i.e., no answer sets can be found. The constraint is defined as follows:

```
:- selected_function(FUNC, _),
   #count{A: active_application(A, _),
             application(A, FUNC)} != 1.
```

$P_{\text{CRC\_4}}$: Diverse Application Determination

To increase the safety and reliability of the autonomous vehicle, some applications have to be executed redundantly. Recall that the required level of redundancy for an individual application is defined via the attribute `redundancy`. For instance, `redundancy(fus1, 2)` specifies that in case `fu1` is selected as active sensor-fusion application, two redundant sensor-fusion applications are needed. Whether these redundant applications have to be diverse from `fus1` is defined by the predicate `diversity`. Assuming `diversity(fus1, 1)` holds, one of the redundant applications has to be diverse, whereas the other redundant application is like the active application an instance of `fus1`.

The program $P_{\text{CRC\_4}}$ determines for each selected function all feasible diverse applications. A rated application is classified as a feasible diverse application for a selected function if the application has not been selected as active application, and the predicate `max_special_env_cx` is defined for this application. Recall that the predicate `max_special_env_cx` indicates that an application has been rated for an environmental-context set that includes all environmental-context values of the current context.

We introduce the predicate `feasible_divers_application`/3 to identify feasible diverse applications. The first argument of this predicate identifies the application, whilst the second one specifies the maximum specialization value. The third argument specifies the rating of the application in the current context. The latter two arguments are subsequently used by PostProc to decide which feasible diverse application shall be selected. The following rule determines the feasible diverse applications:

```
feasible_diverse_application(APP, SPEC_L, RATING) :-
    application(APP, FUNC),
    selected_function(FUNC, _),
    max_special_env_cx(APP, ENV_CX),
        cx_application_rating(APP, RATING),
        specialization_level(APP, ENV_CX, SPEC_L),
        SPEC_L = #max{S_L: specialization_level(APP, _, S_L)},
        not active_application(APP, _).
```

Note that the previously introduced rule does not consider non-rated applications. Therefore, an additional rule which identifies non-rated feasible diverse applications has to be defined.

A non-rated application is classified as a feasible diverse application for a selected function if the application is not selected. As before, the predicate `feasible_diverse_application` is used to indicate that a non-rated application is a feasible diverse application. As the procedure for selecting diverse applications prefers diverse rated applications over non-rated applications the specialization level and the rating is set to 0.

The following rule determines all non-rated feasible diverse applications:

```
    feasible_diverse_application(APP, 0, 0):-
        application(APP, FUNC),
        selected_function(FUNC, _),
        not env_cx_implementation(APP, _, _),
        not active_application(APP, _).
```

3.3.4 The Post-Processing Component

Based on the answer set A_{SAR} computed by CRC, PostProc determines for each selected task the required set of active-hot applications. Recall that active-hot applications are those executed in the active-hot operation mode and are part of the redundant software architecture. Active-hot applications are either of the same instance as the active application or are diverse. Therefore, in a first step, PostProc determines for each function whether the level of redundancy of a function is greater than the level of diversity. If this is the case, PostProc adds active-hot applications that are of the same instance as the active application using the predicate `active_hot_application`/2. The first term of this predicate specifies the name of the application, and the second term holds the redundancy-instance number.

In order to select for each function the required diverse active-hot applications, the feasible diverse applications are sorted by their specialization level and their rating. Recall that the predicate `feasible_diverse_application` holds for feasible diverse applications.

Finally, the applications with the highest specialization level and rating are selected as diverse active-hot applications and added to the answer set. The resulting extended answer set, A'_{SAR}, is forwarded to D-DEG.

Implementing such multilevel sorting, i.e., ranking feasible diverse applications according to their specialization level and their rating, in ASP is feasible yet cumbersome. Consequently, we decided to implement the selection of active-hot applications in Python.

The following illustrates the implementation of the function selection of active-hot applications:

```
def get_active_hot_applications(active_apps, functions, answer_set):        1
   active_hot_apps = []                                                        2
   active_hot_app_literal = 'active_hot_application({}, {})'                   3
                                                                               4
   for active_app in active_apps:                                              5
                                                                               6
      diverse_apps = []                                                        7
      redundancy_instance_number = 1                                           8
                                                                               9
      non_diverse_active_hot_apps = active_app.redundancy - \                 10
                                    active_app.diversity                      11
                                                                              12
      for _ in range(non_diverse_active_hot_apps):                            13
         active_hot_apps.append(                                              14
            active_hot_app_literal.format(active_app.name,                    15
                                          redundancy_instance_number))        16
         redundancy_instance_number += 1                                      17
                                                                              18
      apps = functions[active_app.function].all_apps                          19
                                                                              20
      for predicate in answer_set:                                            21
         predicate_name = predicate[0]                                        22
         if predicate_name == 'feasible_diverse_application':                 23
            diverse_app = DiverseApplication()                                24
            diverse_app.name = predicate[1][0]                                25
            diverse_app.specialization_level = predicate[1][1]                26
            diverse_app.rating = predicate[1][2]                              27
                                                                              28
            if diverse_app.name in apps:                                      29
               diverse_apps.append(diverse_app)                               30
                                                                              31
      diverse_apps_sorted = sorted(diverse_apps, reverse=True,                32
                                   key=lambda d: (d.specialization_level,     33
                                                  d.rating))                  34
      for i in range(active_app.diversity):                                   35
         active_hot_apps.append(                                              36
            active_hot_app_literal.format(diverse_apps_sorted[i].name,        37
                                          redundancy_instance_number))        38
            redundancy_instance_number += 1                                   39
                                                                              40
   return active_hot_apps                                                     41
```

The first part of the function, i.e., lines 10 to 17, determines the number of non-diverse active-hot applications. Those applications are of the same application instance as the primary application. Note that the redundancy-instance number allows distinguishing the individual non-diverse active-hot applications.

In the second part of the function, i.e., lines 19 to 39, the diverse active-hot applications are determined. In the first step, the function extracts all feasible diverse applications of the primary application, which CRC has determined. Next, the diverse applications are sorted by their specialization level and their rating. Finally, the applications with the highest specialization level and rating are selected as diverse active-hot applications.

3.4 Dynamic Cooperation-Based Degradation Approach

As the available resources, e.g., computing power, bandwidth, and power consumption, in autonomous vehicles are limited, due to, e.g., cost, space, and technical constraints, an economical use of resources is required. Furthermore, reducing resource consumption can also benefit, for instance, the range of electrically operated autonomous vehicles or the lifetime of hardware components.

To equip Aptus with resource-reduction capabilities, we outline in what follows a dynamic cooperation-based degradation approach for applications, which we refer to as D-deg.

The basis of D-deg is that autonomous vehicles that use the same sensor set, execute the same applications, and are in close proximity perceive and compute similar data. The idea is to share application outputs between vehicles using VANET (Vehicular Ad-Hoc Network) technologies, including 5G-NR, Wi-Fi, and UWB [5]. The transferred information is used to achieve resource preservation, whereby D-deg aims to reduce resource consumption by degrading applications.

In addition to D-deg, other approaches aiming for preserving resources which are based on using information received by other vehicles have been introduced in the past. For instance, Hexmoor and Yelasani [93] study resource-efficient platooning approaches. They point out that all vehicles, except for the leading vehicle, can shut down all sensors to save energy.

Vahidi and Sciarretta [213] discuss energy-saving potentials when operating a network of connected vehicles. Their approach aims to decrease fuel consumption by sharing data relevant to the driving condition. The authors predict further energy-saving capabilities in operating vehicles in platoons.

Utilizing the information received by other vehicles is not limited to aim for a reduction of energy consumption. For instance, Kausar et al. [126] discuss the possibility of sharing the sensor data already processed by a particular vehicle in a way that another vehicle can use this data to process and detect collision courses.

Saxena et al. [192] introduce an approach where vehicles share the obtained surrounding information with nearby vehicles. The shared information is utilized to validate the perceived environment data of the vehicle. Therefore, each autonomous vehicle represents the perceived environment in a size-efficient data format, which is shared with other vehicles. These vehicles can fuse received information into their environment representation.

Jia et al. [112] present an overview of platoon-based vehicular cyber-physical systems. Vehicles with common interests can cooperatively form a platoon. To build and operate

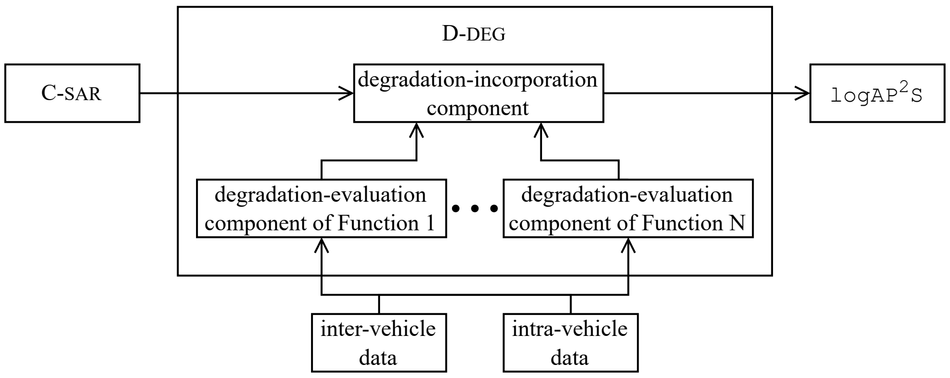

D-DEG
C-SAR
degradation-incorporation component
logAP²S
degradation-evaluation component of Function 1
degradation-evaluation component of Function N
inter-vehicle data
intra-vehicle data

Figure 3.5: Overview of D-DEG and its components.

platoons, vehicles exchange information using VANET technologies. According to Jia et al., platoons can improve road capacity, safety, and energy efficiency.

Aoki et al. [11] present a deep reinforcement learning approach for cooperative perception to increase detection accuracy, which aims to reduce the amount of data transferred between vehicles. According to Aoki et al., a cooperative perception can increase road safety as, for instance, vehicles can eliminate blind spots by using the perception data received from surrounding vehicles. A similar approach of a cooperative perception approach using VANET technologies is discussed by Günther [85].

We continue now with an overview of the intended functionality of D-DEG and discuss examples that illustrate the different degradation approaches applied by D-DEG.

3.4.1 Approach Overview

D-DEG, as illustrated in Figure 3.5, consists of the so-called *degradation-incorporation component* and a *degradation-evaluator component* for all functions which include degradable applications. Note that the application developer is responsible for determining whether an application is eligible for being degraded using D-DEG.

In the first step, D-DEG instructs, based on the answer set A'_{SAR} received from C-SAR, the degradation-evaluation component of those functions which are intended to be executed to evaluate whether the applications of that function can be degraded. In particular, the degradation-evaluation component can choose between two so-called *degradation modes*, which we refer to as DEG_1 and DEG_2, for degrading applications.

The degradation mode DEG_1 aims to lower the resource use by reducing the number of redundant applications. For instance, a degradation-evaluation component may conclude that the information received by another vehicle can compensate for the output information of an active-hot application, which is used to validate the output of active application. Consequently, the degradation-evaluation component selects the respective active-hot application for being removed for the configuration.

On the other hand, the degradation mode DEG_2 is designed to decrease resource requirements by degrading active applications. Therefore, we introduce an additional operation mode called *active-low*. An active-low application has the same characteristics as an application executed in active operation mode. However, active-low applications process the input data with a lower allocation of resources, e.g., a reduced memory or CPU utilization.

Once the degradation-evaluation components finish their analysis, they report their results to the degradation-incorporation component. This component then incorporates the determined degradation measures into the answer set A'_{SAR}. We refer to this adapted answer set $A_{D\text{-}DEG}$.

If a degradation-evaluation component reports that DEG_1 can be applied, then the degradation-incorporation component removes the specified active-hot applications in the answer set. On the other hand, in case DEG_2 can be applied, the degradation-incorporation component removes the fact from the answer set which specifies that the respective application is executed in the active operation mode. Instead of this fact, the degradation-incorporation component adds a fact that specifies that the application shall run in the active-low operation mode.

To indicate that an application is executed in active-low operation mode, we introduce the predicate `active_low_application`/2. The first argument of this predicate specifies the name of the application, and the second term holds the so-called redundancy-instance number.

In the final step, the degradation-incorporation component provides $A_{D\text{-}DEG}$ to the application-placement problem solver `logAP`^{2}`S`, which is responsible for finding a configuration that fulfills the received software-architecture requirements.

In what follows, we discuss examples that illustrate the two degradation modes.

3.4.2 Application Degradation Use-Cases

To illustrate degradation mode DEG_1, we assume a scenario, as shown in Figure 3.6, in which vehicle V_1 is following vehicle V_2 on a highway. Furthermore, we assume that vehicle V_1 redundantly executes a drive-planning application.

As V_1 follows V_2, the drive planning of both vehicles is very similar. Consequently, V_2 can transmit the drive planner output information to V_1 that, in return, uses the received data to validate the output of the active drive planner. As long as the data exchange between V_1 and V_2 is uninterrupted and the context remains unchanged, the redundancy of the drive-planning application can be reduced.

On the other hand, the scenario illustrated in Figure 3.7 which shows three vehicles driving beside one another demonstrates the use of the degradation mode DEG_2.

We assume that all three vehicles redundantly execute an interpretation and prediction application. As vehicle V_2 is surrounded by vehicle V_1 and V_3, V_2 will only perceive

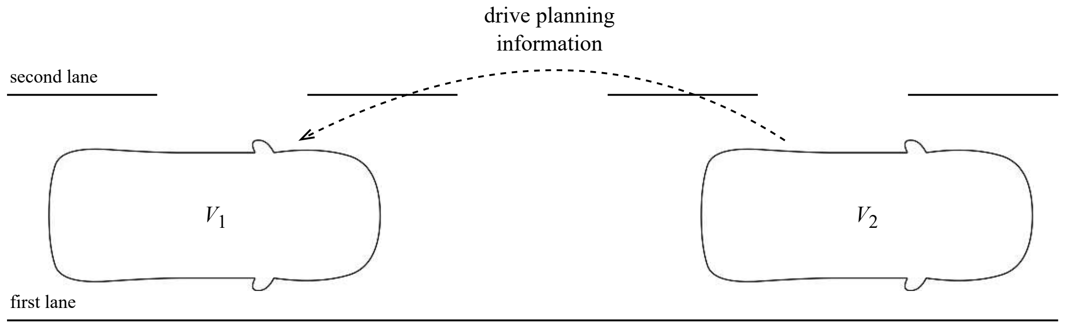


Figure 3.6: Scenario of two cars driving behind one another on a highway. D-DEG applies the degradation mode DEG_1 in this scenario.

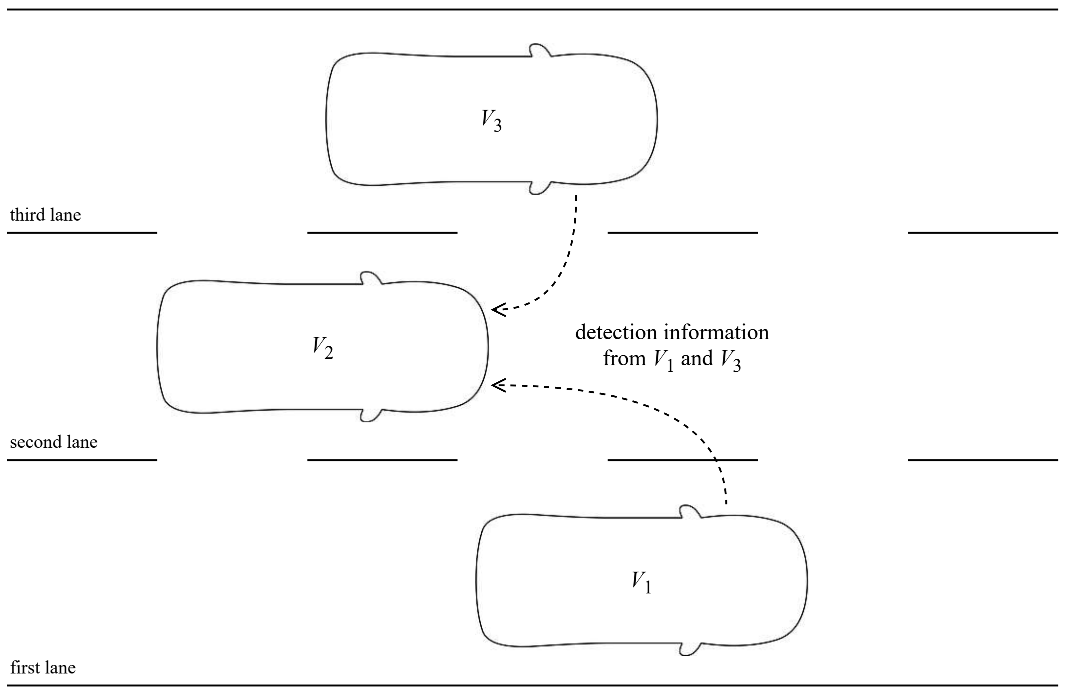


Figure 3.7: Scenario of three vehicles driving next to each other. D-DEG applies the degradation mode DEG_2 in this scenario.

objects that are also perceived by V_1 and V_3. Therefore, V_2 can use the interpretation and prediction information received by V_1 and V_3 to incorporate that information in the environment model. Consequently, the active interpretation and prediction application of V_2 can be downgraded to active-low. For instance, this application can lower the frequency at which the future state of the detected objects are forecasted. The output of the active-low application can be used to validate the information received by V_1 and V_3.

3.5 Summary of the Context Layer

In this chapter, we introduced the context layer of Aptus, which implements the context-awareness property. This layer consists of two components, namely, C-sar and D-deg.

C-sar, which employs answer-set programming, is responsible for determining, based on the current context, the software-architecture requirements that the autonomous vehicle shall apply. The implementation illustrated in the chapter shows that answer-set programming is a suitable knowledge-representation language for the task of C-sar, as the programs are compact and comprehensible.

The requirements determined by C-sar are, in the next step, passed to D-deg for further optimization. The aim of D-deg is to refine the received set of software-architecture requirements by degrading applications where appropriate. To assess whether applications are eligible for degradation, D-deg analyzes data received from nearby vehicles. In the final step, D-deg provides the optimized set of software-architecture requirements to the reconfiguration layer of Aptus, which is described in the following chapter.

CHAPTER 4

The Reconfiguration Layer

In this chapter, we introduce the reconfiguration layer of Aptus, which is responsible for implementing the self-configuration property, the self-optimization property, and anticipativeness. To implement those properties, the reconfiguration layer consists of the following four main components: C-po$_{\text{Aptus}}$, logAP^{2}S, linAP^{2}S, and CGM.

As the central task of the configuration layer is to determine configurations, we describe in the first part of this chapter the application-placement problem as studied in this dissertation, following our previous investigations [116, 123]. Next, the so-called *configuration graph*, which represents the interconnection of the individual configurations, is presented. Moreover, we present the workflow of the reconfiguration layer and introduce the main components of the reconfiguration layer. Finally, we provide a summary of this chapter.

4.1 The Application-Placement Problem

The task of determining the assignment between applications and computing nodes is referred to as the *application-placement problem* [205]. The relevance of the application-placement problem is not limited to the area of autonomous vehicles. Indeed, the placement of applications on computing nodes is a well-studied topic in other areas. In particular, research in cloud and edge computing has addressed this problem in various publications, where the focus of these works is on optimizing different properties including, e.g., energy consumption [143], network traffic load [7], or resource utilization [27].

4.1.1 Definition of the Problem in the Context of Aptus

In our setting [116, 123], the input of the application-placement problem is a set A of applications and a set N of computing nodes, and the task is to find a function C, called *configuration*, that maps each application $a \in A$ to exactly one node $n \in N$ such that certain conditions, defined in terms of a set of parameters, are satisfied. Moreover, in order to discriminate among a potentially large number of solutions, we utilize an additional optimization function that specifies which valid node assignment is desired most. Conceivable optimization goals are, e.g., minimizing the number of applications that have to be displaced or maximizing the number of computing nodes that execute no applications.

For the application-placement problem studied in this dissertation, the application parameters provided by the context layer, i.e., the memory demand, the performance demand, the list of required supporting software, and the function that the application implements, are considered. As mentioned in Section 3.3.1, we assume that one application implements only one function (e.g., localization, sensor fusion, drive planning, etc.) and multiple applications can implement the same function. For each function, we consider the parameters defined by the context layer, i.e., the priority, the level of redundancy, the level of diversity, and the level of separation.

Furthermore, for each computing node, we specify the memory capacity, the performance capacity, and the installed software.

The conditions a valid solution of our application-placement problem needs to satisfy are the following:

(C_1) An application has to be executed by exactly one computing node.

(C_2) The sum of the memory demands of all applications running on a computing node cannot exceed the memory capacity of that node.

(C_3) The sum of the performance demands of all applications running on a computing node cannot exceed the performance capacity of that node.

(C_4) An application runs only on such a computing node that offers the software required by that application.

(C_5) The applications that implement the same function have to run on at least a certain number of distinct computing nodes, i.e., the level of separation has to be satisfied for each function.

From a computational point of view, solving the application-placement problem is a complex task. Indeed, it is easily seen that the version of the application-placement problem studied by Tang et al. [205] is a special case of our variant and as the former one is NP-hard, it follows that our formulation of the problem is NP-hard too. In fact, NP-hardness follows from the fact that the *class constrained multiple-knapsack problem* [198], which is NP-hard, can be encoded by a polynomial-time reduction to the application-placement problem.

4.1.2 Approaches to Solve the Problem

Several approaches can be used to effectively model and solve the application-placement problem. In what follows, we discuss some of these approaches.

Integer Linear Programming

Integer linear programming is a mathematical optimization approach where decision variables are constrained to discrete values [196]. The problem is modeled using linear constraints and linear objective functions.

To model the application-placement problem using integer linear programming, *decision variables* can be used to represent whether a specific application is assigned to a particular computing node. Furthermore, the problem's conditions can be encoded as linear constraints.

As stated before, the objective function is restricted to being linear. Consequently, alternative optimization approaches must be employed if nonlinear optimization goals are required.

Mixed-Integer Nonlinear Programming

An extension of integer linear programming is *mixed-integer nonlinear programming* [26]. Compared to integer linear programming, mixed-integer nonlinear programming allows the use of continuous variables as well as nonlinear constraints and nonlinear objective functions. Therefore, mixed-integer nonlinear programming is a suitable option for solving the application-placement problem if the desired objective function is nonlinear.

Logic Programming

The application-placement problem can also be modeled in terms of logic programming, whereby facts may be used to represent static information, such as applications, nodes, and their respective properties, and rules are used to define the conditions of the problem.

Prominent languages for logic programming include, besides answer-set programming [78], Prolog [49] and constraint logic programming [134]. Several well-known logic programming systems exist [141, 75, 221, 12] allowing the formulation of optimization goals, like using, e.g., minimization and maximization statements.

SAT Solving

Satisfiability solving (SAT) [34] is a method that aims to determine whether there exists a satisfying truth-value assignment to a given formula, which is typically expressed in *conjunctive normal form* (CNF). In the context of the application-placement problem, atomic formulas can represent whether a specific application is assigned to a particular computing node. Furthermore, the conditions of the application-placement problem can be encoded as a CNF formula. Note that this encoding can be cumbersome and difficult. However, SAT solvers, like FourierSAT [138] and MiniCARD [147], that support cardinality constraints, can simplify this process.

To incorporate an optimization function, e.g., MaxSAT [144] can be applied, which extends SAT by aiming to find a truth assignment that maximizes the number of satisfied clauses.

Satisfiability Modulo Theories

An extension of SAT is *satisfiability modulo theories* (SMT) [21]. Unlike SAT solvers, which are limited to determining the satisfiability of CNF formulas, SMT solvers are more versatile and can evaluate the satisfiability of formulas with respect to a given background theory.

Theories that can be chosen to model the application-placement problem are, e.g., the theory of real arithmetic, the theory of arrays, or the theory of bit vectors. In order to incorporate optimization functions, *optimization modulo theories* (OMT) solvers can be employed [36, 197, 145], which extend SMT solving with optimization capabilities.

Approaches based on Machine Learning

Approaches which use machine-learning techniques can be used to predict optimized application placements. These approaches are particularly effective when historical data or simulation results are available.

Especially reinforcement learning could be used for solving the application-placement problem [164]. The idea is that an agent learns an optimal policy for placing applications by receiving feedback.

Furthermore, also supervised learning can be employed to solve the application-placement problem [183]. However, to use supervised learning, a comprehensive set of application-placement problem inputs and their corresponding valid configurations is required to train a model.

Metaheuristic Algorithms

Metaheuristic algorithms are generic optimization procedures that can be applied to a variety of problems [1]. Rather than finding an optimal solution, these algorithms aim for near-optimal solutions.

Metaheuristic algorithms that can be used to solve the application-placement problem include, for instance, genetic algorithms [95], simulated annealing [130], and tabu search [81].

4.1.3 Chosen Approaches for Aptus

Aptus is designed to be solver-independent and, therefore, supports the integration of various different solving approaches for the application-placement problem. Determining the approaches that are best suited for the application-placement problem requires an exhaustive comparison. However, in this dissertation, we focus on introducing the general concept of Aptus rather than optimized implementations. Consequently, we adopted a pragmatic approach and selected two well-supported and complementary approaches for solving the application-placement problem: integer linear programming and answer-set programming.

Integer linear programming was chosen for its ability to quickly compute solutions. Therefore, this approach is excellently suited for determining configurations after hardware and software faults, as in such cases, fast reconfiguration times are required. However, integer linear programming is restricted to linear constraints and linear optimization functions. In contrast, answer-set programming allows intuitive modeling of the application-placement problem and supports the use of nonlinear optimization goals. Therefore, ASP is ideal for implementing a wide range of optimization functions.

4.2 The Configuration Graph

New configurations have to be computed in case certain events occur. For instance, the fault of an application or a computing node can cause that some software-architecture requirements are no longer satisfied. Consequently, a new configuration, which aims to fulfill the current software-architecture requirements, is needed. Note that several applications and several computing nodes can fail at the same time. Furthermore, a new configuration is required in case a failed computing node is reactivated, whereby we assume several computing nodes can be reactivated at the same time.

A computation of a new configuration might also be initiated in case the current software-architecture requirements change. Such an update of the prevailing software-architecture requirements can occur if the current context changes. Furthermore, as configurations can be optimized according to various characteristics, the computation of a new configuration might be motivated by the aim to optimize the current configuration.

We refer to the events that trigger the computation of a new configuration as *initiator events*. In total, we distinguish between the aforementioned five different types of initiator events:

- the fault of a specific application,
- the fault of a specific computing node,
- the reactivation of a specific computing node,
- refined software-architecture requirements, and
- the request to optimize the current configuration.

We define the set of all initiator events, which is denoted by $\mathcal{I}$, as follows:

$$\mathcal{I} = (2^{\mathcal{I}_f} \cup 2^{\mathcal{I}_r} \cup \{\iota_u\} \cup \{\iota_o\}) \setminus \emptyset,$$

whereby $\mathcal{I}_f$ is the set of all application and computing node faults, $\mathcal{I}_r$ denotes the set of all computing node reactivations, ι_u defines the initiator event that indicates updated software-architecture requirements, and ι_o specifies the optimization initiator event.

The initiator events can be used to link configurations in order to trace their origin. However, initiator events solely do not uniquely identify the transition from one configuration to its successive configuration. To uniquely identify the transitions between configurations, also the current software-architecture requirements determined by the context layer are required.

Let $\mathcal{P}$ be the set of all software-architecture requirements, then the tuple (ι, ρ), whereby $\iota \in \mathcal{I}$ and $\rho \in \mathcal{P}$, identifies the transitions between configurations. We refer to these tuples as *configuration transitions*. The set of all *configuration transitions*, which is denoted by T, is defined as follows:

$$T = \{(\iota, \rho) \mid \iota \in \mathcal{I}, \rho \in \mathcal{P}\}.$$

Assume that Γ is the set of all configurations and F is the function that computes a new configuration $C \in \Gamma$ based on a configuration $C' \in \Gamma$ and a configuration transition $\tau \in T$. For all configurations except for the initial configuration, C_{init}, which is the configuration the vehicle is delivered with, it holds that there exists a configuration, C', and a configuration transition, τ, such that $F(C', \tau) = C$ holds. This can be expressed by the following condition:

$$\forall C \in \Gamma \setminus \{C_{init}\},\ \exists C' \in \Gamma,\ \exists \tau \subseteq T : F(C', \tau) = C.$$

We call the resulting graph *configuration graph*. As illustrated in Figure 4.1, the configuration graph $CG = (V, E)$ is an edge-labeled directed graph, whereby V is the set of vertices, and E defines the directed edges between the vertices as well as the corresponding labels. The vertices of the configuration graph are a subset of all configurations, i.e., $V \subseteq \Gamma$. The edges of the configuration graph are a set of triples (C', C, τ), whereby $F(C', \tau) = C$ holds.

4.3 Workflow of the Reconfiguration Layer

The reconfiguration layer consists of multiple elements. These elements and their activities are illustrated in Figure 4.2.

The point of entrance of the reconfiguration layer is the so-called `AP`^{2}`Initiator`, whereby "`AP`^{2}" is an abbreviation for "application-placement problem". This component is responsible for setting up the application-placement problem if a request for computing a new configuration arises.

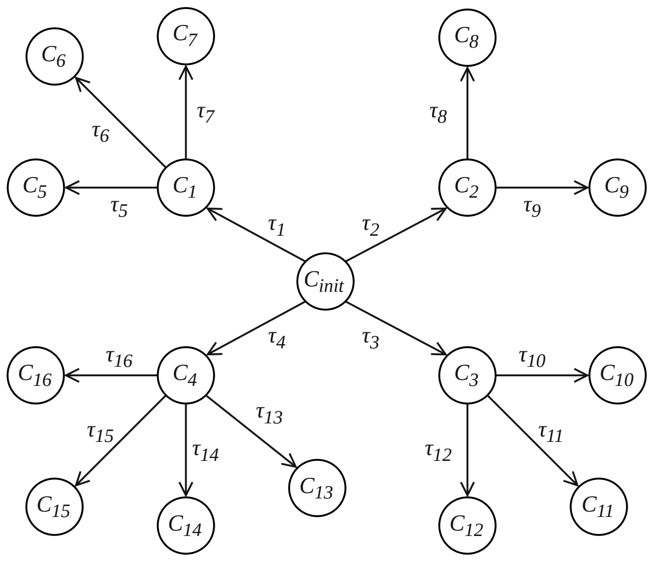


Figure 4.1: Extract of a configuration graph. The initial configuration is denoted by C_{init}. For $C_i \in \Gamma$ and $\tau_i \in T$, whereby $i \in \{1, \ldots, 4\}$, $F(C_{init}, \tau_i) = C_i$ holds.

In order to set up the application-placement problem and initiate its processing, the following input information is required for `AP`^{2}`Initiator`:

- the software-architecture requirements determined by the context layer,
- the initiator event, as defined in Section 4.2,
- the so-called *precompute flag*,
- the current operation modes of the applications, and
- the so-called *currently applied configuration*, which is the configuration that was determined by the reconfiguration layer in the previous iteration in which the precompute flag was not set.

A request for computing a new configuration is, for instance, triggered by a change in the software-architecture requirements, which results from a revised current context. In this case, the context layer generates a request for a new configuration after determining the software-architecture requirements.

Furthermore, new configurations are required after the fault of a software or hardware component. Such faults are initially handled by the component layer. In particular, Aptus employs the fault-tolerance approach Fdiro. However, the recovery and the optimization step of Fdiro require applying new configurations. Therefore, these two steps are executed by the reconfiguration layer.

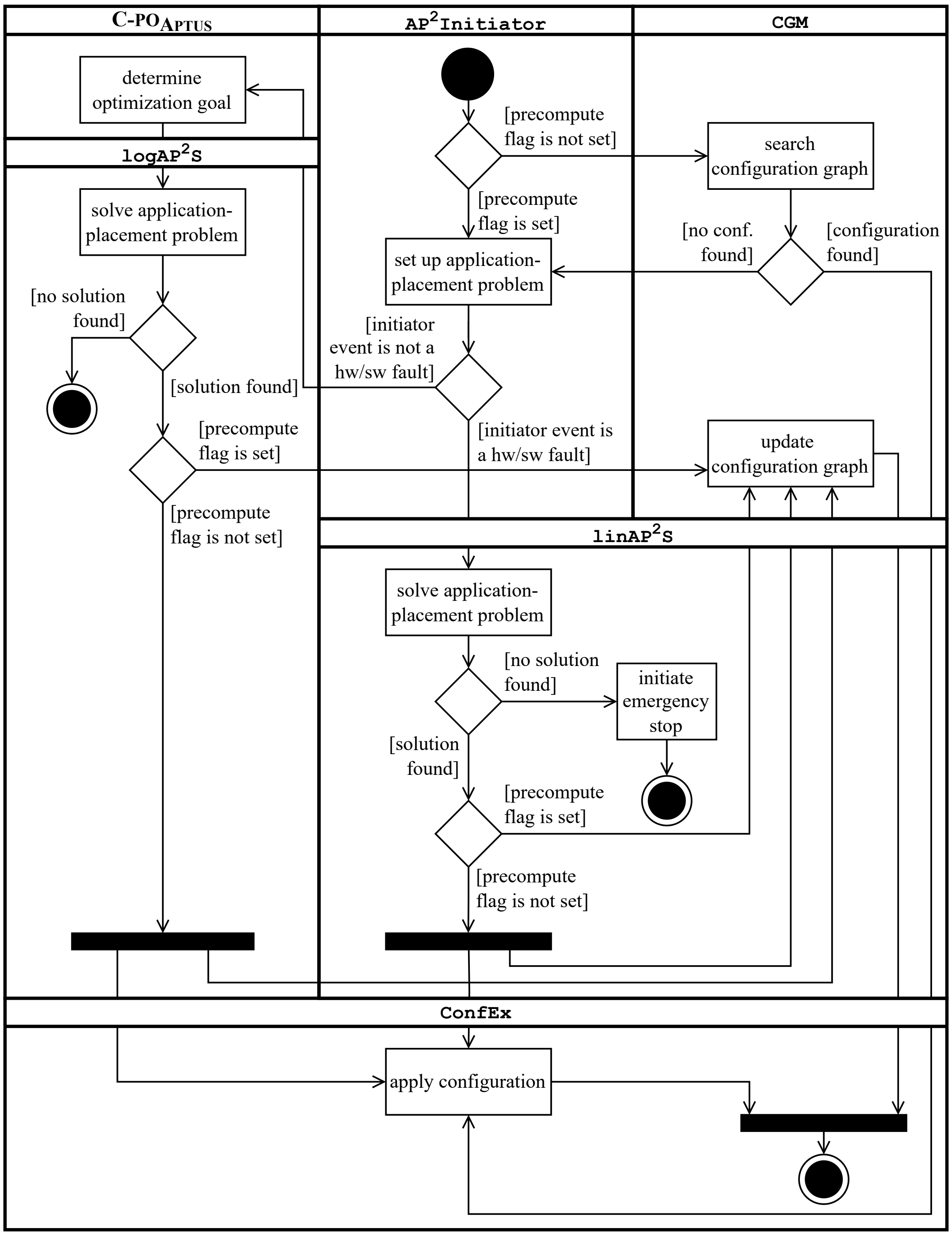

C-POAPTUS
AP²Initiator
CGM
determine optimization goal
logAP²S
solve application-placement problem
[no solution found]
[solution found]
[precompute flag is set]
[precompute flag is not set]
[precompute flag is not set]
[precompute flag is set]
set up application-placement problem
[initiator event is not a hw/sw fault]
[initiator event is a hw/sw fault]
search configuration graph
[no conf. found]
[configuration found]
update configuration graph
linAP²S
solve application-placement problem
[no solution found]
initiate emergency stop
[solution found]
[precompute flag is set]
[precompute flag is not set]
ConfEx
apply configuration

Figure 4.2: The activity diagram showing the workflow of the reconfiguration layer.

Apart from changing software-architecture requirements and occurring software and hardware faults, AP^{2}Initiator can also be requested by the reconfiguration layer to initiate the computation of a configuration. The component that initiates these requests is the so-called CGM, standing for "configuration-graph manager". CGM is responsible for managing the configuration graph, which is introduced in Section 4.2. The activities of CGM include, among others, intrinsically extending the configuration graph. To achieve this task, CGM initiates the computation of new configurations which are not yet part of the configuration graph. Requests for computing new configurations that are triggered by CGM are marked by the precompute flag. In all other cases, the precompute flag is not set.

After AP^{2}Initiator is instructed to determine a new configuration the precompute flag is checked. If the precompute flag is not set then AP^{2}Initiator notifies CGM about the new request. CGM then tries to find, based on the currently applied configuration, the initiator event, and the software-architecture requirements, whether the configuration graph holds the requested configuration. If this is the case, then the so-called ConfEx, standing for "configuration executor", applies the configuration found in the configuration graph.

If the required configuration is not part of the configuration graph, AP^{2}Initiator sets up the application-placement problem using the input parameter. Depending on the initiator event, AP^{2}Initiator then selects the application-placement problem solver for computing the configuration.

Recall that as defined in Section 4.2, an initiator event $\iota \in I$ is either reporting faults of applications and computing nodes, indicating updated software-architecture requirements, or triggering a configuration optimization. In case the initiator event indicates the fault of applications or computing nodes AP^{2}Initiator instructs linAP^{2}S to solve the application-placement problem. As mentioned before, the faults of applications and computing nodes are initially handled by FDIRO. However, the reconfiguration step of FDIRO aims to recover compliance with the current software-architecture requirements and, therefore, instructs the reconfiguration layer to determine a new configuration. As a violation of software-architecture requirements can cause a drop in dependability, countermeasures must be applied quickly. Consequently, we employ the application-placement problem solver linAP^{2}S. The solving unit of linAP^{2}S is based on integer linear programming and implements a single optimization function that enables short solving times.

On the other hand, if the initiator event specifies updated software-architecture requirements, the reactivation of a computing node, or the request to optimize the current configuration, AP^{2}Initiator selects logAP^{2}S to solve the application-placement problem. The solving unit of logAP^{2}S employs answer-set programming and is able to apply multiple optimization functions. In contrast to the application-placement problems handled by linAP^{2}S, those assigned to logAP^{2}S require a less time-critical processing. Therefore, logAP^{2}S can consider multiple parameters when optimizing configurations. Furthermore, the applied optimization functions are not restricted to being linear.

Selecting optimization functions that suit the current context is not a trivial task. Thus, before `logAP²S` is instructed to solve the application-placement problem, C-PO$_{\text{Aptus}}$ determines the optimization goals, which are later applied in the solving process of `logAP²S`.

After `linAP²S` and `logAP²S`, respectively, solved the application-placement problem, the resulting configuration is supplied to `CGM`. The newly computed configuration is added to the configuration graph by `CGM`. Furthermore, `CGM` uploads the configuration to `GCG`. Note that this activity, as well as other tasks performed by `CGM`, are not illustrated in Figure 4.2 as they are executed in parallel.

In the last step, if the precompute flag is not set, the determined configuration is applied by `ConfEx`. This component determines a set of commands that transfers the current system state to a new configuration.

In the following sections, we discuss `linAP²S`, `logAP²S`, C-PO$_{\text{Aptus}}$, and `CGM`. Note that we do not present implementations for `AP²Initiator` and `ConfEx`. This is because `ConfEx` depends on the underlying vehicle architecture and must be individually implemented for each one. `AP²Initiator` can be implemented straightforwardly, as it only needs to transform the output of the context layer into the input formats required by `linAP²S` and `logAP²S`, and call other procedures.

4.4 The Application-Placement Problem Solver linAP²S

The application-placement problem solver `linAP²S`, standing for "linear application-placement problem solver", is specifically designed to perform the recovery procedure of our fault-tolerance approach FDIRO which is described in Section 5.3. As mentioned before, the recovery procedure of FDIRO aims to fulfill the current software-architecture requirements. Furthermore, the recovery procedure of FDIRO requires a fast reconfiguration of the system. Therefore, the solving time of `linAP²S` and the *configuration-rollout time*, i.e., the time that is required to apply a computed configuration, have to be low.

The solving approach implemented by `linAP²S`, which is developed in Python, is *integer linear programming* [196], a problem formulation that is restricted to linear conditions and objective functions. The framework used to specify our problem is the constraint-optimization framework OR-Tools, developed by Google and available at

`https://developers.google.com/optimization.`

To support short configuration-rollout times, `linAP²S` implements an optimization function that aims for new application placements which minimize the number of application displacements. We consider an application to be displaced if it is assigned to two distinct computing nodes in two consecutive configurations. A low number of application displacements results in a low number of reconfiguration actions, such as launching or terminating an application, that have to be performed in order to apply the new configuration. Since executing reconfiguration actions is time-consuming, configurations with fewer application displacements can be applied more quickly.

`linAP`^{2}`S` determines a new configuration based on a set of software-architecture requirements, S, which is represented as a 9-tuple of the form $(A, F, N, R, \Phi, \Omega, \phi, \omega, \pi)$, whose elements are defined as follows:

- A is the set of active, active-low, and active-hot applications;
- F is a set of functions, where each application $a \in A$ belongs to exactly one function $f \in F$;
- N is the set of computing nodes;
- R is the *placement restriction function*, which specifies whether a computing node $n \in N$ fulfills all software requirements of $a \in A$, defined by setting $R(n, a) = 1$ if n fulfills the requirements of a and $R(n, a) = 0$ otherwise (we write $R_{a,n}$ for $R(n, a)$ in what follows);
- Φ is the function which assigns each $n \in N$ its memory capacity $\Phi(n) = \Phi_n$ in megabytes;
- Ω is the function which assigns each $n \in N$ its performance capacity $\Omega(n) = \Omega_n$ in TFLOPS;
- ϕ is the function which assigns each $a \in A$ its memory demand $\phi(a) = \phi_a$ in megabytes;
- ω is the function which assigns each $a \in A$ its performance demand $\omega(a) = \omega_a$ in TFLOPS; and
- π is the function assigning each $f \in F$ the minimum number $\pi(f) = \pi_f$ of computing nodes F has to run on, i.e., the level of separation of f.

Based on the set of software-architecture requirements S, `linAP`^{2}`S` computes a configuration function C which specifies whether a computing node $n \in N$ executes an application $a \in A$ by setting $C(n, a) = 1$ if n executes a and $C(n, a) = 0$ otherwise. Similar to the representation of the placement restriction function, we will use $C_{a,n}$ to stand for $C(n, a)$.

Since all elements of the configuration function C are binary, at most $2^{|A|\cdot|N|}$ potential solutions exist. Valid solutions, however, must satisfy the conditions (C_1)–(C_5) from Section 4.1, which can be modeled in terms of linear constraints, described as follows.

First of all, conditions (C_1)–(C_4) are expressed in the following way:

$$\forall a \in A : \sum_{n \in N} C_{a,n} = 1. \tag{4.1}$$

$$\forall n \in N: \sum_{a \in A} \phi_a \cdot C_{a,n} \leq \Phi_n. \tag{4.2}$$

$$\forall n \in N: \sum_{a \in A} \omega_a \cdot C_{a,n} \leq \Omega_n. \tag{4.3}$$

$$\forall a{\in}A, \forall n{\in}N\colon \text{if } R_{a,n} = 0, \text{ then } C_{a,n} = 0. \tag{4.4}$$

Equation (4.1), formalizing (C_1), assures that each application is executed by exactly one computing node. Equation (4.2) and Equation (4.3), formalizing in turn (C_2) and (C_3), ensure that the memory capacity and the computing performance of each computing node is not exceeded by the applications running on it. Finally, Equation (4.4), capturing (C_4), guarantees that an application can only run on a computing node that provides the software required by the application.

For expressing condition C_5, we introduce an ancillary variable $h_{a,n}$ for each function $f{\in}F$ and computing node $n{\in}N$, which is defined as follows:

$$\forall f{\in}F, \forall n{\in}N\colon h_{f,n} = \begin{cases} 1, & \text{if } \sum_{a \in f} C_{a,n} \geq 1, \\ 0, & \text{otherwise.} \end{cases}$$

With the use of these variables, we can formalize condition (C_5) by the following four linear expressions:

$$\forall f \in F, \forall n \in N: h_{f,n} = 0 \text{ or } h_{f,n} = 1, \tag{4.5}$$

$$\forall f \in F, \forall a \in f, \forall n \in N: C_{a,n} \leq h_{f,n}, \tag{4.6}$$

$$\forall f \in F, \forall n \in N: \sum_{a \in f} C_{a,n} \geq h_{f,n}, \text{ and} \tag{4.7}$$

$$\forall f \in F: \sum_{n \in N} h_{f,n} \geq \pi_f. \tag{4.8}$$

Equation (4.5) enforces the binary nature of $h_{f,n}$, i.e., restricting it to the values 0 and 1. Equation (4.6) ensures that $h_{f,n} = 1$ holds if at least one application $a \in f$ is assigned to computing node n. On the other hand, Equation (4.7) guarantees that $h_{f,n} = 0$ holds if no application $a \in f$ is running on computing node n. Finally, Equation (4.8) ensures that the applications implementing function f are distributed across at least π_f distinct computing nodes to satisfy the required level of separation.

To discriminate among the different solutions, specifying which solutions are desired the most, we instruct the solver to find an application placement which maximizes the following optimization goal, where $\overline{C}$ represents the current node assignment which is provided by the underlying vehicle platform:

$$\sum_{a \in A, n \in N} C_{a,n} \times \overline{C}_{a,n}.$$

Due to this optimization goal, application placements that minimize the number of application displacements are preferred. Recall that `linAP`^{2}`S` aims for a low number

Table 4.1: Average solving time required by `linAP²S` for a randomized test case for which a solution exists.

Problem-Size Class	Solving Time
$\|N\| = 3,\ \|A\| = 30$	1.7 ms
$\|N\| = 3,\ \|A\| = 60$	3.6 ms
$\|N\| = 3,\ \|A\| = 90$	5.3 ms
$\|N\| = 3,\ \|A\| = 120$	7.1 ms

of application displacements since displacements are considered to be time-consuming. Note that the term $C_{a,n} \times \overline{C}_{a,n}$ models the application displacements by indicating if an application a remains assigned to the same computing node n in both the current configuration $\overline{C}$ and the computed configuration C. An application a is considered displaced if $C_{a,n} \neq \overline{C}_{a,n}$.

The source code of `linAP²S` is provided in Appendix A.2.

4.4.1 Performance Evaluation

As mentioned before, `linAP²S` is required to compute a new configuration, which introduces lost redundancies within a short time. Therefore, we conducted a performance evaluation to test the solving time of `linAP²S`.

The computer used to run the performance test was equipped with an Apple M2 chip with 8 cores and 8 GB memory.

Since the computational effort of solving the application-placement problem strongly depends on the size of the problem, we defined four problem-size classes for which we fixed the number of computing nodes at 3 while the number of executed applications ranged from 30 to 120. One-third of the applications are active applications, while the others are active hot, whereby each active application requires a level of redundancy of 2.

Each problem-size class consists of 1,000 randomized test cases. The average solving times of the different problem-size classes are shown in Table 4.1.

This table shows that the correlation between the number of applications and the average solving time is almost linear. Note that all test cases are designed such that a valid solution exists. However, as not for all application-placement problem instances valid solutions exist, we designed sets of randomized test cases for which no solution exists. The results of these test runs are shown in Table 4.2.

These results indicate an exponential correlation between the number of applications and the average solving time in case no solution exists, reflecting the NP-hardness of the general problem.

To potentially reduce the computing time, we adopted a parallel-solving heuristic as described next.

Table 4.2: Average solving time required by `linAP`^{2}`S` for a randomized test case for which no solution exists.

Problem-Size Class	Solving Time
$\|N\| = 3,\ \|A\| = 30$	9 ms
$\|N\| = 3,\ \|A\| = 33$	25 ms
$\|N\| = 3,\ \|A\| = 36$	85 ms
$\|N\| = 3,\ \|A\| = 39$	199 ms
$\|N\| = 3,\ \|A\| = 42$	634 ms
$\|N\| = 3,\ \|A\| = 45$	1,543 ms

4.4.2 Parallel-Solving Heuristic

To increase the chances of obtaining a valid solution, we divide a given set A of applications into several subsets. These subsets are computed based on the priorities of the functions implemented by those applications. To build the priority-based application subsets, we introduce the function H, which determines all applications that implement a function of a given priority, i.e., HIGH, MEDIUM, and LOW.

Using these priority classes, `linAP`^{2}`S` computes the following subsets of applications:

$$\begin{aligned}
A_0 &= A = H(A, \mathit{HIGH}) \cup H(A, \mathit{MEDIUM}) \cup H(A, \mathit{LOW}),\\
A_1 &= H(A, \mathit{HIGH}) \cup H(A, \mathit{MEDIUM}),\ \text{and}\\
A_2 &= H(A, \mathit{HIGH}).
\end{aligned}$$

For each of those three subsets, `linAP`^{2}`S` starts a thread that computes an application placement that considers all applications that are part of the respective subset. Since $A_2 \subseteq A_1 \subseteq A_0$ holds, we can assume, as illustrated in Figure 4.3, that the time required to find a valid application placement is highest for A_0 and lowest for A_2.

We conducted an experiment to justify the assumption that the solving time is highest for the subset A_0 and lowest for A_2. As before, we examined different problem-size classes. Each problem-size was evaluated by running 1,000 randomized test classes. The applications of the individual test cases were evenly distributed among the three priority classes. The results, illustrated in Table 4.3, show that the assumption that the solving time is highest for the subset A_0 and lowest for the subset A_2.

Furthermore, it holds that in case a valid application placement for subset A_i exists, for $0 \leq i \leq 2$, a solution for subset A_{i+1} exists as well. On the other hand, if for subset A_i no solution exists, we can infer that also for each subset A_{i-j}, for $1 \leq j \leq i$, no solution exists.

After the termination of all threads, `linAP`^{2}`S` selects the best available solution, whereby an application placement that considers all applications of the subset A_0 is the most desired solution and an application placement that only maps applications of priority class HIGH is referred to as the *worst-case solution*.

Table 4.3: Average solving time of 1000 test cases for the subset A_0, A_1, and A_2. For all test cases, a solution exists. The applications are evenly distributed among the three priority classes, i.e., *HIGH*, *MEDIUM*, and *LOW*.

Problem-Size Class	Solving Time A_0	Solving Time A_1	Solving Time A_2
$\|N\| = 3,\ \|A\| = 30$	2.0 ms	1.1 ms	0.6 ms
$\|N\| = 3,\ \|A\| = 60$	3.9 ms	2.2 ms	1.1 ms
$\|N\| = 3,\ \|A\| = 90$	5.6 ms	3.6 ms	1.8 ms
$\|N\| = 3,\ \|A\| = 120$	7.5 ms	4.7 ms	2.4 ms

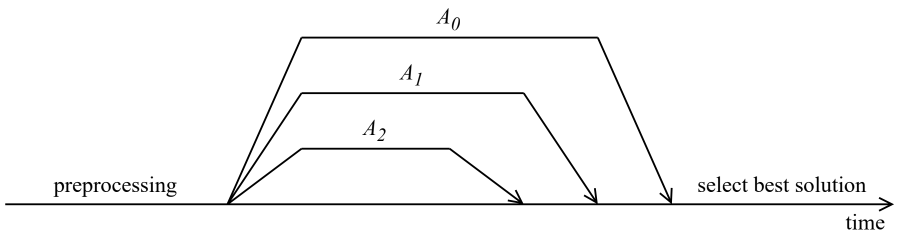


Figure 4.3: Multiple threads for determining the application-placement problem for subsets of applications.

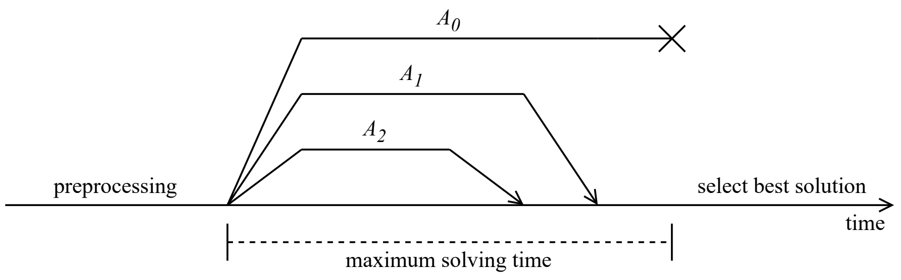


Figure 4.4: Maximum solving time causing two application-placement problem solver threads to abort.

Besides allowing to find an application placement for the most important applications in case the system cannot run all applications, this priority-based approach also allows to define an upper bound for the solving time for the application placement. As illustrated in Figure 4.4, defining a maximum solving time causes the application-placement problem solver threads that cannot find a valid solution within the specified time to be aborted.

Defining a maximum solving time is desirable since some safety-critical situations require a new configuration within a guarded time rather than a placement that considers all applications. In this version of Aptus, it is the responsibility of the system architect to define the maximum solving time at design time. However, it is also conceivable to adapt the maximum solving time in future versions of Aptus according to the current context.

If finding a solution for any of the three subsets is not feasible, FDIRO will initiate a safe stop of the vehicle.

4.5 The Application-Placement Solver logAP^{2}S

We next discuss the details of the logic-based application-placement solver logAP^{2}S. Like C-SAR, logAP^{2}S is an Python program that uses clyngor to deploy answer-set programming.

The software-architecture requirements determined by the context layer constitute the input of logAP^{2}S. Furthermore, logAP^{2}S takes as input the set of currently available computing nodes as well as their memory and performance capacity.

The available computing nodes are defined by the predicate `computing_node`/1, which holds as an argument the unique name of the computing node. Furthermore, the memory and performance capacity are specified similarly to the memory and performance demands of applications using the predicates `memory`/2 and `performance`/2, respectively. The first argument indicates the computing node and the second specifies the parameter value. Recall that the memory capacity is given in Megabytes and the performance capacity in TFLOPS. In addition, we introduce the predicate `provided_supporting_software`/2 to specify which supporting software is installed on the individual computing nodes. This predicate has two arguments: the first identifies the computing node, and the second specifies the corresponding supporting software. Note that on a computing node, several supporting software can be installed.

Besides the software-architecture requirements and the specification of the available computing nodes, logAP^{2}S might require additional input parameters depending on the implemented optimization functions. For the sake of simplicity, we only consider the application placement of the current system state as an additional input parameter. Therefore, we introduce the predicate `current_assignment`/3. The first argument of this predicate holds the name of an application, and the second argument defines the redundancy-instance number. The third argument indicates which computing node executes this application. We will use this predicate to implement the optimization function, which minimizes the displacement of the currently executed applications.

The output of logAP^{2}S is an assignment of applications to computing nodes. To indicate the desired assignment, we introduce the predicate `assignment`/2. Like the previously defined `current_assignment` predicate `assignment` holds two terms. The first term specifies the name of an application, and the second term indicates which computing node shall execute this application.

In order to determine a valid assignment, logAP^{2}S has to implement condition (C_1)–(C_5) defined in Section 4.1.

The first condition of the application-placement problem expresses that an application has to be executed by exactly one computing node. To implement this condition, we introduce the following two rules:

```
 {assignment(APP, R_INST, CN): computing_node(CN)} = 1 :-
     active_application(APP, R_INST).
{assignment(APP, R_INST, CN): computing_node(CN)} = 1 :-
     active_low_application(APP, R_INST).
{assignment(APP, R_INST, CN): computing_node(CN)} = 1 :-
     active_hot_application(APP, R_INST).
```

Note that the cardinality constraints ensure that each active, active-low, and active-hot application is assigned to exactly one computing node.

Besides this condition, the application-placement problem also states that the memory capacities of computing nodes have to be respected. Therefore, logAP^{2}S defines the following ASP constraint:

```
:- computing_node(CN), memory(CN, MEM),
     #sum{M, APP, R_INST: memory(APP, M),
                          application(APP, R_INST),
     assignment(APP, _, CN)} > MEM.
```

Likewise, logAP^{2}S has to comply with the performance capacity limitations of computing nodes. Consequently, we add the following constraint:

```
:- computing_node(CN), performance(CN, PERF),
     #sum{C, APP, R_INST: performance(APP, C),
                          application(APP, R_INST),
     assignment(APP, _, CN)} > PERF.
```

Furthermore, logAP^{2}S has to ensure that applications are only assigned to computing nodes that fulfill the supporting software demands of the assigned applications. This is implemented by the following rule:

```
:- assignment(APP, _, CN), req_supporting_software(APP, SW),
     not provided_supporting_software(CN, SW).
```

Finally, to ensure that the demanded level of separation of the application is fulfilled, we add the following two constraints:

```
:- active_application(A_APP, _),
     application(A_APP, FUNC), separation(A_APP, SEP),
     #count{CN: application(APP, FUNC),
                assignment(APP, _, CN)} < SEP.
```

```
:- active_low_application(AL_APP, _),
   application(AL_APP, FUNC), separation(AL_APP, SEP),
   #count{CN: application(APP, FUNC),
              assignment(APP, _, CN)} < SEP.
```

As already noted in Section 4.1, to discriminate among a potentially large number of solutions, optimization functions that specify which valid assignments are desired the most can be defined. However, different contextual conditions require different optimization functions. Therefore, we introduce in Section 4.6 an approach to select optimization functions based on the current context. The selected optimization function is then added to the ASP program of logAP^{2}S.

4.5.1 Performance Evaluation

In order to compare the performance of logAP^{2}S and linAP^{2}S, we conducted the same performance tests as introduced in Subsection 4.4.1 using logAP^{2}S.

As mentioned before, logAP^{2}S is designed to implement various optimization functions. On the other hand, linAP^{2}S implements only one static optimization function, which aims to minimize the number of application displacements. Therefore, we incorporated, for this performance evaluation the same optimization function into logAP^{2}S. Note that the source code of logAP^{2}S used for this performance evaluation is presented in Appendix A.3.

To incorporate this optimization function into logAP^{2}S we introduce a new predicate called displacement_count/1, whereby the argument specifies how many applications are displaced. This predicate is determined by comparing the planned assignment and the current assignment as the following rule shows:

```
displacement_count(CNT) :-
   CNT = #count{APP, R_INST: assignment(APP, R_INST, CN),
                             current_assignment(APP_CUR,
                                                R_INST_CUR,
                                                CN_CUR),
                            APP = APP_CUR,
                            R_INST = R_INST_CUR,
                            CN != CN_CUR}.
```

To instruct the ASP solver to find the answer set that minimizes the number of displacements, we introduce the following optimization statement:

```
#minimize{CNT:displacement_count(CNT)}.
```

Table 4.4: Average solving time required by `logAP`^{2}`S` for a randomized test case for which a solution exists.

Problem-Size Class	Solving Time
$\lvert N\rvert = 3,\ \lvert A\rvert = 30$	6.3 ms
$\lvert N\rvert = 3,\ \lvert A\rvert = 60$	8.9 ms
$\lvert N\rvert = 3,\ \lvert A\rvert = 90$	11.4 ms
$\lvert N\rvert = 3,\ \lvert A\rvert = 120$	14.7 ms

Table 4.4 illustrates the average solving time of the different problem-size classes. Recall that each problem-size class consists of 1,000 randomized test cases.

This performance evaluation shows that `logAP`^{2}`S` is capable of finding solutions for the defined problem-size class within a few milliseconds. However, a comparison of the performance results for `linAP`^{2}`S`, provided in Table 4.1, with those for `logAP`^{2}`S` reveals that the average solving time of `logAP`^{2}`S` is higher than that of `linAP`^{2}`S`. On average, `linAP`^{2}`S` is 2.6 times faster than `logAP`^{2}`S`. Due to the lower solving times of `linAP`^{2}`S`, in APTUS we use `linAP`^{2}`S` in case a quick reconfiguration is required after a hardware or software fault.

Note that we did not modify the parameters of the the solvers used by `linAP`^{2}`S` and `logAP`^{2}`S`. Optimizing these parameters may lead to a further reduction of the solving times.

4.6 Context-Based Application Placement Optimization

As mentioned in Section 4.1, an application placement can be optimized according to various parameters, whereby, in the case of autonomous vehicles, the optimal application placement strongly depends on the current context. For instance, if the vehicle has encountered safety-critical faults, the goal is to determine an application placement that restores the level of safety. On the other hand, if the vehicle is in optimal condition, the application placement shall be optimized according to the optimization goal determined by C-SAR.

To enable the configuration layer of APTUS to determine a context-based optimization function for the application-placement problem, we incorporate a concert implementation of C-PO. The name C-PO stands for "context-based placement optimization" and the general idea of C-PO was introduced in our prior work [116].

Generally speaking, different configurations of the configuration graph fulfill different safety and reliability properties. The goal of an autonomous vehicle should be to achieve the highest possible level of safety and reliability to satisfy customer requirements. Consequently, an autonomous vehicle should strive to execute a configuration that maximizes the safety and reliability properties.

The idea of C-PO is to subdivide the configuration graph into multiple levels, whereby the individual levels indicate different safety and reliability properties of the associated configurations. For each layer, C-PO requires the definition of an optimization function that aims to reach a better level. Once the highest safety level is reached, the optimization function determined by C-SAR is used.

Optimizing the application placement is a well-known research challenge. For instance, Kumar et al. [136] discuss a reconfiguration and placement optimization approach for platoons of autonomous vehicles based on bond-graph modeling. By determining offsets during operation, measures are taken at different levels of the system. The measures can range from switching off individual wheels to switching off entire vehicles.

Cooray et al. [52] introduce a framework, called RESIST, for a proactive reconfiguration for situated software systems. This framework aims to maintain the reliability of the systems due to a dynamic optimization of the architectural configuration.

Lapouchnian et al. [140] employ a goal-oriented requirements-engineering approach to adapt a system dynamically to react to occurring environmental changes.

Furthermore, Hemmati et al. [91] study the *redundancy-allocation problem*, which faces similar optimization challenges as the application-placement problem. The idea there is to use a multi-objective harmonic search algorithm to determine an optimal redundancy configuration to maximize the mean time to the first failure.

To distinguish the configurations according to their safety and reliability properties, we define in what follows a level-based classification of configurations that is used by C-PO.

4.6.1 The General Method of the Context-Based Application Placement Optimization

As mentioned before, the idea of C-PO is to add a layer on top of the configuration graph. This layer, as illustrated in Figure 4.5, subdivides the configuration graph into multiple levels, whereby a level number, $l \in \mathbb{N}$, for which $0 \leq l \leq N$ holds, identifies each level, where $N \in \mathbb{N}$ is the number of levels. Each configuration of the configuration graph is assigned to one level.

To classify the individual configurations of the configuration graph, we define the function $\Lambda : \Gamma \to \mathbb{N}$. This function has to fulfill the following three requirements:

RQ_1: For each level, specific properties that a configuration of that level has to fulfill need to be defined (like, e.g., minimal redundancy requirements). Furthermore, the levels have to be mutually exclusive, i.e.,

$$\nexists C \in \Gamma : \Lambda(C) = x \ \wedge \ \Lambda(C) = y \ \wedge \ x \neq y$$

has to hold.

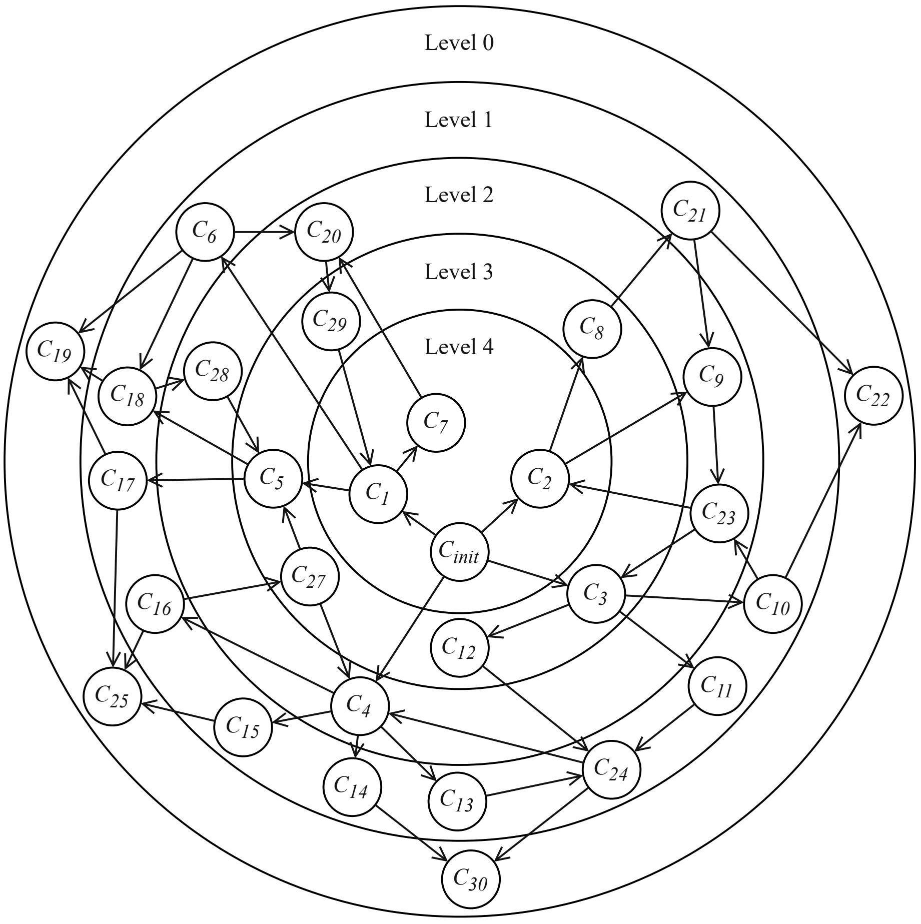


Figure 4.5: Visualization of the level-based safety and reliability classification of configurations. The layers subdivide the configuration graph into N levels, where in this example $N = 4$.

RQ_2: The levels need to be defined such that the safety and reliability of the system does not decline as the level number increases. Level N can be considered as the "best" level, meaning that this level is the most desired one. Accordingly, level 0 is the "worst" level. Since in this level, the minimal safety and reliability requirements are not satisfied anymore, an emergency system has to take over control of the vehicle and bring it to a safe stop.

RQ_3: Edges to configurations that are associated to level 0 may only originate from level 1. This means that it has to be excluded that a triggering event resulting from a system context change originating from a configuration of level 2 or above causes a drop to level 0, i.e.,

$$\nexists C \in \Gamma,\ \exists T \subseteq \Omega : \Lambda(C) > 1 \wedge F(C,T) = C' \wedge \Lambda(C') = 0$$

has to hold.

4.6.2 The Context-Based Application Placement Optimization Instance used in our Framework

Based on the general method of C-PO, we now introduce the concrete instance that is incorporated into the reconfiguration layer, which we refer to as C-PO$_{\text{Aptus}}$. The function Λ that is applied by C-PO$_{\text{Aptus}}$ uses the function priority classes *HIGH*, *MEDIUM*, and *LOW*, which are introduced in Section 3.3.1, to define the levels. Consequently, we call this function Λ_{prio}.

We set N, the number of levels, to 4. The levels are specified as follows:

- *Level 0*: At least one function of priority *HIGH* is not executed, i.e., no application that implements this function is executed by a computing node.
- *Level 1*: All functions of priority *HIGH* are executed, i.e., at least one application of each function of priority *HIGH* is executed. However, the minimum redundancy or separation requirement of at least one active application or active-low application that implements a function of priority *HIGH* is not fulfilled.
- *Level 2*: The minimum redundancy and separation requirements of all functions of priority *HIGH* are fulfilled, i.e., the required number of active-hot applications are executed and they are separated among the required number of computing nodes. However, the minimum redundancy or separation requirement of at least one active application or active-low application that implements a function of priority *MEDIUM* is not fulfilled.
- *Level 3*: The minimum redundancy and separation requirements of all functions of priority *HIGH* and *MEDIUM* are fulfilled. However, the minimum redundancy or separation requirement of at least one active application or active-low application that implements a function of priority *LOW* is not fulfilled.
- *Level 4*: The minimum redundancy and separation requirements of all active application and active-low application that implements a function of priority *HIGH*, *MEDIUM*, and *LOW* are fulfilled.

As a prerequisite, we demand that for functions of priority class *HIGH*, at least one diverse redundant application and a level of separation of at least 2. Furthermore, we require that the computing nodes are diverse.

Λ_{prio} fulfills RQ_1 as each level defines for the individual priority classes whether the redundancy and separation requirements must be fulfilled. Furthermore, the levels are mutually exclusive.

From level 0 to level 2, the level of redundancy and the level of separation of functions of priority *HIGH* is increasing. Note that the foundation of fault tolerance is separated redundancies, as explained in Section 5.1. An improved fault tolerance, in return, improves the safety and the reliability. Consequently, Λ_{prio} fulfills the requirement RQ_2.

Requirement RQ_3 holds, since, for levels 2, 3, and 4, we have that for all functions of priority class *HIGH*, the minimum redundancy, the minimum diversity level, and the minimum separation level is 2. A drop to level 0 requires that at least one function of priority class *HIGH* is not executed. The triggering events that we consider are the fault of any application that implements a function of priority *HIGH* and the fault of a computing node which executes an application that implements a function of priority *HIGH*. We show that both these triggering events do not cause a drop to level 0 if a configuration is in level 2, level 3, or level 4:

- In case an application that implements a function of priority *HIGH* fails, a configuration C for which $\Lambda_{prio}(C) \geq 2$ holds does not drop to level 0 as a redundant application of that function exists.
- In case a computing node fails that executes an application which implements a function of priority *HIGH*, a configuration C for which $\Lambda_{prio}(C) \geq 2$ holds does not drop to level 0 as a redundant application of that function which is executed on a separate computing node exists.

Note that we do not consider simultaneous random faults of two applications that implement the same function, which is of priority *HIGH*. Since the applications are diverse and executed on two separate computing nodes the probability of such a simultaneous random fault is limited. The likelihood of such an event can be further lowered by a diverse design of the computing nodes.

4.6.3 The Optimization Functions of our Instance

According to the definition of C-PO, C-PO$_{\text{APTUS}}$ has to specify optimization functions for the individual layers of Λ_{prio}. Since those optimization functions are incorporated by `logAP`^{2}`S`, they are represented in the form of ASP optimization statements.

Recall that level 4 of Λ_{prio} is the most desired level. Therefore, the goal of all the other levels is to get as fast as possible to level 4. This can be achieved by an optimization function that prioritizes application placements that fulfill as many properties requested by the next level as possible. The optimization functions of levels 1, 2, and 3 are defined as in Table 4.5.

In order to incorporate these optimization goals in `logAP`^{2}`S`, we relaxed condition (C_1). Instead of demanding that each application is assigned to exactly one computing node, we define that an application is assigned to at most one computing node. Therefore, it becomes feasible to assign only applications that implement a function of a specific priority class. The following rules show the easing of the previously introduced rules for constraint (C_1):

Table 4.5: Context-based application-placement optimization functions for level 1, level 2, and level 3.

Level	Optimization Function
1	Try to fulfill the minimum redundancy and separation requirements of all functions of priority *HIGH*.
2	Try to fulfill the minimum redundancy and separation requirements of all functions of priority *HIGH* and *MEDIUM*.
3	Try to fulfill the minimum redundancy and separation requirements of all functions of priority *HIGH*, *MEDIUM*, and *LOW*.

```
{assignment(APP, R_INST, CN): computing_node(CN)} <= 1 :-
    active_application(APP, R_INST).
{assignment(APP, R_INST, CN): computing_node(CN)} <= 1 :-
    active_low_application(APP, R_INST).
{assignment(APP, R_INST, CN): computing_node(CN)} <= 1 :-
    active_hot_application(APP, R_INST).
```

Note that, as defined in Table 4.5, the optimization functions of higher levels include the optimization functions of lower levels. For instance, if the optimization function of level 3 is fulfilled, then the optimization function of level 2 and level 1 is also satisfied. Therefore, we incorporated the optimization functions of levels 1, 2, and 3 into one joint optimization function, which includes three graded maximization goals.

In order to define the graded maximization goals, we introduce for each priority class a predicate:

- `assignment_count_prio_high`/1,
- `assignment_count_prio_medium`/1, and
- `assignment_count_prio_low`/1.

The arguments of these predicates indicate indicates how many applications that implement a function of the particular priority class are assigned to a computing node. The following rules specify these predicates:

```
assignment_count_prio_high(CNT) :-
    CNT = #count{APP, R_INST: assignment(APP, R_INST, _),
                              application(APP, FUN),
                              selected_function(FUN, PRIO),
                                            PRIO = 2}.

    assignment_count_prio_medium(CNT) :-
          CNT = #count{APP, R_INST: assignment(APP, R_INST, _),
                                    application(APP, FUN),
                                    selected_function(FUN, PRIO),
                                    PRIO = 1}.

    assignment_count_prio_low(CNT) :-
          CNT = #count{APP, R_INST: assignment(APP, R_INST, _),
                                    application(APP, FUN),
                                    selected_function(FUN, PRIO),
                                    PRIO = 0}.
```

Recall that the optimization function of level 1 states that the minimum redundancy and separation requirements of as many functions of priority *HIGH* as possible are fulfilled. `logAP`^{2}`S` already ensures that the separation requirements of functions are fulfilled if the corresponding applications are assigned to computing nodes. Consequently, to implement the optimization function of level 1, we introduce an optimization statement that maximizes the number of assignments for applications that implement a function of priority *HIGH*.

Recall that if an answer-set program contains multiple optimization statements, priorities can be introduced using the "`@`" annotation to define the significance of the individual optimization statements.

The rule that maximizes the number of assigned applications that implement a function of priority *HIGH* is defined as follows:

```
    #maximize{CNT@4: assignment_count_prio_high(CNT)}.
```

Building on this optimization statement, which is of priority 4, we introduce an optimization statement with a lower priority to implement the optimization function of level 2. The optimization statement that we add maximizes the number of assignments for applications that implement a function of priority *MEDIUM*:

```
    #maximize{CNT@3: assignment_count_prio_medium(CNT)}.
```

Likewise, we introduce an additional optimization statement, which builds on the previous two optimization statement, to implement the optimization function of level 1:

```
    #maximize{CNT@2: assignment_count_prio_low(CNT)}.
```

In addition to the optimization functions that aim to introduce the required number of applications, we define an optimization function for minimizing the number of displacements of applications. This optimization function aims to ensure that as few as possible already placed applications have to be displaced, thus avoiding dispensable displacements. In order to prevent this optimization goal from interfering with the overall goal of reaching level 4, we set its priority to 1. Note that this optimization function is identical to the one utilized for evaluating the performance of `logAP`^{2}`S` as described in Subsection 4.5.1:

```
match_count(CNT) :-
   CNT = #count{APP, R_INST: assignment(APP, R_INST, CN),
                             current_assignment(APP_CUR,
                                                R_INST_CUR,
                                                CN_CUR),
                             APP = APP_CUR,
                             R_INST = R_INST_CUR,
                             CN != CN_CUR}.

#minimize{CNT@1:match_count(CNT)}.
```

Once level 4 is reached, the optimization function determined by C-SAR can be applied. Therefore, C-PO$_{\text{APTUS}}$ holds for each optimization function specified by C-SAR the corresponding ASP implementation.

Table 4.6 illustrates the ASP implementation of some optimization functions, including those of the optimization functions defined in Subsection 3.3.3. These optimization functions aim to enhance different properties of the resulting configurations:

- The optimization function `min_comp_nodes` minimizes the number of computing nodes that execute applications. This optimization function might reduce the system's energy consumption as it is conceivable that computing nodes that do not execute applications can be shut down.
- In contrast to `min_comp_nodes`, the optimization function `max_comp_nodes` maximizes the number of computing nodes that execute applications. This optimization function might be used during the testing phase of the vehicle to ensure that all computing nodes are operational.
- The optimization function `min_displ_act` aims to minimize the displacement of active applications. Note that `min_displ_act` is similar to the optimization function used as part of the level 1, level 2, and level 3 optimization, which minimizes the displacements of active and active hot applications. Minimizing only the displacement of active applications might be desired since the displacement of active applications can impact the system's reliability and safety, especially if

the vehicle is driving. Furthermore, the solution space for possible subsequent optimization functions becomes larger by not demanding a minimum displacement of active-hot application. Consequently, these optimization functions are more likely to be satisfied.

- The optimization function `max_sep` aims to maximize the separation of functions regardless of the demanded level of separation. For instance, increasing the level of separation can improve the system's reliability in case of hardware faults.

The ASP implementation of the optimization functions are defined by the system architect at design time. Therefore, if a new optimization function is introduced in C-SAR, the corresponding ASP implementation has to be added in C-PO$_{\text{APTUS}}$ simultaneously. In case C-SAR determined multiple optimization functions, C-PO$_{\text{APTUS}}$ prioritizes the optimization statements according to the prioritization values specified by C-SAR.

Note that after a change in the software-architecture requirements, we assume that the Λ_{prio} level classification remains unaffected. However, if the application-placement problem cannot be solved using the determined optimization function, a new reconfiguration attempt is triggered using the subjacent level of Λ_{prio}.

4.7 The Configuration Graph Manager CGM

Computing new configurations is a resource intensive task as has been illustrated in the previous sections. Furthermore, autonomous vehicles that are equipped with the same hardware and software components are likely to compute the same configurations. Consequently, in order to avoid that vehicles compute the same configurations multiple times, we introduce `CGM`.

`CGM` consist, as illustrated in Figure 4.6, of two main components: (i) the *local configuration graph manager* and (ii) the *global configuration graph manager*. The former component is referred to as `locCGM`, and the latter is called `globCGM`.

The idea is that each vehicle that is equipped with APTUS runs a `locCGM` instance. These instances are responsible for managing the so-called *local configuration-graph*, which stores those configurations that are relevant in the current context of the vehicle. The `locCGM` instances are connected to `globCGM` in order to exchange configurations, whereby `globCGM` is responsible for managing the so-called *global configuration-graph*. The global configuration-graph persistently stores configurations that are provided by the `locCGM` instances.

As illustrated in Figure 4.6, `locCGM` and `globCGM` contain several subcomponents. In particular, `locCGM` consists of the *local-configuration graph database*, denoted by `locCGDB`, the *precompute component*, the *vehicle-synchronization component*, the *search component*, and the *update component*. Furthermore, `globCGM` contains the *global-configuration graph database*, called `globCGDB`, and the *global-synchronization component*.

In what follows, we describe the workflow of `locCGM` and `globCGM`.

Table 4.6: Different optimization functions for level 4 and their corresponding ASP implementation.

Opt. Func.	ASP Implimentation

`min_comp_nodes`	`used_computing_nodes(CNT) :-`
	`    CNT = #count{CN: assignment(_, _, CN)}.`
	`#minimize{CNT:used_computing_nodes(CNT)}.`

`max_comp_nodes`	`used_computing_nodes(CNT) :-`
	`    CNT = #count{CN: assignment(_, _, CN)}.`
	`#maximize{CNT:used_computing_nodes(CNT)}.`

`min_displ_act`	`act_app_displ_count(CNT) :-`
	`    CNT = #count{APP, R_INST:`
	`        assignment(APP, R_INST, CN),`
	`        current_assignment(APP_CUR,`
	`                           R_INST_CUR,`
	`                           CN_CUR),`
	`        active_application(APP, R_INST),`
	`        APP = APP_CUR, R_INST = R_INST_CUR,`
	`        CN != CN_CUR}.`
	`#minimize{CNT:act_app_displ_count(CNT)}.`

`max_sep`	`func_separation(FUNC, SEP) :-`
	`    selected_function(FUNC, _),`
	`    SEP = #count{CN: application(APP, FUNC),`
	`                     assignment(APP, _, CN)}.`
	
	`func_separation_sum(SEP_SUM) :-`
	`    SEP_SUM = #sum{SEP,FUNC:`
	`                  func_separation(FUNC, SEP)}.`
	
	`#maximize{SEP_SUM:`
	`          func_separation_sum(SEP_SUM)}.`

4.7.1 The Workflow of CGM

As specified before, the local configuration-graph is stored in the `locCGDB` database. Note that different database types, including, e.g., key-value stores, document-oriented

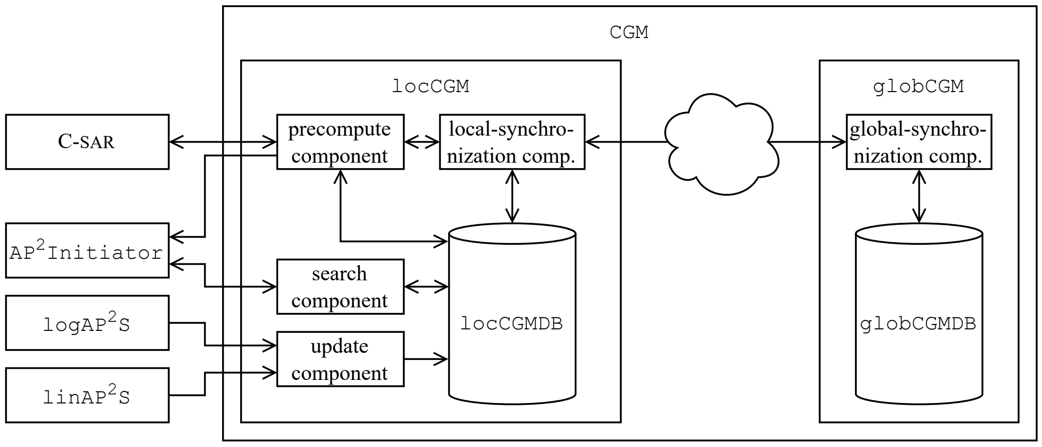


Figure 4.6: Overview of CGM and its sub-components.

databases, and graph databases, can be used to realize locCGDB. The configurations that are stored in locCGDB are inserted by the update component and the local-synchronization component. The update component, in turn, receives new configurations from the two application-placement problem solvers that are part of the reconfiguration layer, i.e., linAP^{2}S and logAP^{2}S. On the other hand, the local-synchronization component receives new configurations from the global-synchronization component.

Access to the locCGDB is provided via the search component. In particular, this component receives requests from AP^{2}Initiator. Those requests include, as mentioned in Section 4.3, the currently applied configuration, the initiator event, and the software-architecture requirements. Based on this information, the search component checks whether the requested configuration is stored in locCGDB. If this is the case, the search component returns the determined configuration. Otherwise, AP^{2}Initiator is informed that the requested configuration is not included in the local configuration-graph.

The precompute component continuously analyzes the local configuration-graph and determines whether a new configuration shall be precomputed. The aim of this component is to increase the probability that a requested configuration is part of the local configuration-graph. To accomplish that, the precompute component can implement various precomputation strategies. Figure 4.7 illustrates the options of a precomputation strategy in a given local-configuration graph.

We introduced two precomputation strategies previously [116], viz., the *naive precomputation-strategy* and the *informed reconfiguration-strategy*. The naive precomputation strategy aims to precompute a certain number of hierarchy levels of the local configuration-graph. On the other hand, the idea of the informed precomputation strategy is that the precomputation of reconfigurations is based on the probability of the occurrence of configuration transitions and the severity of those transitions.

In general, a precomputation strategy needs to select a successor configuration and

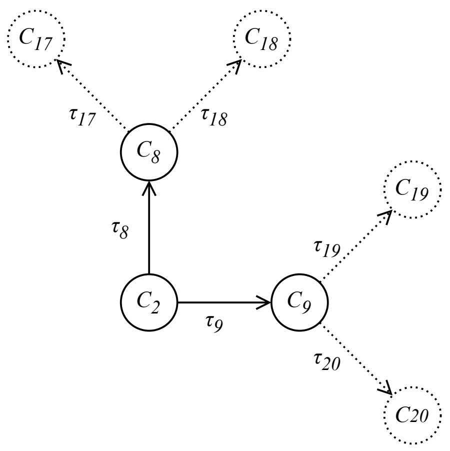


Figure 4.7: Excerpt of a local configuration-graph, including configurations that can be precomputed. Note that C_2, C_8, and C_9 represent configurations that are part of the local configuration graph. C_{17}, C_{18}, C_{19}, and C_{20} represent configurations that are not part of the local configuraiton-graph. Thus, they can be precomputed.

specify a configuration transition. Recall that the configuration transition is a tuple that holds the software-architecture requirements and the initiator event. As the software-architecture requirements are determined by C-SAR, we introduce an additional interface in C-SAR that allows the precomputation component to send requests. This interface takes as input a set contextual observations that are interpreted by C-SAR as the current context. Based on the assumed current context C-SAR computes the software-architecture requirements which are then provided to the precomputation component. Note that for the use case of precomputing configurations, C-SAR does not forward the determined software-architecture requirements to D-DEG.

Before the precomputation component instructs the computation of a new configuration, a request containing the successor configuration and the configuration transition is sent to the local-synchronization component. This component forwards the request to the global-synchronization component, which checks whether the needed configuration is already part of the global-configuration graph. If this is the case, then the requested configuration is sent to the local-synchronization component, which then adds it to the local-configuration graph. Otherwise, the local-synchronization component instructs the precompute component to continue its precomputation procedure.

To precompute configurations, the precompute component sends a request to the AP^{2}Initiator. This request includes the successor configuration, the software-architecture requirements, the initiator event, and the precompute flag which is set to true. AP^{2}Initiator then instructs, as described in Section 4.3, based on the initiator event either linAP^{2}S or logAP^{2}S to compute a new configuration. The new configuration is finally sent to the update component of CGM which adds it to the local-configuration graph.

Once a new configuration is added to the local-configuration graph, the local-synchronization component sends this configuration to the global-synchronization component. In turn, this component attaches the received configuration to the global-configuration graph.

4.8 Summary of the Reconfiguration Layer

In this chapter, we introduced the reconfiguration layer of Aptus, which implements the self-configuration property, the self-optimization property, and anticipativeness. It comprises four key components: `logAP`^{2}`S`, `linAP`^{2}`S`, C-PO$_{\text{Aptus}}$, and `CGM`.

The application-placement problem solvers `logAP`^{2}`S` and `linAP`^{2}`S` implement the self-configuration property. While `logAP`^{2}`S` employs answer-set programming, `linAP`^{2}`S` uses integer linear programming. We implemented two application-placement problem solvers to address the two main events that can trigger the computation of a new configuration: (i) requests generated by the context layer, detailed in the next chapter, following a context change and (ii) requests issued by the component layer in response to a software or hardware fault. The former scenario necessitates support for diverse, potentially nonlinear optimization goals, while the latter event demands a quick computation of configurations.

Our implementation of `logAP`^{2}`S` shows that it can fulfill the requirement to support various potentially nonlinear optimization goals. In particular, we implemented C-PO$_{\text{Aptus}}$ to determine the optimization function that shall be applied. As C-PO$_{\text{Aptus}}$ aims to continuously optimize the safety and reliability of the configurations, it implements the self-optimization property. Furthermore, we illustrated that `linAP`^{2}`S` can compute computations quickly.

The configurations computed by `logAP`^{2}`S` and `linAP`^{2}`S` are applied to the system by the configuration layer. Additionally, these configurations are provided to `CGM`, which is responsible for implementing anticipativeness. `CGM` facilitates the reuse of configurations by sharing them with other vehicles and expands the configuration graph by instructing `logAP`^{2}`S` and `linAP`^{2}`S` to precompute configurations.

CHAPTER 5

The Component Layer

In this chapter, we present the component layer of Aptus, which is responsible for implementing the self-healing property. The component layer consists of two components: Fdiro and Hrr. Fdiro [116, 122], developed as a precursor to this dissertation, defines four successive steps to transfer the system after an application fault or the fault of a computing node into a safe and optimized state. However, since Fdiro lacks the ability to recover failed hardware components such as computing nodes or sensors, only a limited number of hardware faults can be tolerated. To overcome this limitation, this chapter introduces the hardware redundancy-recovery approach Hrr, which extends Fdiro.

As both Fdiro and Hrr enhance the dependability of autonomous vehicles, we provide in the first part of this chapter foundational information on dependability. Next, in Section 5.2, we present a literature review on fault-tolerant approaches for autonomous systems in different domains. Afterwards, in Section 5.3, we provide an overview of the workflow of Fdiro. Moreover, Hrr is introduced in Section 5.4 and, in Section 5.5, we introduce AT-CARS, a tool that allows us to determine the reliability of an autonomous vehicle over time. Note that we use AT-CARS later on in Section 6.4 to analyze the impact of Fdiro and Hrr in a concrete use case. Finally, the chapter concludes with a brief summary.

5.1 Dependability

Autonomous vehicles, as well as other autonomous systems, have to be *dependable* as they are controlled by computer systems and not by human operators [158]. Dependability is defined as the capability of a system to avoid failures that are more frequent, more severe, or the resulting outages are longer than it is acceptable for the users of the system [6]. Broadly speaking, according to this definition, an autonomous vehicle is dependable in case its users can reasonably rely on the provided transportation service.

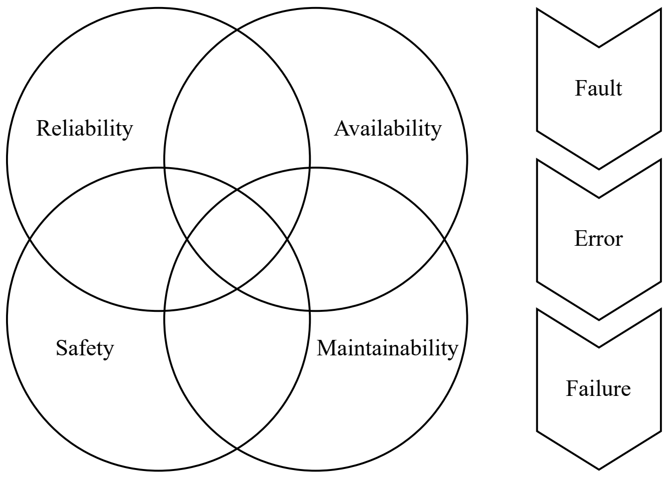


Figure 5.1: Visualization of the RAMS parameters and their threats.

5.1.1 Dependability Attributes

According to Aviziens et al. [6], dependability comprises several attributes, including, *reliability*, *availability*, *maintainability*, and *safety*. These attributes, collectively referred to as *RAMS parameters*, are defined as follows:

- reliability is the ability of a system to continue performing a designed function;
- availability is the ability of a system to be ready to perform a designed function when requested;
- maintainability is the ability of a system to be restored to a state in which the designed function can be performed again after modification and repair actions are performed; and
- safety is the absence of catastrophic events that endanger the integrity of the users and the environment of the system.

As illustrated in Figure 5.1, the RAMS parameters are interdependent. For instance, improved maintainability can also cause the availability to increase as a failed system can potentially be repaired faster and is therefore available sooner.

5.1.2 Threats of Dependability Attributes and Countermeasures

The RAMS parameters are threatened by *failures*, *errors*, and *faults*, which can be defined as follows [109, 171, 105, 6]:

- A failure is the inability of a system to perform the designed function. Failures are caused by errors, which are defined as discrepancies between the measured, computed, or received value and the intended value.

- Errors, in turn, are consequences of faults which are defects of the system.
- Faults can be grouped into various categories including, for instance,
  - hardware faults (e.g., cross talk, short circuit, defect memory section),
  - software faults (e.g., use incorrect data type, division by zero, wrong interpretation of values), and
  - manufacturing faults (e.g., incorrect mounting, damaged wire, miscalibrated sensor).

Note that a fault does not necessarily have to lead to an error. For instance, a defective section in a memory might never be used. Consequently, this fault has no impact.

5.1.3 Means to Maintain Dependability Attributes

Aviziens et al. [6] define the following four means to maintain dependability:

- *fault prevention*,
- *fault removal*,
- *fault forecasting*, and
- *fault tolerance*.

The goal of fault prevention is to avoid introducing faults in the first place. For instance, programming languages that support strong typing can prevent functions from returning wrong data types. Fault removal methods aim to eliminate faults during development and use. Both fault prevention and fault removal are categorized as *fault-avoidance measures*, i.e., methods that avoid the occurrence of faults. On the other hand, fault tolerance and fault forecasting are classified as means that accept the occurrence of faults and are, therefore, referred to as *fault-acceptance means*. The goal of fault forecasting is to predict future faults and their consequences. Fault-tolerance means aim to prevent failures in case faults occur.

As autonomous vehicles are complex systems consisting of various hardware and software components, numerous faults may occur. Therefore, we must assume that fault prevention and fault-removal means cannot eliminate all faults [182]. Faults that fault-avoidance measures have not addressed have to be handled by fault-acceptance means. Fault-forecasting measures can eliminate predictable faults. However, unpredictable faults that have not been eliminated by fault-avoidance means, e.g., undetected software bugs, are not handled by fault-forecasting measures. Only fault-tolerance strategies can handle such faults. Consequently, fault-tolerance measures have to be implemented in autonomous vehicles to avoid that faults that are neither handled by fault prevention, fault removal, nor fault forecasting measures do not cause failures.

5.1.4 Characteristics of Fault-Tolerant Strategies

A fault-tolerant strategy can improve the reliability, availability, and safety of a system:

- Reliability: A fault-tolerant system may handle occurring faults so that they do not cause failures. Hence, the system can continue performing the designed function.
- Availability: A fault-tolerant system may handle faults that occur before the operation of the system. Thus, the system is ready to perform a designed function when requested.
- Safety: A fault-tolerant system may handle occurring faults so that they do not cause faults that, in return, cause catastrophic events that endanger the integrity of the users and the environment of the system.

The foundation of a fault-tolerant system is *redundancy* [171], i.e., auxiliary components are introduced that perform the same or a similar function as the existing components [105]. Redundancy can be defined on different levels of the system, including, for example, on the software and hardware level [35].

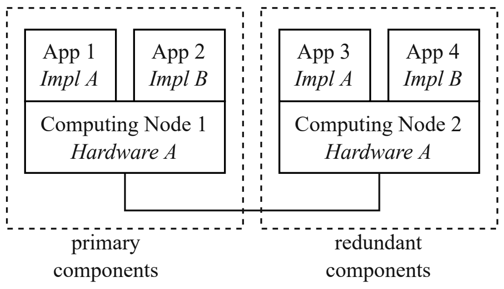
As illustrated in Figure 5.2a, to keep the development effort low, copies of the primary components, i.e., the minimum required components to execute the desired function, can be used as redundancies. However, such systems can not handle *common-mode failures*.

Common-mode failures are caused by a single fault which causes multiple components that implement the same function to fail at the same time [139]. For instance, a software bug in an application, e.g., a division by zero, causes the primary and the redundant application to fail simultaneously if the applications process the same input data and execute the same code lines concurrently. Consequently, a system's reliability, availability, and safety can be affected negatively.

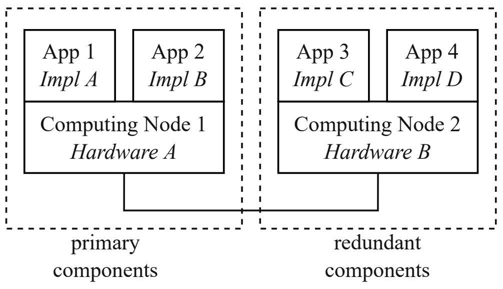
An approach to limit common-mode failures is *design diversity*, i.e., primary and redundant components are designed divisively [18]. Figure 5.2b shows a system that uses diverse hardware and software components. On the software level, diversity can be achieved, for instance, by employing two independent teams that implement the same software function in different programming languages. On the other hand, hardware diversity can be achieved by using hardware components from different manufacturers.

Another measure to counteract common-mode failures is the separation of redundant components [142]. For instance, primary software applications and the corresponding redundant software components shall be executed by distinct computing nodes. Through this separation, the probability of both applications failing simultaneously due to a failure of a computing node can be reduced. Note that both systems depicted in Figure 5.2 implement a separation of redundant components.

To further limit the probability of common-mode failures, redundant computing nodes can be located in a separate physical location. Therefore, the probability of a common-mode failure caused by, e.g., an accident or a fire within the vehicle, is reduced.



(a) A system which uses copies of the primary components to implement redundancy. Computing Node 1 and Computing Node 2 use on the same hardware. Furthermore, the implementations of App 1 and App 3 as well as App 2 and App 4 are identical.



(b) A system which applies design diversity. The redundant hardware and software components and the primary components are diverse. Computing Node 1 and Computing Node 2 use different hardware. Furthermore, App 1 and App 3, as well as, App 2 and App 4 implement the same function divisively.

Figure 5.2: Visualization of two different redundancy approaches, i.e., copying primary components and design diversity.

Besides the introduction of redundancies into a system, fault-tolerance strategies require further elements including, for instance, *detection*, *diagnosis*, *isolation*, and *reconfiguration* measures [6, 171]. Detection means are required to identify errors and hence initiate the subsequent steps of the fault-tolerance strategy. Diagnosis measures aim to identify the cause of an error. Isolation actions are performed to prevent the further propagation of an error. Finally, reconfiguration procedures ensure that a correct functioning component takes over the task of the faulty component.

5.2 Fault-Tolerant Approaches for Autonomous Systems in Different Domains

In several domains, besides the automotive sector, a pursuit towards autonomous systems can be observed. Many of these autonomous systems execute safety-critical tasks. Consequently, fault-tolerant approaches have to be implemented to maintain a safe and reliable operation.

In what follows, we examine fault-tolerant approaches of autonomous systems used in the aviation, aerospace, railway, and nuclear-power domain.

5.2.1 Aviation Domain

The main incentive for automation in aviation is the reduction of human errors to increase safety [48]. Further benefits of automation in aviation are the decrease of operational costs, workload reduction, and an improved job satisfaction. Therefore, a

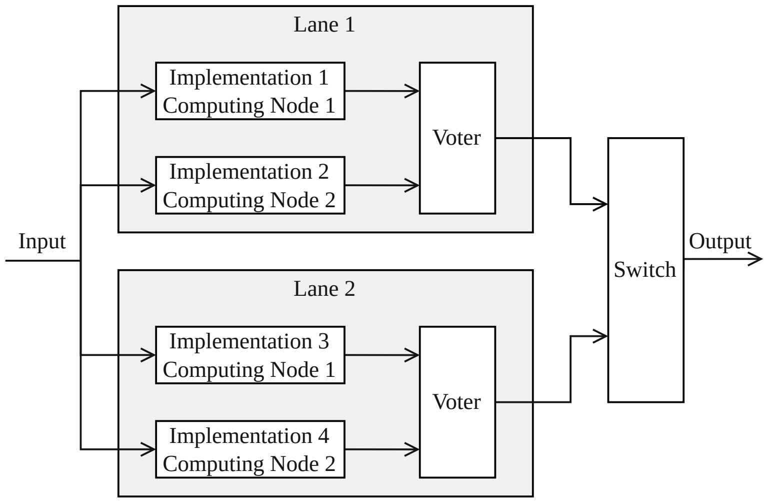


Figure 5.3: Duo-duplex architecture.

continuous development towards a higher level of automation in the aviation industry can be perceived. The implemented automation approaches have to fulfill a high level of safety and reliability as faults can lead to a large number of fatalities [172].

To improve the safety of airplanes, fault-tolerant systems based on *duo-duplex*, *triple-triple*, or *quadruplex architectures* are employed [66, 40]. A duo-duplex architecture, as illustrated in Figure 5.3, consists of two lanes, whereby each lane includes two identical computing nodes, which differ from the computing nodes of the other lane. Each of the four computing nodes executes a distinct software implementation. Although the implementations are not identical, they provide the same functionality. A voting mechanism executed by each lane checks whether the output of the two implementations is identical. In case diverging outputs are detected, the voter instructs the switch to use the output determined by the redundant lane. Consequently, duo-duplex systems, which are, for example, found in airplanes manufactured by Airbus [208], can tolerate one failure.

An architecture that can tolerate more failures is the triple-triple redundancy architecture, which is used, for example, by Boeing [224]. A triple-triple system, as shown in Figure 5.4, consists of three homogeneous lanes, whereby each lane includes three distinct computing nodes, which execute distinct software implementations. Furthermore, each lane implements a so-called *2oo3 voting logic* (“two out of three”), which compares the output of the three distinct implementations. Only if all three outputs diverge, the voter instructs the switch to use the output of another lane. Otherwise, the dominating output value is considered to be valid. Hence, triple-triple can tolerate two-lane failures as well as one implementation failure of the remaining lane.

Also, the quadruplex architecture can tolerate up to two lane failures [37]. Systems based on this architecture, as illustrated in Figure 5.5, implement four homogeneous

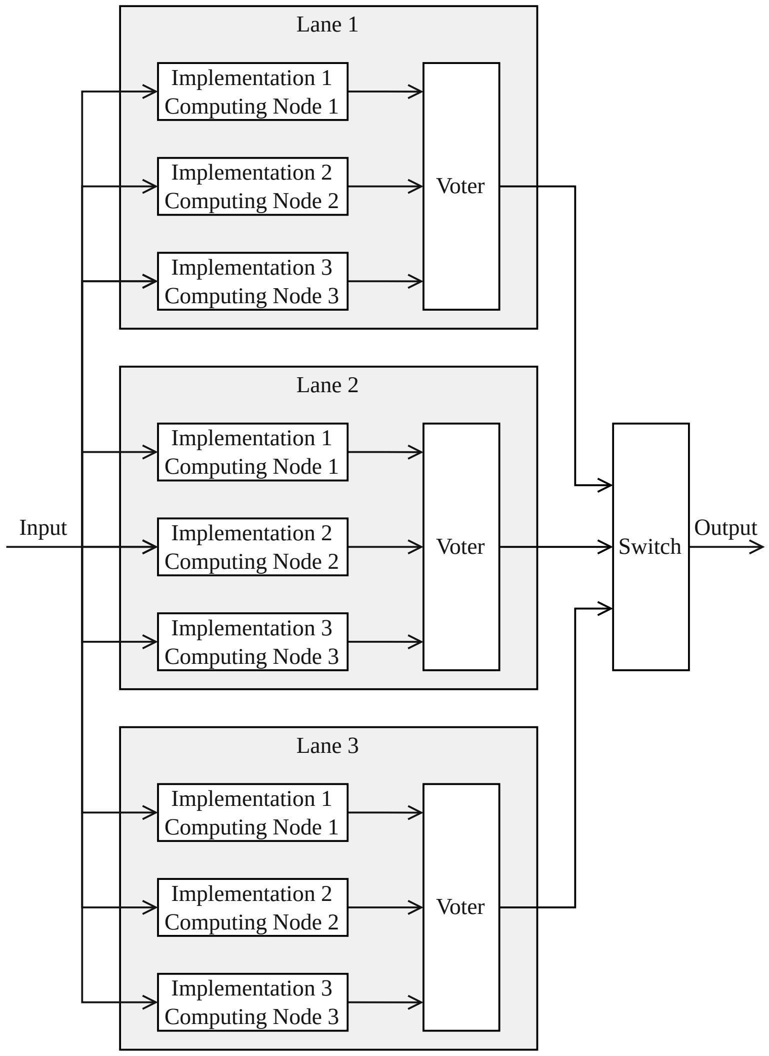

Lane 1
Implementation 1 Computing Node 1
Implementation 2 Computing Node 2
Implementation 3 Computing Node 3
Voter
Lane 2
Implementation 1 Computing Node 1
Implementation 2 Computing Node 2
Implementation 3 Computing Node 3
Voter
Lane 3
Implementation 1 Computing Node 1
Implementation 2 Computing Node 2
Implementation 3 Computing Node 3
Voter
Input
Switch
Output

Figure 5.4: Triple-triple architecture.

lanes, whereby each lane consists of a voter, a switch, as well as a computing node, which executes the software implementation.

The voters implement a *3oo4 voting logic* (“three out of four”), i.e., if three of the four lanes provide the same output, this output is considered correct. After the first fault, the lane which provided the faulty output will be isolated, and the voting logic degrades to 2oo3. Therefore, another fault can be tolerated.

5.2.2 Aerospace Domain

In general, space missions require high levels of automation. These high automation levels are, for instance, a consequence of the fact that space is inhabitable and that

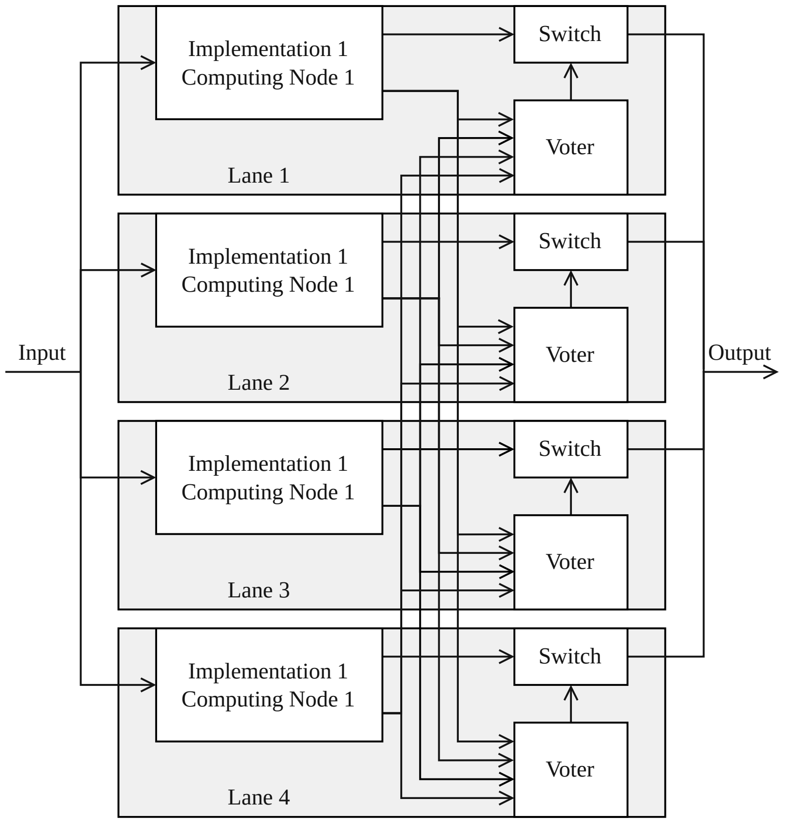


Figure 5.5: Quadruplex architecture.

physical contact with the spacecraft becomes almost impossible after the launch. For example, the Rosetta asteroid lander Philae [38] traveled ten years in space to reach its destination and to perform a complex, pre-programmed landing maneuver, followed by another two years of operation. However, also in manned space missions, the demand for autonomous systems is increasing, ranging from general tasks such as controlling the water quality [67] and managing errors to robots performing dedicated tasks in scientific experiments [150] as well as performing repair and maintenance tasks [32].

In order to ensure the safety and reliability of space missions, several standards and guidelines have been published by the European Cooperation for Space Standardization (ECSS). For instance, the ECSS-Q-ST-40C Rev.1 standard [61] defines fault tolerance to be the basic safety requirement to control hazards. Therefore, the system design shall include an on-board redundancy management for safety-critical functions and mechanisms for fault detection, fault isolation, and switching to redundant components, which are also referred to as FDIR approaches. Furthermore, it is required to provide the ground station with necessary information concerning fault detection, isolation, tolerance, and the current redundancy status.

The computational complexity of the applied FDIR systems is, in general, rather limited due to the limited computational power of the spacecraft [176]. Therefore, advanced methods such as AI-supported FDIR methods are hardly applicable, especially in smaller, low-power satellites. However, research regarding AI-supported FDIR approaches has increased in the last few years. For instance, Jaekel and Scholz [111] present an AI-supported FDIR architecture for future spacecrafts.

As far as satellites are concerned, according to Olive [176], two main strategies for model-based FDIR methods are applied: the *half-satellite FDIR strategy* and the *hierarchical FDIR strategy*.

The half-satellite FDIR strategy defines that the satellite is fully reconfigured after the fault detection, and all components are switched to redundant ones without performing any fault isolation. Afterward, the ground station has to identify the cause of the anomaly and correct it. On the other hand, the hierarchical FDIR strategy specifies that the fault is recovered on the lowest possible layer after the fault is detected and isolated. In order to permit a graduated reaction, four levels of faults are distinguished:

- Level 0 includes faults that have no impact on the performance of the satellite, e.g., a single bit-flip. The detection and automated recovery are performed locally in the affected component.

- Level 1 concerns faults requiring switching to a redundant component, leading to a temporary degraded mode that does not affect the mission goal. In this case, the detection must be performed outside the affected component, and the concerned subsystem performs the recovery.

- Levels 2 and 3 share the same kind of detection and recovery scheme, while Level 2 considers faults that previous levels could not have covered, Level 3 faults derive from the FDIR components. Faults of Level 2 and 3 are recovered by switching to redundant modules.

- Level 4 is the most critical level, which involves multiple faults on Level 2 and 3 or hardware alarms. Such faults are a consequence of a critical breakdown. The recovery of Level 4 faults is performed by the ground station and requires an interruption of the mission.

As redundancy is crucial for a successful FDIR process, choosing the right amount of redundancy is essential. Suich and Patterson [202] discuss considerations between financial costs and reliability benefits regarding redundancy. Although redundancy is indispensable, it can also decrease the safety of the spacecraft by, for example, adding additional faults or masking design flaws [175].

5.2.3 Railway Domain

In the railway industry, the unavailability of a train or rail track can lead to the downtime of various trains due to their interdependence. Therefore, a single failure can affect a large number of people. As a consequence, a significant challenge of automation in the railway industry is the increased demand for availability while guaranteeing safety [17]. Consequently, new fault-tolerance approaches are required.

For example, Wang et al. [215] present a concept for an efficient and safe train-control system, i.e., a system that controls, e.g., the track intervals, the train routes, and the train speed. The proposed train control system is a distributed system consisting of subsystems that are located in trains and on the wayside, whereby the safety of the system is enhanced by applying parallel monitoring. For instance, the system that determines the maximum permitted speed of a train is implemented by a unit located on-board the train as well as by the wayside system. The train compares the two independent results, and if they are equal, the determined maximum speed is further processed. Due to the heterogeneously computed values, common-mode faults are excluded.

Furthermore, Lopez et al. [153] propose an approach to increase the reliability of a railway signaling system, i.e., a system that transmits commands and information between trains and the control center. Their approach makes use of the migration of railway signaling systems from a circuit-switched to a packet-switched network. The authors suggest implementing an approach that combines spatial and temporal redundancy in order to increase reliability. Note that old approaches, which are based on circuit-switched networks, employ spatial redundancy by transmitting information through different network paths. Temporal redundancy, which the packet-switched network enables, is realized by delivering the same information multiple times with an offset in time. Sending the packets repeatedly with a time delay can improve the resilience against burst errors, i.e., a sequence of incorrect bits transmitted via a communication channel.

5.2.4 Nuclear-Power Domain

Operating nuclear power plants efficiently is a complex task [23]. Thus, automation approaches are applied to control and run several parts of the systems. For instance, the startup of the reactors [28] and the in-core fuel management [188] are tasks that can be optimized using automation.

Besides increasing the efficiency of nuclear power plants, another major challenge is to ensure a safe operation. Accidents in the past, such as those at the Three Mile Island (1979), Chernobyl (1986), or the Fukushima Daiichi (2011) plants, have demonstrated the enormous environmental consequences of radiation leakage, which contaminate affected areas for decades and leaving them uninhabitable [31, 30].

In general, for systems that are essential for the safety of nuclear-power plants, redundant backup systems are installed [106]. These systems take over the operation in case the primary system fails. For instance, as a consequence of the Fukushima disaster, all power plants in the United States were equipped, e.g., with additional emergency pumps, generators, and battery banks [51].

However, backup systems can also have unexpected and unwanted side effects. For instance, Kettunen, Reiman, and Wahlström [128] argue that the introduction of redundant systems increases the complexity of systems, which in turn can cause that faults might not get recognized and thus accumulate over time. Furthermore, common-cause failures are more likely to occur.

To handle faults in safety-critical systems as well as in their backup systems, fault detection and diagnosis (FDD) approaches can be employed [159]. FDD methods aim to detect, isolate, and identify the size and time-variant behavior of faults [110]. Application areas of FDD methods in nuclear-power plants include instrument calibration monitoring [94], reactor core monitoring [168], and monitoring of loose parts that can damage the reactor coolant system [65]. Instrument calibration monitoring, for instance, can be implemented by using a redundant set of sensors [88]. The average of the measurements of the redundant sensors is used to determine whether the sensor under investigation shows an abnormal derivation. Instead of the data perceived by redundant sensors, correlating measurements in the system can be used to predict the output of a sensor. The advantage of such an analytical approach is that hardware can be reduced.

By introducing FDD methods, the safety of nuclear-power plants can be improved as, for instance, the equipment reliability is enhanced, and the exposure of personal to radiation can be reduced [159]. Furthermore, also the efficiency is increased since, for example, the availability and the lifetime can be optimized.

5.3 A Stepwise Fault-Tolerance Approach

As mentioned before, many of the applications executed by autonomous vehicles are safety-critical, i.e., a fault might result in a hazardous situation. Therefore, to guarantee the safety of the passengers and other road users in case an occurring fault causes a safety-critical application to misbehave, measures have to be implemented to ensure a safe operation in such situations.

Although handling faults by performing an emergency stop is feasible, such behavior is not always desirable since this will cause customer satisfaction to decline [133]. Furthermore, in case the vehicle is, for example, moving at a very high speed or driving in a tunnel, it might not be safe to stop abruptly.

On the other hand, increasing the fault-acceptance rate might cause vehicles to operate unintendedly, leading to a loss of customer confidence.

In order to address the above issues, we proposed in previous work FDIRO [116, 122] (standing for "Fault Detection, Isolation, Recovery, and Optimization"), a fault-tolerance approach for handling faults in a stepwise fashion, extended with a configuration-optimization procedure, and based on the FDIR approach (cf. Subsection 5.2.2). In

Aptus, we employ Fdiro in the component layer to implement the required self-healing property.

5.3.1 Overview

The idea of Fdiro is to handle faults in a stepwise manner by executing the following four procedures:

1. detection of the fault,
2. isolation of the fault by a switchover between redundant instances,
3. recovery of the software-architecture requirements, and
4. optimizing the configuration.

The first three steps correspond to the *detection*, *isolation*, and *recovery actions* as specified by Fdir. The execution of those steps eliminates the fault and ensures that the vehicle can continue its operation. However, since the resulting configuration might not be optimal, we extended the Fdir strategy by an additional optimization step, hence the name "Fdiro". The newly introduced optimization step aims to enhance the system according to goals that depend on the current situation.

The motivation for implementing a stepwise reconfiguration is that, for some faults, the reconfiguration time is critical. In particular, certain cases require that the time until a fault is isolated and a redundant instance takes over the actions of the faulty instance must lie within a range of milliseconds. For instance, the maximum acceptable reconfiguration time of the application that controls the steering is, in some cases, 90 ms, according to Orlov et al. [177].

The stepwise design of our reconfiguration approach meets this requirement since the complexity of the detection and isolation step is low, enabling that those steps can be performed within a guaranteed time.

Note that, to provide a fast fault detection, the monitoring period has to be short, and the detection actions during a monitoring cycle have to be of low complexity.

Fast fault isolation by switching over to a redundant application is ensured due to the implementation of different operation modes. The idea is that each safety-critical function is executed redundantly, whereby one application is executing in active or active-low operation mode. The other redundant applications are executing in active-hot operation mode. The active-hot applications are capable of taking over the responsibilities of the active and active-low applications within milliseconds if required.

In contrast to the detection and isolation steps, the complexity of the redundancy recovery and optimization steps is significantly higher. However, since the time criticality of the latter steps is lower than that of the former steps, such a behavior is acceptable. Figure 5.6 illustrates this complementary effect.

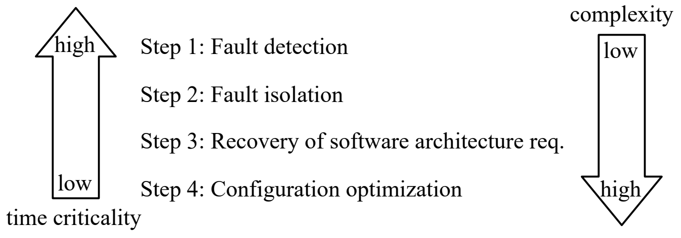


Figure 5.6: Comparison of time criticality and computational complexity of the FDIRO steps.

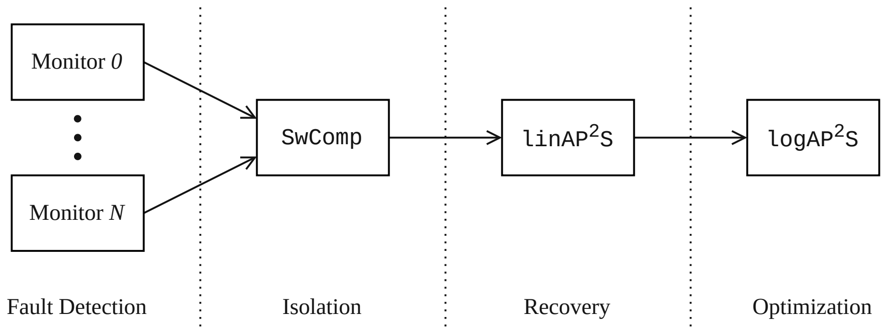


Figure 5.7: Workflow of FDIRO.

5.3.2 Workflow

The individual steps defined by FDIRO are, as illustrated in Figure 5.7, executed by different components.

The components that detect faults and consequently trigger a reconfiguration are called *monitors*. Detecting faults in autonomous vehicles is a well-known challenge [155]. Several approaches have been published to detect faults in various software and hardware components, including, for instance, perception applications [10], localization applications [113], computing nodes [14], and sensors [82]. The implementation of the monitors in an autonomous vehicle strongly depends on the applied applications and hardware. As APTUS focuses on the generic aspects of extending the system architecture to become self-managing, we do not further discuss monitoring approaches.

In case any monitor detects a fault, it reports the fault to the so-called *switchover component*, `SwComp`. In the first step, `SwComp` determines all applications that are affected by the fault. For instance, if a monitor reports that a computing node failed, all applications executed by that computing node are considered to be affected. On the other hand, if a monitor, which monitors a specific application, reports a fault, only the failed application is considered affected.

For each affected application, `SwComp` executes in the next step the procedure illustrated in Figure 5.8. First, `SwComp` isolates the affected application. To perform the isolation

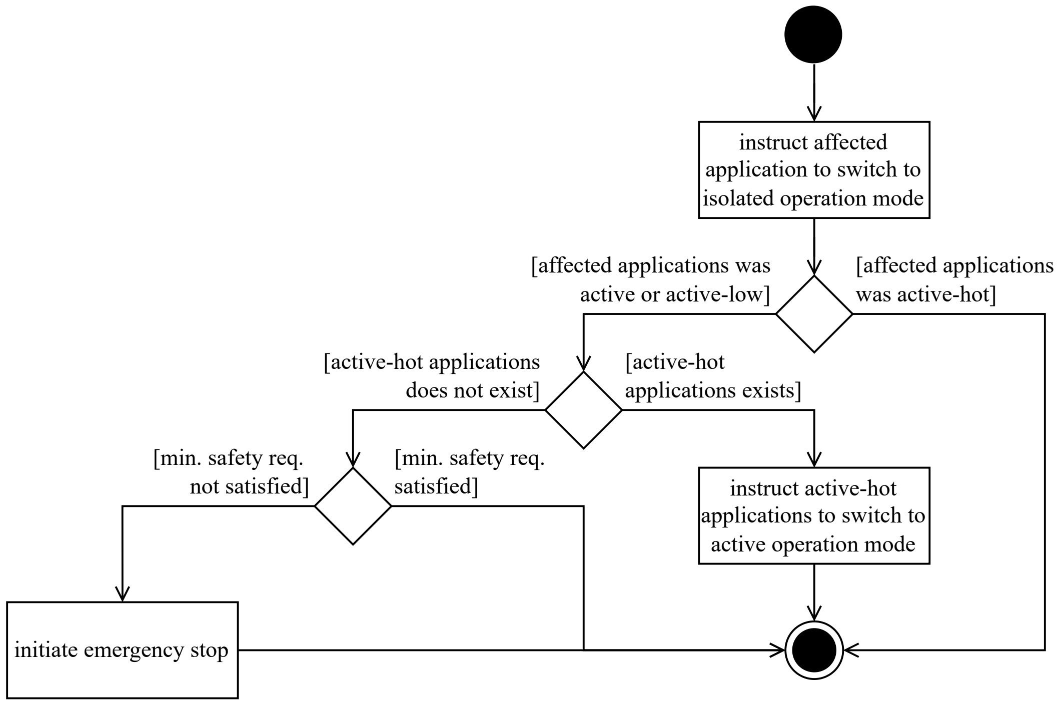

instruct affected application to switch to isolated operation mode
[affected applications was active or active-low]
[affected applications was active-hot]
[active-hot applications does not exist]
[active-hot applications exists]
[min. safety req. not satisfied]
[min. safety req. satisfied]
instruct active-hot applications to switch to active operation mode
initiate emergency stop

Figure 5.8: Activity diagram of the procedure executed by `SwComp` to isolate and restore the functionality of an application affected by a fault.

procedure, we introduce an additional operation mode called *isolated*. If the operation mode of an application is set to isolated, the output of the application is no longer sent to other applications or actuators in order to avoid propagating the fault. Furthermore, isolated applications are terminated by the host platform.

The further actions executed by `SwComp` depend on the previous operation mode of the affected application, i.e., the operation mode before the application was isolated. If the application was in active-hot operation mode, then no further actions are required by `SwComp` and the control is handed over to `AP`^{2}`Initiator`, which is part of the reconfiguration layer and responsible for executing the redundancy-recovery procedure.

On the other hand, if the application was in the active or active-low operation mode, then `SwComp` analyzes in the next step whether active-hot applications exist which implement the same function as the affected application. In case the system executes such applications, `SwComp` selects an active-hot application and instructs it to switch to the active operation mode. Through this operation mode upgrade, the prior active-hot application takes over the tasks of the application affected by the fault.

If multiple suitable active-hot applications exist, `SwComp` deterministically selects the one with the lowest redundancy-instance number. A deterministic selection procedure is essential as otherwise, the configuration graph becomes non-deterministic, i.e., the same input configuration and a configuration transition might lead to different successive configurations. Note that active-hot applications also affected by the currently considered fault can not be selected for becoming the new active application.

In case no suitable active-hot instance exists, `SwComp` has to evaluate whether the failure of the instance causes the safety level to drop below an acceptable threshold. `SwComp` uses the function Λ_{prio}, which we introduced in Subsection 4.6.1, to determine whether the minimum safety requirements are fulfilled. Recall that Λ_{prio} defines five levels, whereby configurations that belong to level 0 do not fulfill the minimal safety and reliability requirements.

If the current system state does not fulfill the minimum safety requirement, then `SwComp` initiates an emergency stop. However, in case the current system state satisfies the minimum safety requirement, the redundancy-recovery step of FDIRO is executed in the next step.

Once the procedure illustrated in Figure 5.8 has been completed for all applications affected by the fault, `SwComp` instructs the reconfiguration layer to proceed with the execution of the redundancy-recovery procedure of FDIRO. In particular, `SwComp` sends a recovery request to AP^2`Initiator`. This request includes the initiator event, which specifies the fault that triggered the FDIRO procedure. Since the initiator event defines a fault, AP^2`Initiator` instructs `linAP`^{2}`S` to compute a new configuration unless the required configuration is part of the configuration graph.

As discussed in Section 4.3, `linAP`^{2}`S` aims to quickly recover compliance with the current software-architecture requirements. Since the present fault might cause that the current software-architecture requirements cannot be fulfilled, `linAP`^{2}`S` computes in parallel configurations based on weakened software-architecture requirements. In case no configuration can be found, an emergency stop is initiated. Otherwise, the new configuration is applied, and the final step of FDIRO, i.e., the configuration optimization step, is executed.

Like the reconfiguration step, the optimization procedure of FDIRO is also executed by the reconfiguration layer. The optimization step is triggered by sending a request to AP^2`Initiator` which specifies as initiator event the request to optimize the current configuration. Consequently, AP^2`Initiator` instructs `logAP`^{2}`S` to compute an optimized configuration unless the configuration graph already contains the required configuration. Once the optimized configuration is applied, the FDIRO procedure is successfully completed.

5.4 A Hardware Redundancy-Recovery Extension

FDIRO, as described in the previous section, implements a redundancy-recovery approach that can recover the redundancy of software applications by starting new application instances. However, FDIRO is not capable of recovering failed hardware components like computing nodes or sensors. Consequently, only a limited number of hardware faults can be tolerated before the mission has to be terminated by an emergency stop. Therefore, we introduce the hardware redundancy-recovery approach Hrr, which is an extension to Fdiro.

The idea of Hrr is that in case of a hardware fault, Fdiro instructs Hrr to recover the faulty component, after performing the isolation step. Hrr then analyzes the fault and determines a so-called *recovery plan*, which is an ordered list of recovery approaches. In case of a successful reactivation, the recovered hardware is again integrated into the system.

In what follows, we describe the Hrr procedure. Furthermore, we provide a use case to illustrate the introduced procedure.

5.4.1 Workflow of the Hardware Redundancy-Recovery

The procedure of the hardware redundancy-recovery approach is executed by the so-called *Hrr component*. The Hrr procedure is executed in parallel to the Fdiro procedure. As illustrated in Figure 5.9, the Hrr component first analyzes the *error report*, i.e., a report created by the monitor which classified the hardware component under investigation as malfunctioning.

Based on that error report, the Hrr component determines a so-called *fault category*. The latter contains a pre-sorted list of several faults that might have caused the component under investigation to fail. The faults listed in a fault category can be, for instance, ranked using an FMEA-like ("failure mode and effects analysis") approach [206]. The idea is that, for each fault f, its severity, $S(f)$, probability of occurrence, $O(f)$, and probability of detection, $D(f)$, is determined at design time. The values of those parameters are bounded between 1 and 10, whereby 1 is considered insignificant/unlikely, and 10 corresponds to very significant/probable. By multiplying those three parameters, the so-called *risk-priority number*, $RPN(f)$, is calculated:

$$RPN(f) = S(f) \cdot O(f) \cdot D(f).$$

The risk priority number of the individual faults is used to rank them in a descending fashion. Each fault entry of a fault category holds a predefined list of so-called *recovery approaches*, i.e., approaches to fix the fault that caused the hardware component under investigation to fail. The recovery approaches are determined at design time and are ordered hierarchically concerning, for example, their effectiveness, estimated computational effort, and execution time. Furthermore, for each recovery approach, a list of recovery approaches that can be carried out in parallel is specified.

According to the risk priority number of the corresponding fault, the recovery approaches of all faults that are part of the identified fault category are sequentially added to the so-called *recovery plan*. In case the Hrr component cannot identify a fault category based on the provided error report, the recovery approaches of all fault categories of the component under investigation are added to the recovery plan.

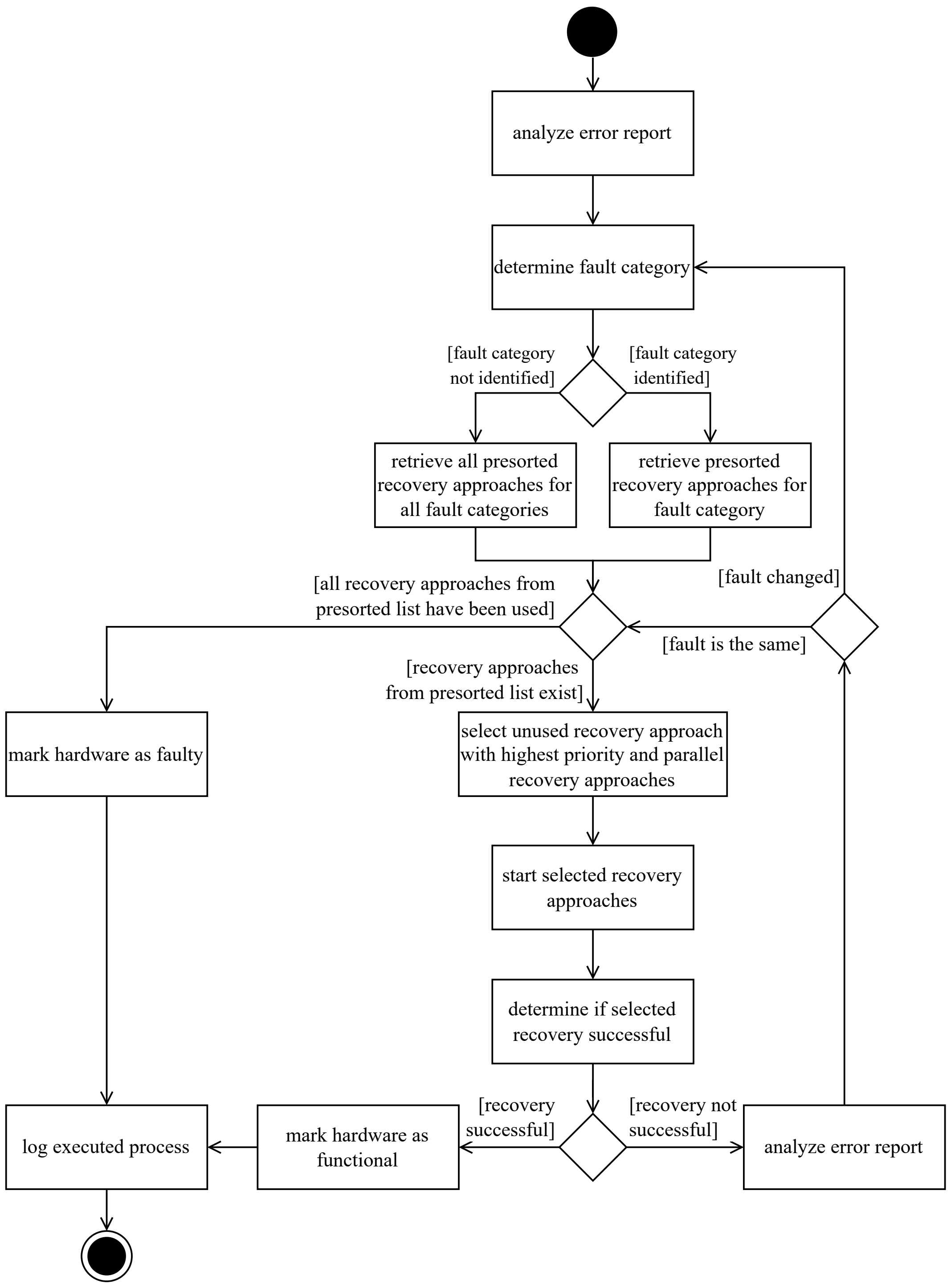

analyze error report
determine fault category
[fault category not identified]
[fault category identified]
retrieve all presorted recovery approaches for all fault categories
retrieve presorted recovery approaches for fault category
[all recovery approaches from presorted list have been used]
[fault changed]
[fault is the same]
[recovery approaches from presorted list exist]
mark hardware as faulty
select unused recovery approach with highest priority and parallel recovery approaches
start selected recovery approaches
determine if selected recovery successful
[recovery successful]
[recovery not successful]
log executed process
mark hardware as functional
analyze error report

Figure 5.9: The activity diagram of the Hrr component.

As the fault categories, the faults, and the corresponding recovery approaches are determined at design time, they are considered as being static. However, using, for instance, an OTA ("over the air") update mechanism, the set of fault categories, faults, and recovery approaches can be adjusted.

After the list of recovery approaches that might fix the fault is identified, the processing of those approaches starts. The Hrr component first selects the approach which is ranked the highest. Additionally, all recovery approaches which can be executed in parallel to that approach are selected as well. Next, the selected recovery approaches are executed.

Once all selected recovery approaches are terminated, the Hrr component determines whether the recovery was successful. A recovery is considered successful if the monitor of the component under investigation does not detect any error. In that case, the hardware component under investigation is marked as *functional* and can therefore be again used by the system.

Before the Hrr component terminates, the executed procedure, including, for instance, the determined fault category, the executed recovery approaches as well as their results, are logged. The logs of several vehicles can then be used to optimize, for example, the prioritization of recovery approaches.

In case the executed recovery approaches did not fix the fault, the Hrr component determines whether the error report of the monitor still indicates the same fault. If so, the next recovery approach in the recovery plan that has not been conducted yet is selected. In case no recovery approach that has not been executed yet exists, the hardware component under investigation remains marked as *faulty* and can, therefore, not be used by the system. After logging the executed procedure, the Hrr component terminates.

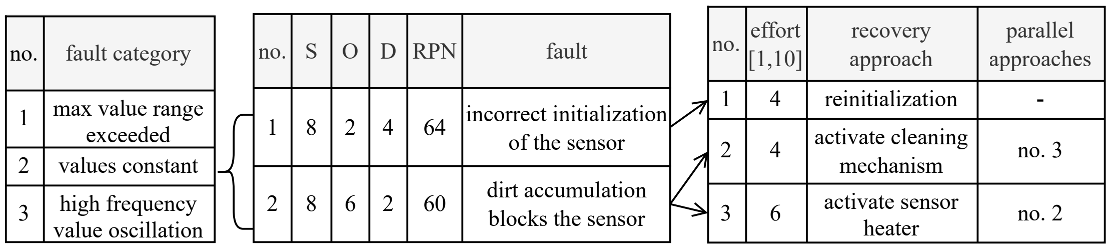
However, if the analysis of the error report yields a fault that is different from the one determined at the beginning of the Hrr procedure, the procedure has to start over.

5.4.2 Use Case: Radar Fault

To illustrate the workflow of Hrr, we assume that an application failed due to a defective front-right radar. Consequently, Fdiro isolates this application and instructs the Hrr component to start its recovery procedure for the defective radar.

A subset of the fault categories for radar sensors is illustrated in Figure 5.10. In this use case, we assume that according to the error report provided by the monitor, the fault category "value constant" is the most fitting, i.e., the data points of the radar sensor are constant, e.g., in terms of the measured distance or relative velocity.

As illustrated in Figure 5.10, two faults are defined for the considered fault categories, namely "incorrect initialization of the sensor" and "dirt accumulation blocks the sensor", which are ordered according to their risk-priority number. Furthermore, for both faults, recovery approaches are specified. In order to attempt to recover the faulty radar sensor,



no.	fault category
1	max value range exceeded
2	values constant
3	high frequency value oscillation

no.	S	O	D	RPN	fault
1	8	2	4	64	incorrect initialization of the sensor
2	8	6	2	60	dirt accumulation blocks the sensor

no.	effort [1,10]	recovery approach	parallel approaches
1	4	reinitialization	-
2	4	activate cleaning mechanism	no. 3
3	6	activate sensor heater	no. 2

Figure 5.10: The fault categories, faults, and recovery approaches of a radar sensor.

the Hrr component first initiates a reinitialization of the sensor, as this recovery approach is defined as a countermeasure for the fault having the highest risk-priority number. After the reinitialization of the front-right radar sensor terminates, a monitor determines whether the sensor is still faulty. In this use case, we assume that the reinitialization of the sensor does not fix the fault, and the fault does not change during the recovery procedure. Consequently, the Hrr component executes the remaining recovery approaches.

The next approach conducted by the Hrr component in order to fix the fault is to activate the cleaning mechanism of the sensor. While executing this recovery approach, the sensor heating is activated as well since those two approaches can be performed in parallel. After the termination of both recovery approaches, the monitor again checks whether the fault was fixed. Assuming this is the case, the Hrr component logs the fault as well as the executed recovery approaches and activates the sensor again.

5.5 The Tool **`AT-CARS`**

To determine the reliability of an autonomous vehicle over time, we developed the tool `AT-CARS`, standing for "analyzing tool for complex autonomous and reliable systems".

The basic idea of `AT-CARS`, as illustrated in Figure 5.11, is that a user provides the following items:

- the definition of the computing nodes, sensors, functions, and applications the vehicle is equipped with,
- the failure probability distribution of the computing nodes, sensors, and applications,
- the recovery probability after an occurring failure of the computing nodes, sensors, and applications,
- the initial configuration, and
- a set of simulation parameters.

`AT-CARS` determines, based on these input parameters, the reliability of the system over time using a Monte Carlo simulation [166], which is a probabilistic technique that is used

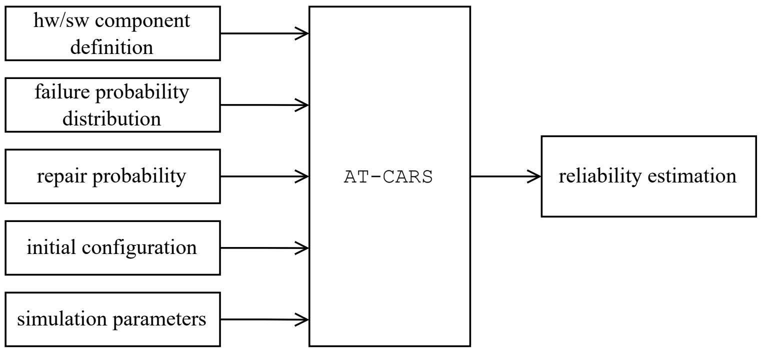


Figure 5.11: System overview of AT-CARS.

in various fields, including finance [165], manufacturing [156], and medicine [9].

The basic idea of Monte Carlo simulation is to model, e.g., a process or a system that involves random variables by generating random samples from the probability distributions associated with those random variables. After running multiple iterations of sampling the random variables, the results can be analyzed. The more iterations are executed, the closer the results converge to the true behavior of the process or system.

To determine the system's reliability, AT-CARS, which is implemented in Matlab[1], simulates the system behavior over a specified time by injecting faults, which are then handled by a fault-tolerance method. The fault-tolerance method that shall be applied is defined in the simulation parameters, whereby the user can select between three methods, denoted by m_1, m_2, and m_3. If the user selects fault-tolerance method m_1, then FDIRO, together with HRR, is applied. On the other hand, in case m_2 is chosen, FDIRO without HRR is applied. If the fault-tolerance method m_3 is selected, then AT-CARS only executes the fault detection and isolation step of FDIRO, i.e., the redundancy levels of failed applications are not recovered.

Since AT-CARS is based on a Monte Carlo simulation, several iterations are required in order to determine a sound reliability estimation of the system. The user defines the number of iterations that shall be executed in the simulation parameters.

Apart from employing a Monte Carlo simulation, various other approaches to determine the reliability of a system exist. We discuss some of these approaches and analyze whether they are suitable for analyzing autonomous systems in the next section. Afterwards, in Section 5.5.2, we describe the workflow of AT-CARS.

5.5.1 Modeling Approaches for Autonomous Systems

Classical reliability analysis approaches, such as *reliability block diagrams* (RBD) [108, 15] and *fault tree analysis* (FTA) [107, 148], can model dependencies between components

[1]MATLAB version: 9.13.0.2126072 (R2022b) Update 3.

in case simple system structures such as parallel or serial-like structures are considered. However, those approaches do not consider recovery behavior or dynamic behavior, e.g., the introduction of new redundant components. Thus, they are not capable of analyzing autonomous systems that consider such behaviors. Nevertheless, there are various approaches to enhance RBD and FTA.

An approach based on RBD is, e.g., *dynamic reliability block diagrams* (DRBD) [57]. The idea of DRBD is to introduce additional blocks that interact with each other, which allows modeling, e.g., common-cause failures and dynamically changing operation modes. However, due to the increased complexity of the block diagrams, other approaches than Boolean algebra are required to solve DRBD [184]. On the other hand, *dynamic fault tree analysis* (DFTA) [58, 162] is an approach based on FTA, which introduces additional gates that can represent a more dynamic system behavior.

Both above mentioned dynamic approaches, i.e., DRBD and DFTA, are not suited for modeling systems containing different operation and failure modes, as well as complex dynamic behavior. The problem is that those enhancements lead to the introduction of various additional parameters, which cause the modeling and analysis procedure to become tedious or even impossible.

Another frequently applied approach for analyzing complex systems is *Markov analysis* [15]. This analysis uses states and state transitions to represent different system behaviors. Therefore, recovery behaviors and dynamic behaviors can be modeled. However, the drawback of Markov analysis is that the *Markov property*, which states that the probability of a transition between two states only depends on the current state and the current point in time, has to be satisfied.

An enhancement of Markov analysis is the *semi-Markov process* [83]. This process allows the consideration of recovery behavior and is not limited to a memoryless failure behavior [15]. However, Markov models of complex systems can contain thousands of states, which leads to a modeling state space of enormous size. This state explosion causes the modeling and calculations to become tedious due to the complex computations involved in the semi-Markov process. As a result, this Markov variant is not convenient for analyzing large and complex systems.

Consequently, Markov models, which are designed to minimize the state space to avoid a state space explosion, have been introduced in the past. For instance, Heinrich et al. [115] introduced a tailored Markov model that reduces the size of the state space by combining different system states into one single state. However, this modeling approach violates the Markov condition. Consequently, differential state equations cannot be applied to analyzing the modeled systems. Therefore, Heinrich et al. implement, similar to `AT-CARS`, a Monte Carlo simulation to analyze the tailored Markov models. Like `AT-CARS`, their approach allows analyzing fault-tolerance methods. However, in contrast to `AT-CARS`, their approach does not support modeling the dependency of computing nodes and applications, i.e., a failure of a computing node does not cause the applications executed by this computing node to fail.

	Probability Density Function
Weibull distribution	$wei(t;\lambda,k) = \begin{cases} \frac{k}{\lambda}\left(\frac{t}{\lambda}\right)^{k-1} e^{-(t/\lambda)^k}, & t \geq 0 \\ 0, & \text{otherwise} \end{cases}$
Exponential distribution	$exp(t;\lambda) = \begin{cases} \lambda e^{-\lambda t}, & t \geq 0 \\ 0, & \text{otherwise} \end{cases}$
Normal distribution	$norm(t;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(t-\mu)^2}{2\sigma^2}}$

Table 5.1: Probability distribution functions supported by AT-CARS to define the fault probability of components. Note that the variable t denotes the time in hours. The other parameters need to be defined in the file J_1.

5.5.2 Workflow of AT-CARS

In the first step, as illustrated in Figure 5.12, AT-CARS parses the input parameters, which are specified in three JSON files [53] that are referred to as J_1, J_2, and J_3:

- The JSON file J_1 defines the properties of the computing nodes, the sensors, the functions, and the applications. Note that in the context of AT-CARS, we denote computing nodes, sensors, and applications as *components*.
- The JSON file J_2 specifies the initial configuration parameters.
- The JSON file J_3 defines the simulation parameters.

AT-CARS employs JSON as input format since JSON files are convenient to read and parse. Note that we illustrate the structure of the JSON files in Section 6.4.

Once the input is parsed, AT-CARS initiates the first iteration of the simulation by applying the initial configuration. This configuration specifies which applications are executed by which computing nodes at the beginning of the simulation time.

Next, for each component c that is part of the initial configuration, a failure time $t_{c,failure}$ is determined. The failure time of a component is a concrete value drawn from the continuous random variable $T_{c,failure}$ that follows a probability density function which is specified by the user in JSON file J_1. AT-CARS supports Weibull distributions [217], exponential distributions, and normal distributions to define the fault probability of components. These are commonly used failure probability distributions for components [167]. Table 5.1 lists the supported functions and their parameters. Note that AT-CARS is easily extensible to support further failure distributions.

Once the failure times are calculated, the failure that occurs chronologically first is determined. If this failure occurs after the *maximum simulation time*, which is specified in the JSON file J_3, then the iteration is completed. In this case, the system is considered functional during the entire simulation time. AT-CARS then proceeds by starting the

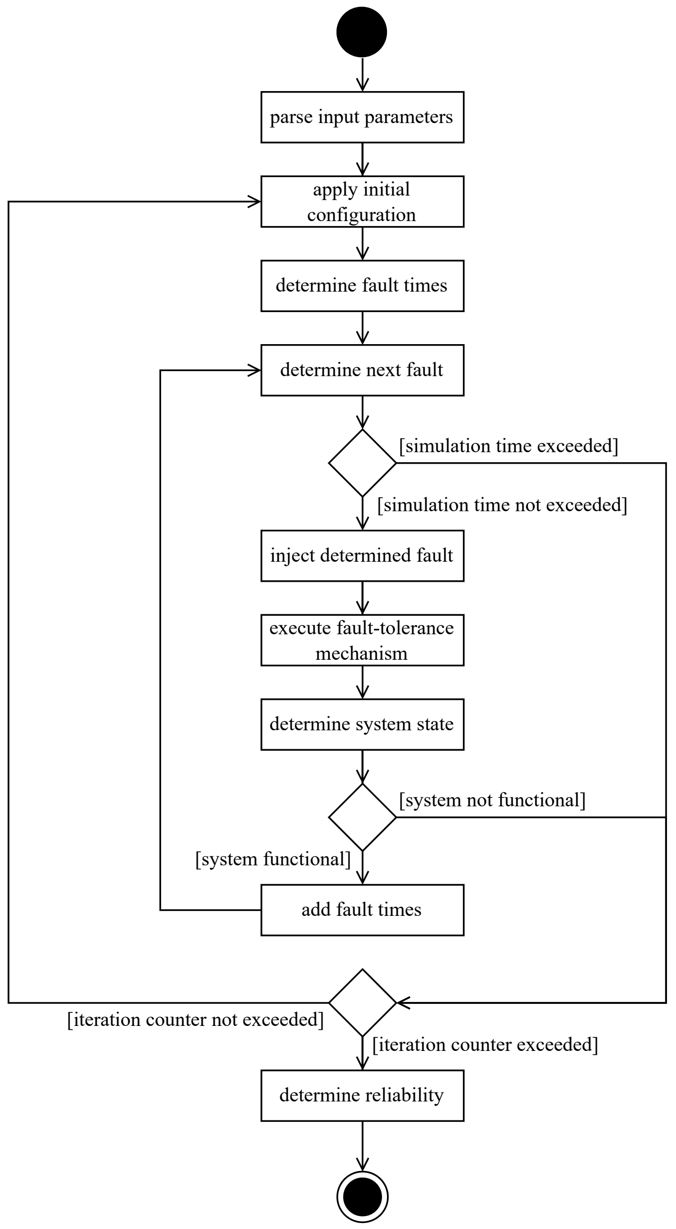


Figure 5.12: Workflow of AT-CARS.

next iteration unless the *total number of iterations*, which is specified in JSON file J_3, is already reached.

If the determined failure occurs within the specified maximum simulation time, then it is injected into the current configuration. AT-CARS indicates that a component failed, by setting the so-called *failed flag* for this component.

To handle this failure, AT-CARS executes in the next step the fault-tolerance mechanism, i.e., m_1, m_2, or m_3, which is specified in the JSON file J_3.

In case m_1 or m_2 is selected, AT-CARS performs the fault detection, fault isolation, redundancy recovery, and the configuration optimization procedure of Fdiro. If the fault-tolerance mechanism m_3 is selected, only the fault detection and the fault isolation step of Fdiro are executed. Notably, the implementation of the fault detection procedure is relatively lightweight since AT-CARS sets the failed flag for failed components, which can be accessed by Fdiro. The successive steps, i.e., the fault-isolation step, the redundancy-recovery step, and the configuration-optimization step, are performed as described in Section 5.3.

Additionally, if the failed component is a computing node or a sensor and m_1 is selected, Fdiro instructs Hrr after the execution of the isolation procedure to recover the failed component. AT-CARS does not simulate the individual steps of Hrr but instead uses the specified recovery probability of that component to determine whether the component can be recovered after the occurrence of a fault.

Furthermore, aside from defining recovery probabilities for computing nodes and sensors, AT-CARS also allows users to specify recovery probabilities for applications. Note that within the redundancy-recovery procedure of Fdiro, a new instance of the failed application is introduced. Nevertheless, there exists the possibility that this new instance immediately encounters the same fault that also caused the application instance it is replacing to fail. Therefore, AT-CARS provides users with the capability to define recovery probabilities for applications.

The recovery probability of a component c after the occurrence of a failure at time $t_{c,failure}$ is given by the conditional probability

$$P(R_c = \text{recovery successful} \mid T_{c,failure} = t_{c,failure}),$$

where R_c is a discrete random variable which can take on the values "recovery successful" and "recovery not successful". In what follows, we denote this conditional probability by $P_{c,recovery}(t_{c,failure})$. The user can specify the recovery probability in the file J_1.

After the fault-tolerance procedure finished its execution, AT-CARS determines the so-called *system state* of the vehicle. The system state indicates whether the system is functional. AT-CARS considers a system functional if all functions of priority *HIGH* are executed and certain sensors have not failed. The specific sensors that need to be functional are defined within the simulation parameters.

If the system state indicates that the system is functional and the simulation time has not exceeded the maximum simulation time, the fault times of newly introduced application instances and recovered components must be determined. Note that to calculate the fault time, the current simulation time is added to a random value from the specified failure probability distribution since those fault times are calculated at a later stage in the simulation. Afterwards, the simulation of the current iteration is continued by injecting the fault that is next in chronological order.

In case the system state indicates that the system is not functional, then the current iteration is aborted. AT-CARS then starts the next iteration unless the total number of iterations is already achieved.

Once all iterations have been completed, the reliability of the system is determined. The reliability, $R(t)$, over time t is given by

$$R(t) = \frac{\sum_{i=1}^{N} f(t, t_{i,failed})}{N},$$

where t denotes the time in hours, i the iteration number, and N the total number of iterations. Furthermore, the function f is defined thus:

$$f(t, t_{i,failed}) = \begin{cases} 1, & t < t_{i,failed} \\ 0, & \text{otherwise} \end{cases},$$

where $t_{i,failed}$ indicates the time at which the system state transitioned to not functional in iteration i. In case the system did not fail within the maximum simulation time, $t_{i,failed} = \infty$ holds.

Appendix A.4 outlines the implementation of AT-CARS in pseudo-code.

5.6 Summary of the Component Layer

In this chapter, we introduced the component layer of APTUS, which implements the self-healing property. It contains two main components: FDIRO and HRR.

FDIRO is a fault-tolerance approach consisting of a fault detection, isolation, recovery, and optimization step. These steps are executed in response to a hardware or software fault, aiming to restore the system to a safe and optimized state. To further enhance FDIRO, we introduced HRR, which enables the recovery of failed hardware components.

In addition to FDIRO and HRR, we also presented in this chapter AT-CARS, which is a simulation environment that employs a Monte Carlo simulation to evaluate the reliability of the configurations computed by APTUS. In the next chapter, AT-CARS is used to analyze the impact of FDIRO and HRR on system reliability.

CHAPTER 6

Use-Case Scenario

We now present a use-case scenario that illustrates the overall workflow of APTUS. In this scenario, an autonomous vehicle performs commercial missions in an urban environment.

Section 6.1 defines the conditions assumed in this use case. In Section 6.2, we illustrate the workflow of the configuration layer in three stages, whereby the vehicle experiences a different current context in the individual stages. Section 6.3, then, shows the processing of the determined software-architecture requirements by the reconfiguration layer. Furthermore, a hardware fault is discussed in order to illustrate the interplay between the component layer and the reconfiguration layer. Note that for the sake of simplicity, we omit the procedures performed by D-DEG and CGM in this example. Finally, in Section 6.4, we analyze a previously computed configuration using AT−CARS, and Section 6.5 concludes with a brief summary.

6.1 Example Setup

The autonomous vehicle assumed in this example is equipped with the functions required for autonomous driving, which are defined in Section 2.1. Furthermore, the vehicle is capable of performing the functions described in Table 3.1. The applications which implement those functions are listed in Table 6.1.

The level of redundancy and the level of separation are set to 2 for all applications of the localization function, the sensor-fusion function, the ADS-mode-manager function, the interpretation and prediction function, the drive-planning function, the motion-control function, and the body-control function. Furthermore, the level of diversity of those applications is set to 1. The level of redundancy and the level of diversity of the applications that implement the update-management function are set to 1, and the level of separation is set to 2. For the remaining applications, the level of redundancy, the level of diversity, and the level of separation are set to 0. The memory demands, the

Table 6.1: The applications that implement the defined functions.

Functions	Applications
localization	loc1, loc2
sensor_fusion	fus1, fus2, fus3
ads_mode_manager	amm1, amm2
interpretation_prediction	int_pred1, int_pred2
drive_planning	dr_plan1, dr_plan2, dr_plan3
motion_control	m_cont1, m_cont2
body_control	b_cont1, b_cont2
traffic_optimization	tfc_opt1
ride_management	rd_mgmt1
update_management	up_mgmt1, up_mgmt2
shared_event_recording	sh_ev_rec1
logging	log1
ride_visualization	rd_vis1
entertainment	ent1

performance demands, and the list of required supporting software of the applications are defined in Table 6.2.

Furthermore, we assume that the vehicle is equipped with three computing nodes. Table 6.3 shows the provided memory, performance, and supporting software resources. Note that the resource parameter values of the applications and computing nodes are assumed values.

The environmental-context values that we consider in this example are clear, rainy, city, daylight, and dark. Furthermore, we define the environmental-context sets clear_city, city_daylight, rainy_dark, city_dark, rainy_city_dark, and clear_city_daylight. Note that the names of environmental-context sets indicate the environmental-context values that are part of the sets.

For the applications which implement the localization function, the sensor-fusion function, the interpretation and prediction function, the drive-planning function, and the motion-control function we define environmental-context ratings. The ratings are given in Tables 6.4, 6.5, and 6.6. For the applications of the remaining functions, we do not specify environmental-context ratings. Note that the environmental-context ratings used in this example are assumed values.

The vehicle-operation modes, operation properties, and user contexts that we use in this example are those defined in Section 3.3.1. We also use the mapping between functions and the introduced vehicle-operation modes, operation properties, and user contexts that were introduced in that section. Furthermore, we use the optimization functions and relations between optimization functions and context information as defined in that same section.

Table 6.2: The memory demands, the performance demands, and the required supporting software of the applications. Note that the memory demands are given in Megabytes and the performance demands in TFLOPS (10^{12} floating point operations per second).

	memory	performance	req_supporting_software
loc1	2000	70	cuda
loc2	2500	80	cuda
fus1	4000	130	cuda
fus2	3000	120	cuda
fus3	3000	120	cuda
amm1	1000	20	-
amm2	1500	20	-
int_pred1	5000	100	cuda
int_pred2	4500	90	cuda
dr_plan1	3000	80	cuda
dr_plan2	4000	90	cuda
dr_plan3	4000	100	cuda
m_cont1	2000	70	-
m_cont2	1500	80	-
b_cont1	1000	50	-
b_cont2	1000	60	-
tfc_opt1	2000	60	-
rd_mgmt1	1000	10	java
up_mgmt1	6000	20	-
up_mgmt2	6500	20	-
sh_ev_rec1	2000	40	-
log1	2000	40	-
rd_vis1	2000	50	java
ent1	3000	60	java

6.2 Software-Architecture Requirement Determination

This section illustrates the workflow of the configuration layer. As mentioned before, we define three stages, which define a different current context. Table 6.7 gives an overview of the current context that the vehicle experiences in the individual stages.

In what follows, we describe the output of C-SAR in those stages. Recall that we do not consider software-architecture degradations determined by D-DEG in this example.

6.2.1 First Stage

In the first stage, we assume that the autonomous vehicle is operated in autonomy mode. Furthermore, we define that the passengers traveling with the vehicle booked a premium

Table 6.3: The provided memory, the provided performance, and the provided supporting software of the computing nodes. Note that the provided memory is given in Megabytes and the provided performance is specified in TFLOPS (10^{12} floating point operations per second).

	memory	performance	supporting_software
cn1	32000	1000	cuda
cn2	32000	750	cuda, java
cn3	64000	1000	cuda

Table 6.4: The environmental-context ratings of loc1, fus1, int_pred1, dr_plan1, and m_cont1. Note that "-" indicates that no rating is given.

	loc1	fus1	int_pred1	dr_plan1	m_cont1
clear	82	70	84	79	72
rainy	51	42	72	71	87
city	79	74	92	80	91
daylight	84	71	89	65	84
dark	63	77	76	61	86
clear_city	82	72	-	92	76
clear_daylight	89	65	-	89	71
city_daylight	78	73	-	82	79
clear_city_daylight	86	-	-	-	82
rainy_city	62	39	-	-	70
rainy_dark	-	-	-	-	63
city_dark	-	-	-	-	68
rainy_city_dark	-	-	-	-	72

ride. The ride is performed in daylight, and the weather is clear. Moreover, we assume that the vehicle is low on battery.

Figure 6.1 illustrates the output of C-SAR in the first stage. The computer used to generate this output was equipped with an Apple M2 chip with 8 cores and 8 GB memory. C-SAR generated the output in 11 ms.

As can be seen in the output, the optimization functions selected for the current context are min_displ_act, min_comp_nodes, and max_sep, whereby min_displ_act is of the highest priority. The optimization functions min_displ_act and min_comp_nodes are selected since the current context contains the vehicle operation mode autonomous. On the other hand, the optimization function min_comp_nodes is selected since the vehicle is low on battery, i.e., the operation property low_power is part of the current context.

Since the vehicle is operated in the autonomous operation mode, all functions which are

```
selected_optimization_function(min_displ_act, 80).
selected_optimization_function(min_comp_nodes, 60).
selected_optimization_function(max_sep, 40).

selected_function(localization, 2).
selected_function(sensor_fusion, 2).
selected_function(ads_mode_manager, 2).
selected_function(interpretation_prediction, 2).
selected_function(drive_planning, 2).
selected_function(motion_control, 2).
selected_function(body_control, 2).
selected_function(ride_management, 1).
selected_function(shared_event_recording, 1).
selected_function(ride_visualization, 0).
selected_function(entertainment, 0).

active_application(loc1, 0).
active_application(fus2, 0).
active_application(amm1, 0).
active_application(int_pred1, 0).
active_application(dr_plan1, 0).
active_application(m_cont1, 0).
active_application(b_cont1, 0).
active_application(rd_mgmt1, 0).
active_application(sh_ev_rec1, 0).
active_application(rd_vis1, 0).
active_application(ent1, 0).

active_hot_application(loc1, 1).
active_hot_application(loc2, 2).
active_hot_application(fus2, 1).
active_hot_application(fus1, 2).
active_hot_application(amm1, 1).
active_hot_application(amm2, 2).
active_hot_application(int_pred1, 1).
active_hot_application(int_pred2, 2).
active_hot_application(dr_plan1, 1).
active_hot_application(dr_plan2, 2).
active_hot_application(m_cont1, 1).
active_hot_application(m_cont2, 2).
active_hot_application(b_cont1, 1).
active_hot_application(b_cont2, 2).
```

Figure 6.1: Partial output of C-SAR in first stage of the discussed example.

Table 6.5: The environmental-context ratings of loc2, fus2, int_pred2, dr_plan2, m_cont2. Note that “-” indicates that no rating is given.

	loc2	fus2	int_pred2	dr_plan2	m_cont2
clear	76	83	72	71	79
rainy	72	84	81	70	87
city	70	79	80	72	70
daylight	65	82	75	74	72
dark	75	-	69	70	89
clear_city	79	69	-	89	-
clear_daylight	-	71	-	82	-
city_daylight	86	68	-	71	69
clear_city_daylight	82	-	-	-	-
rainy_city	90	87	-	79	91
rainy_dark	-	-	-	-	92
city_dark	82	-	-	-	89
rainy_city_dark	87	-	-	73	81

Table 6.6: The environmental-context ratings of fus3 and dr_plan3. Note that “-” indicates that no rating is given.

	fus3	dr_plan3
clear	64	72
rainy	89	82
city	71	92
daylight	61	78
dark	82	77
clear_city	-	-
clear_daylight	-	-
city_daylight	-	-
clear_city_daylight	-	-
rainy_city	83	83
rainy_dark	87	79
city_dark	84	-
rainy_city_dark	92	82

required for operating the vehicle autonomously, as well as the shared event-recording function and the ride-visualization function, are selected. Furthermore, since the passengers booked a premium ride, the entertainment function is also selected.

Recall that the applications which implement the localization function, the sensor-fusion function, the interpretation and prediction function, the drive-planning function, and the motion-control function are rated applications. Consequently, C-SAR has to consider the

Table 6.7: The current context of the three stages.

	First Stage	Second Stage	Third Stage
vehicle-operation mode	autonomous	parked	autonomous
operation properties	commercial, low_power	commercial	commercial
user context	premium_ride	-	low_cost_ride
environmental-context values	clear, city, daylight	clear, city, daylight	rainy, city, dark

context-environment ratings in order to select the active application.

In what follows, we discuss the selection of the active and active-hot applications that implement the localization function and the sensor-fusion function, respectively. Note that the selection of the active application and the active-hot applications for the interpretation and prediction function, the driving planning function, and the motion-control function follow the same selection criteria.

The localization function is implemented by the application `loc1` and `loc2`. In the current context, `loc1` is used as an active application since its environmental-context rating for the environmental-context sets `clear_city_daylight` is higher than the rating of `loc2`. Note that the specialization level of `loc1` and `loc2` is 3 in the current context.

Recall the level of redundancy required by `loc1` is 2 and the level of diversity is 1. Consequently, two active-hot applications have to be determined. As the level of diversity is 1, the first active-hot application is like the active application, an instance of `loc1`. The second active-hot application is an instance of `loc2`.

The sensor-fusion function is implemented by three applications, `fus1`, `fus2`, and `fus3`. The specialization level of `fus1` and `fus2` is 2, and the specialization level of `fus3` is 1 in the current context. Since the specialization level of `fus3` is lower than the specialization level of `fus1` and `fus2`, this application is not selected as an active application by C-SAR. The applications `fus2` and `fus1` are both rated for the environmental-context sets `clear_city`, `clear_daylight`, and `city_daylight`. Consequently, for each of those two applications the lowest rating concerning the before mentioned environmental-context sets is used to determine whether `fus1` and `fus2` is selected as the active application. The rating that is considered for `fus1` is 65 and the rating for `fus2` is 68. As a result, `fus2` is selected as an active application. The lower rated `fus1` is used as a diverse active-hot application.

For the ADS-mode-manager function, the body-control function, the shared event-recording function, the ride-visualization function, and the entertainment function, only non-rated applications exist. Recall that two applications implement the ADS-mode-manager function and the motion-control function, respectively. Therefore, C-SAR selects

```
selected_optimization_function(min_comp_nodes, 60).

selected_function(ride_management, 1).
selected_function(traffic_optimization, 0).
selected_function(update_manager, 2).

active_application(up_mgmt1, 0).
active_application(rd_mgmt1, 0).
active_application(tfc_opt1, 0).

active_hot_application(up_mgmt2, 1).
```

Figure 6.2: Partial output of C-SAR in second stage of the discussed example.

the first application in the alphabetically ascending order. Since the level of diversity required by those applications is 2, the remaining application is used as an active-hot application.

On the other hand, only one non-rated application exists for the ride-management function, the shared event-recording function, and the ride-visualization function. Consequently, these applications are selected as active applications. Furthermore, the level of redundancy for these applications is 0. Therefore, active-hot applications are not required.

6.2.2 Second Stage

After the requested ride in the first stage is completed, the vehicle is parked and charged. The illumination and weather conditions in this stage are the same as in the previous one. However, in the course of this stage, the weather conditions change from clear to rainy, and the sun sets.

Figure 6.2 illustrates the output of C-SAR in the second stage. C-SAR generated this output in 10 ms. Note that the changes in the current context described above do not influence the output of C-SAR during the second stage.

Since in the current context, the vehicle-operation mode is `parked`, the optimization function `min_comp_nodes` is selected.

Compared to the previous stage, C-SAR only selected three functions for the current context, i.e., the ride-management function, the traffic-optimization function, and the update-management function. The ride-management function and the traffic-optimization function are each implemented by one application, i.e., `rd_mgmt1` and `tfc_opt1`. Since those applications do not require redundancies, no active-hot applications are selected. The update-management function is implemented by `up_mgmt1` and `up_mgmt2`. C-SAR selects up_mgmt1 as active application since this application is listed first in alphabetical order. Consequently, up_mgmt2 is selected as active-hot application.

6.2.3 Third Stage

After the vehicle has been charged, it performs another ride. In the third stage, we assume that a customer booked a low-cost ride. Furthermore, we define that the weather is rainy and that night has fallen.

Figure 6.3 illustrates the output of C-SAR in the third stage. C-SAR generated this output in 11 ms. As can be seen in the output, the optimization functions selected for the current context are min_displ_act and max_sep, whereby min_displ_act is of the highest priority. Those optimization functions are selected since the current context contains the vehicle operation mode autonomous.

As in the first stage, all functions required for the autonomous operation of the vehicle are selected. Furthermore, the ride-management function, the shared-event recording function, as well as the ride-visualization function are selected. Unlike in the first stage, the entertainment function is not selected since the passengers booked a low-cost ride. However, instead, the traffic-optimization function is selected.

Compared to the current context of the first stage, the current context of the third stage includes a different set of environmental-context values. In particular, the weather in the third stage is not clear anymore but rainy, and the illumination changed from daylight to dark. Consequently, for some of the functions for which only rated applications exist, i.e., the localization function, the sensor-fusion function, the interpretation and prediction function, the drive-planning function, and the motion-control function, the selection of active and active-hot applications differ compared to the selection illustrated in the first stage.

The selection of the active and active-hot applications for the localization function and the sensor-fusion function shows that different applications are selected compared to the first stage due to the diverging environmental-context values in the current context.

Recall that the localization application loc1 was selected as an active application in the first stage. In the current context, the specialization level of loc1 is 2 since this application is rated for the environmental-context set rainy_city. On the other hand, the specialization level of loc2 is 3. Therefore, loc2 is used as an active application. Since the level of diversity of loc2 is 1, loc1 is used as an active-hot application.

Compared to the first stage, also different sensor fusion applications are selected as active and active-hot applications. In the current context, the level of specialization of fus3 is 3, while on the other hand, the level of specialization of fus1 and fus2 is 2. Therefore, fus3 is selected as an active application. As the environmental-context rating that is considered for fus2 is higher than the rating of fus1 and the level of diversity of fus3 is 1, fus2 is used as an active-hot application.

```
selected_optimization_function(min_displ_act, 80).
selected_optimization_function(max_sep, 40).

selected_function(localization, 2).
selected_function(sensor_fusion, 2).
selected_function(ads_mode_manager, 2).
selected_function(interpretation_prediction, 2).
selected_function(drive_planning, 2).
selected_function(motion_control, 2).
selected_function(body_control, 2).
selected_function(traffic_optimization, 0).
selected_function(ride_management, 1).
selected_function(shared_event_recording, 1).
selected_function(ride_visualization, 0).

active_application(loc2, 0).
active_application(fus3, 0).
active_application(amm1, 0).
active_application(int_pred1, 0).
active_application(dr_plan2, 0).
active_application(m_cont2, 0).
active_application(b_cont1, 0).
active_application(tfc_opt1, 0).
active_application(rd_mgmt1, 0).
active_application(sh_ev_rec1, 0).
active_application(rd_vis1, 0).

active_hot_application(loc2, 1).
active_hot_application(loc1, 2).
active_hot_application(fus3, 1).
active_hot_application(fus2, 2).
active_hot_application(amm1, 1).
active_hot_application(amm2, 2).
active_hot_application(int_pred1, 1).
active_hot_application(int_pred2, 2).
active_hot_application(dr_plan2, 1).
active_hot_application(dr_plan3, 2).
active_hot_application(m_cont2, 1).
active_hot_application(m_cont1, 2).
active_hot_application(b_cont1, 1).
active_hot_application(b_cont2, 2).
```

Figure 6.3: Partial output of C-sar in third stage of the discussed example.

Note that compared to the first stage, the selection of the active and active-hot application also differs for the drive-planning function and the motion-control function. On the other hand, the same active and active-hot applications are selected for the ADS-mode-manager function and the interpretation and prediction function in the respective stages.

As in the first stage `rd_mgmt1`, `sh_ev_rec1`, `rd_vis1` are selected as active applications for the ride-management function, the shared event-recording function, and the ride-visualization function. Furthermore, since the current context requires the execution of the traffic-optimization function, C-SAR selects `tfc_opt1` as an active application.

6.3 Configuration Determination

In this section, we illustrate the procedures performed by the configuration layer by continuing the example introduced in the previous section. In particular, we show the configurations determined by the configuration layer in the second stage and the third stage. Furthermore, in order to illustrate the interplay between the configuration layer and the component layer, we extend the latter stage by introducing a fault of a computing node.

This example discusses six successive configurations. We refer to the first configuration as C_1, and the final configuration is called C_6. Furthermore, we assume that the configuration C_0, which is the predecessor configuration of C_1, holds the configuration that meets the software-architecture requirements of the first stage. However, note that we do not discuss this configuration in this example. Figure 6.4 shows the configurations in the configuration graph and their classification into the levels defined by Λ_{prio}.

In what follows we discuss the individual configurations.

6.3.1 Configuration C_1

Recall that after the ride of the first stage is completed, the autonomous vehicle is in the second stage, parked and charged. Due to this context change, a new configuration is required. We refer to the configuration that is applied in the second stage as C_1.

The transition between C_1 and C_0 is defined as $(\iota_u, \rho_{\text{stage2}})$. Note that $F(C_0, (\iota_u, \rho_{\text{stage2}}))= C_1$ holds, where F is the function that computes new configurations. The initiator event ι_u signals that a new configuration is required due to refined software-architecture requirements. The new software-architecture requirements are defined by ρ_{stage2}. These requirements include those illustrated in Subsection 6.2.2 as well as the corresponding application demands listed in Table 6.2.

After `AP`^{2}`Initiator` has set up the application-placement problem, `logAP`^{2}`S` is selected for solving it. The application-placement problem solver `logAP`^{2}`S` is selected since the initiator event specifies a refinement of the software-architecture requirements.

Before `logAP`^{2}`S` is instructed to solve the application-placement problem, C-PO$_{\text{APTUS}}$ determines the optimization function. As illustrated in Figure 6.4, we assume that

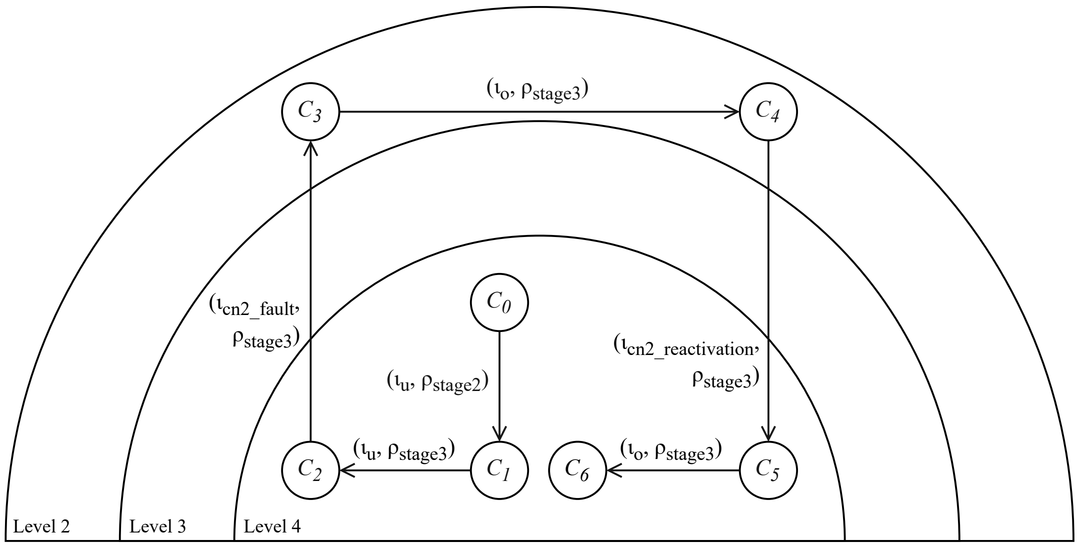

C_3
C_4
(ι_o, ρ_{stage3})
C_0
$(\iota_{cn2\_fault}, \rho_{stage3})$
(ι_u, ρ_{stage2})
$(\iota_{cn2\_reactivation}, \rho_{stage3})$
C_2
(ι_u, ρ_{stage3})
C_1
C_6
(ι_o, ρ_{stage3})
C_5
Level 2
Level 3
Level 4

Figure 6.4: Illustration of configuration C_0, C_1, C_2, C_3, C_4, C_5, and C_6 in the configuration graph. The depicted levels correspond to the levels defined by Λ_{prio}.

$\Lambda_{prio}(C_0) = 4$ holds. Consequently, C-POAPTUS selects the optimization goal determined by C-SAR. Recall that C-SAR, selected for the second stage the optimization function `min_comp_nodes`.

Figure 6.5 illustrates the configuration determined by logAP^{2}S. The active and active-hot applications determined by C-SAR are assigned to computing node 2 and computing node 3, respectively. Note that no applications are assigned to computing node 1 since the optimization function `min_comp_nodes` aims to utilize as few computing nodes as possible. Nevertheless, logAP^{2}S has to use two computing nodes for fulfilling the software-architecture requirements defined by ρ_{stage2}. This is because the update-management function requests a level of separation of 2.

Due to the limited number of applications required in this stage, the memory and performance restrictions can be neglected since enough resources are provided. Furthermore, the applications `tfc_opt1` and `up_mgmt1` do not define any required supporting software. Consequently, the assignment of these applications is not restricted due to the supporting software offered by the computing nodes. However, the application `rd_mgmt1` requires the supporting software `java`. Therefore, `rd_mgmt1` is assigned to computing node 2 since this computing node is the only one providing `java`.

6.3.2 Configuration C_2

After the second stage, the autonomous vehicle is booked for another ride in the third stage. Therefore, C-SAR selects all functions which perform the autonomous driving task. Furthermore, C-SAR selects the traffic-optimization function, the ride-management

```
assignment(up_mgmt1, 1, cn2).
assignment(tfc_opt1, 0, cn2).
assignment(rd_mgmt1, 0, cn2).
assignment(up_mgmt1, 0, cn3).
```

Figure 6.5: The configuration C_1, which is determined by `logAP`^{2}`S`.

```
assignment(loc2, 0, cn1).
assignment(fus3, 0, cn1).
assignment(amm1, 1, cn1).
assignment(int_pred2, 2, cn1).
assignment(dr_plan2, 1, cn1).
assignment(m_cont2, 1, cn1).
assignment(b_cont1, 0, cn1).
assignment(loc2, 1, cn2).
assignment(fus2, 2, cn2).
assignment(amm1, 0, cn2).
assignment(int_pred1, 1, cn2).
assignment(dr_plan2, 0, cn2).
assignment(m_cont1, 2, cn2).
assignment(b_cont1, 1, cn2).
assignment(tfc_opt1, 0, cn2).
assignment(rd_mgmt1, 0, cn2).
assignment(rd_vis1, 0, cn2).
assignment(loc1, 2, cn3).
assignment(fus3, 1, cn3).
assignment(amm2, 2, cn3).
assignment(int_pred1, 0, cn3).
assignment(dr_plan3, 2, cn3).
assignment(m_cont2, 0, cn3).
assignment(b_cont2, 2, cn3).
assignment(sh_ev_rec1, 0, cn3).
```

Figure 6.6: The configuration C_2, which is determined by `logAP`^{2}`S`.

function, the shared-event recording function, and the ride-visualization function to be executed in this stage. The software-architecture requirements of the third stage are contained in ρ_{stage3}.

Since the initiator event again specifies a refinement of the software-architecture requirement, `AP`^{2}`Initiator` selects `logAP`^{2}`S` for solving the application-placement problem.

Like before, C-po$_{\text{Aptus}}$ selects the optimization function that was selected by C-sar as $\Lambda_{prio}(C_1) = 4$ holds.

Note that C-sar selected in the third stage the optimization functions `min_displ_act` and `max_sep`, whereby the priority of the former is higher than the priority of the latter. The optimization function `min_displ_act` aims to minimize the displacements of active applications with respect to the previous configuration. Consequently, as illustrated in Figure 6.6, the active applications that implement the traffic-optimization function and the ride-management function, i.e., `tfc_opt1` and `rd_mgmt1`, remain assigned to computing node 2.

As can be seen in Figure 6.6, the applications that perform the automated driving task are evenly distributed among the three computing nodes. Note that the level of redundancy and the required level of separation of the functions which perform the automated driving is 2. Due to the separation requirement, the active application and the two active-hot application instances of these functions must be assigned to at least two computing nodes. However, because of the optimization function `max_sep`, each function is distributed among all three computing nodes.

Furthermore, note that C_2 obeys the provided memory capacities of the computing nodes. The sums of the memory demands of the applications assigned to computing node 1, computing node 2, and computing node 3 are 17000, 23500, and 20000 Megabytes, respectively. On the other hand, the computing nodes are equipped with at least 32000 Megabytes of memory. Moreover, the configuration C_2 also respects the performance capacities of the computing nodes. The sums of the performance demands of the applications assigned to computing node 1, computing node 2, and computing node 3 are 540, 650, and 590 TFLOPS. Nevertheless, each computing node provides at least 750 TFLOPS.

6.3.3 Configuration C_3

During the execution of the third stage, we assume that computing node 2 fails. In the first step, the fault is detected by the component layer. The component that is responsible for detecting and handling the fault is Fdiro.

In response to the fault, Fdiro tries to activate a redundant application that can take over the tasks of the active applications which were executed by computing node 2, i.e., `amm1`, `dr_plan2`, `tfc_opt1`, `rd_mgmt1`, and `rd_vis1`. Fdiro selects the active-hot applications which have the lowest redundancy instance number. Accordingly, the operation modes of the applications `amm1` and `dr_plan2`, which are executed by computing node 1, are upgraded to active. Since `rd_mgmt1`, and `rd_vis1` are the only applications executing the ride-management function and the ride-visualization function, Fdiro cannot activate any redundant application for those functions.

Next, Fdiro aims to find a new assignment for all applications which were executed by computing node 2. Therefore, Fdiro sends a reconfiguration request which includes the initiator event $\iota_{\mathrm{cn2\_fault}}$ to the `AP`^{2}`Initiator`. The initiator event $\iota_{\mathrm{cn2\_fault}}$ indicates that computing node 2 failed. Besides the initiator event, the software-architecture requirements, and the currently applied configuration, `AP`^{2}`Initiator` also analyzes the current operation modes of the applications for setting up the application-placement problem. Note that the current operation modes of the applications need to be considered since they might distinguish from the initial operation modes stated in the software-architecture requirements due to the actions executed by FDIRO.

After setting up the application-placement problem, `AP`^{2}`Initiator` selects `linAP`^{2}`S` for solving it since a hardware fault triggered the reconfiguration. The aim of `linAP`^{2}`S` is to determine a configuration that reassigns the applications executed by computing node 2.

Recall that the recovery procedure of FDIRO requires a fast reconfiguration of the system. Therefore, `linAP`^{2}`S` employs linear programming and a parallel-solving heuristic. The parallel-solving heuristic is implemented by computing three application-placement problems of different complexity. The three individual application-placement problems are distinct from one another in terms of the set of functions they consider. The sets of functions are referred to as F_0, F_1, and F_2. The set F_0 contains all functions included in ρ_{stage3}. On the other hand, F_1 holds the same functions as F_0 except for the ride-management function and the shared-event recording function, i.e., the functions of priority LOW. Likewise, F_2 contains the functions of F_1 excluding the traffic-optimization function and the ride-visualization function, which are of priority $MEDIUM$.

For the application-placement problems that consider the sets F_0 and F_1, no solution exists. This is because computing node 2 is the only computing node that offers the supporting software `java`, which is, however, required by the applications `rd_mgmt1` and `rd_vis1`. The applications `rd_mgmt1` and `rd_vis1` implement the ride-visualization function and the ride-management function, respectively. Both these functions are included in F_0, and the ride-management function is part of F_1.

However, a solution exists for the application-placement problem that considers the functions included in F_2. Figure 6.7 illustrates the configuration that is determined by `linAP`^{2}`S`, i.e., configuration C_3.

The applications which execute a function of priority $HIGH$ that were assigned to computing node 2 in configuration C_2 are assigned to computing node 1 and computing node 3, respectively in C_3. Since `linAP`^{2}`S` aims to minimize the displacements of applications, the assignments of the applications which are assigned to computing node 1 and computing node 3 in C_2 do not change in C_3. Note that `sh_ev_rec1` is not assigned to computing node 3 in C_3 since the shared-event recording function is of priority LOW. Therefore, this function is not part of F_2.

6.3.4 Configuration C_4

After the recovery procedure is completed, FDIRO initiates an optimization of the configuration. For executing this step, FDIRO uses again the reconfiguration layer.

```
assignment(loc2, 0, cn1).
assignment(fus3, 0, cn1).
assignment(fus2, 2, cn1).
assignment(amm1, 1, cn1).
assignment(int_pred2, 2, cn1).
assignment(int_pred1, 1, cn1).
assignment(dr_plan2, 1, cn1).
assignment(m_cont2, 1, cn1).
assignment(m_cont1, 2, cn1).
assignment(b_cont1, 0, cn1).
assignment(b_cont1, 1, cn1).
assignment(loc1, 2, cn3).
assignment(loc2, 1, cn3).
assignment(fus3, 1, cn3).
assignment(amm2, 2, cn3).
assignment(amm1, 0, cn3).
assignment(int_pred1, 0, cn3).
assignment(dr_plan3, 2, cn3).
assignment(dr_plan2, 0, cn3).
assignment(m_cont2, 0, cn3).
assignment(b_cont2, 2, cn3).
```

Figure 6.7: The configuration C_3, which is determined by `linAP`^{2}`S`.

In particular, FDIRO sends a reconfiguration request to the `AP`^{2}`Initiator` using the optimization initiator event ι_o. The `AP`^{2}`Initiator` sets up the application-placement problem based on the software-architecture requirements ρ_{stage3}, the current operation modes of the applications, and the current configuration.

Due to the initiator event ι_o, `AP`^{2}`Initiator` selects `logAP`^{2}`S` for solving the application-placement problem. Note that since $\Lambda_{prio}(C_3) = 2$ holds, C-PO$_{\text{APTUS}}$ selects the optimization function that aims to introduce currently not assigned applications so that

$$\Lambda_{prio}(C_3) \leq \Lambda_{prio}(C_4)$$

holds.

As can be seen in Figure 6.8, which illustrates C_4, the applications `sh_ev_rec1` and `tfc_opt1` are assigned to computing node 1 and computing node 3, respectively. Apart from these two assignments, the configurations C_3 and C_4 are the same. This is because of the optimization function that aims to minimize the number of displacements of applications. Furthermore,

$$\Lambda_{prio}(C_3) = \Lambda_{prio}(C_4)$$

```
assignment(loc2, 0, cn1).
assignment(fus3, 0, cn1).
assignment(fus2, 2, cn1).
assignment(amm1, 1, cn1).
assignment(int_pred2, 2, cn1).
assignment(int_pred1, 1, cn1).
assignment(dr_plan2, 1, cn1).
assignment(m_cont2, 1, cn1).
assignment(m_cont1, 2, cn1).
assignment(b_cont1, 0, cn1).
assignment(b_cont1, 1, cn1).
assignment(sh_ev_rec1, 0, cn1).
assignment(loc2, 1, cn3).
assignment(loc1, 2, cn3).
assignment(fus3, 1, cn3).
assignment(amm2, 2, cn3).
assignment(amm1, 0, cn3).
assignment(int_pred1, 0, cn3).
assignment(dr_plan3, 2, cn3).
assignment(dr_plan2, 0, cn3).
assignment(m_cont2, 0, cn3).
assignment(b_cont2, 2, cn3).
assignment(tfc_opt1, 0, cn3).
```

Figure 6.8: The configuration C_4, which is determined by `logAP`^{2}`S`.

holds, i.e., the performed optimization could not increase the Λ_{prio} level of the configuration. The reason for this is that the application `rd_vis1` and `rd_mgmt1` cannot be assigned to any computing node since neither computing node 1 nor computing node 2 provides the supporting software `java`.

6.3.5 Configuration C_5

We assume that after the fault of computing node 2 and the performed recovery and optimization procedures, computing node 2 gets reactivated by HRR.

After reactivating computing node 2, HRR triggers a reconfiguration using the initiator event $\iota_{\text{cn2\_reactivation}}$. Based on this initiator event, `AP`^{2}`Initiator` selects `logAP`^{2}`S` for solving the application-placement problem.

Since $\Lambda_{prio}(C_4) = 2$ holds, C-POAPTUS selects as before the optimization function that aims to introduce currently not assigned applications in order to reach a higher Λ_{prio} level. Recall that `rd_vis1` and `rd_mgmt1` could not be assigned to a computing node

```
assignment(loc2, 0, cn1).
assignment(fus3, 0, cn1).
assignment(fus2, 2, cn1).
assignment(amm1, 1, cn1).
assignment(int_pred2, 2, cn1).
assignment(int_pred1, 1, cn1).
assignment(dr_plan2, 1, cn1).
assignment(m_cont2, 1, cn1).
assignment(m_cont1, 2, cn1).
assignment(b_cont1, 0, cn1).
assignment(b_cont1, 1, cn1).
assignment(sh_ev_rec1, 0, cn1).
assignment(rd_vis1, 0, cn2).
assignment(rd_mgmt1, 0, cn2).
assignment(loc1, 2, cn3).
assignment(loc2, 1, cn3).
assignment(fus3, 1, cn3).
assignment(amm2, 2, cn3).
assignment(amm1, 0, cn3).
assignment(int_pred1, 0, cn3).
assignment(dr_plan3, 2, cn3).
assignment(dr_plan2, 0, cn3).
assignment(m_cont2, 0, cn3).
assignment(b_cont2, 2, cn3).
assignment(tfc_opt1, 0, cn3).
```

Figure 6.9: The configuration C_5, which is determined by logAP^{2}S.

in C_4. However, since computing node 2 got reactivated, `rd_vis1` and `rd_mgmt1` can be assigned.

Figure 6.9 illustrates the configuration determined by logAP^{2}S. Note that in C_5 all applications contained in ρ_{stage3} are assigned to computing nodes. Consequently, $\Lambda_{prio}(C_5) = 4$ holds.

6.3.6 Configuration C_6

The current configuration has now again reached the highest level of Λ_{prio}, i.e., $\Lambda_{prio}(C_5) = 4$ holds. Therefore, C-POAPTUS can apply the optimization function selected by C-SAR.

This optimization is triggered by the reconfiguration layer itself by using the initiator event ι_o. Therefore, AP^{2}Initiator selects logAP^{2}S for solving the application-placement problem. The resulting configuration C_6 is depicted in Figure 6.10.

```
assignment(loc2, 0, cn1).
assignment(fus3, 0, cn1).
assignment(amm1, 1, cn1).
assignment(int_pred2, 2, cn1).
assignment(dr_plan2, 1, cn1).
assignment(m_cont2, 1, cn1).
assignment(b_cont1, 0, cn1).
assignment(sh_ev_rec1, 0, cn1).
assignment(loc2, 1, cn2).
assignment(fus2, 2, cn2).
assignment(amm1, 0, cn2).
assignment(int_pred1, 1, cn2).
assignment(dr_plan2, 0, cn2).
assignment(m_cont1, 2, cn2).
assignment(b_cont1, 1, cn2).
assignment(rd_mgmt1, 0, cn2).
assignment(rd_vis1, 0, cn2).
assignment(loc1, 2, cn3).
assignment(fus3, 1, cn3).
assignment(amm2, 2, cn3).
assignment(int_pred1, 0, cn3).
assignment(dr_plan3, 2, cn3).
assignment(m_cont2, 0, cn3).
assignment(b_cont2, 2, cn3).
assignment(tfc_opt1, 0, cn3).
```

Figure 6.10: The configuration C_6, which is determined by `logAP`^{2}`S`.

As mentioned before, C-PO$_{\text{APTUS}}$ selects the optimization functions which are specified in ρ_{stage3}, namely `min_displ_act` and `max_sep`. The optimization function `min_displ_act` causes that no active applications are displaced. On the other hand, the optimization function `max_sep` aims to maximize the separation of the applications that implement the same function. Therefore, the active-hot applications `loc2`, `fus2`, `amm1`, `int_pred1`, `dr_plan2`, `m_cont1`, and `b_cont1` are moved to computing node 2.

Consequently, the configuration C_6 fulfills, like configuration C_2, all software-architecture requirements demanded by ρ_{stage3}. In particular, these two configurations differ in terms of the assignment of the applications `sh_ev_rec1` and `tfc_opt1`. Furthermore, due to the fault of computing node 2, the active applications of the ADS-mode-manager function and the drive-planning function are now executed by computing node 1. In configuration C_2, computing node 2 executed the active applications of these two functions.

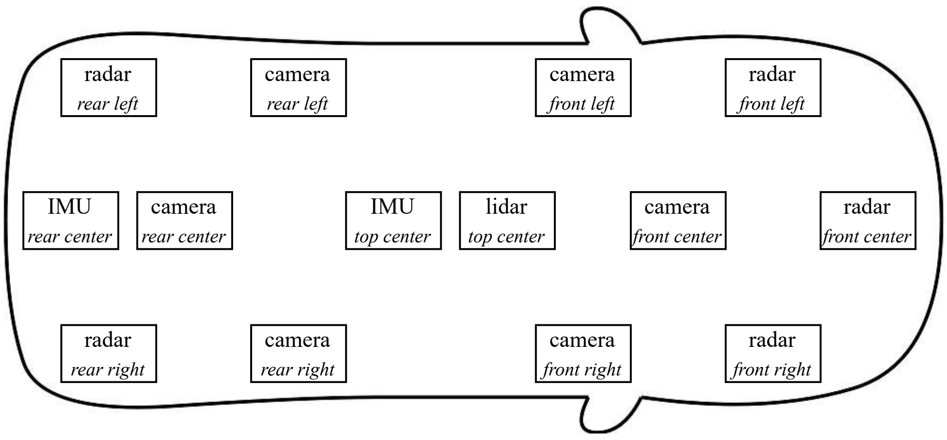


Figure 6.11: Overview of assumed sensor set.

Table 6.8: Fault distribution parameters of computing nodes.

Computing Node	Failure Probability Distribution
cn1	$T_{cn1,failure} \sim wei(t; \lambda = 2 \cdot 10^5, k = 1.5)$
cn2	$T_{cn2,failure} \sim wei(t; \lambda = 9 \cdot 10^4, k = 0.9)$
cn3	$T_{cn3,failure} \sim wei(t; \lambda = 4 \cdot 10^5, k = 2)$

6.4 Reliability Analysis of the Example Vehicle in the Third Stage using AT-CARS

In this section, we analyze the reliability of the example vehicle in the third stage using AT-CARS, which was introduced in Section 5.5.

Recall that the input parameters of AT-CARS are represented in three JSON files. Figure 6.12 illustrates an extract of J_1. The JSON file J_1 defines the properties of the computing nodes, the sensors, the functions, and the applications the example installed on the sample vehicle. Note that we assume that the vehicle considered in this example is equipped with six cameras, five radars, one lidar, and two IMUs. The arrangement of these sensors, which is shown in Figure 6.11., is based on the sensor configuration presented by nuTonomy [42].

For each component, the JSON file J_1 defines the failure probability distribution and the recovery probability. The complete list of failure probability distributions and recovery probabilities that we assume in this example is given in Tables 6.8–6.11. Note that the values defined in these tables are assumed values.

The JSON file J_2, which is depicted partially in Figure 6.13, specifies the initial configuration that is used for this example. Recall that we analyze the reliability of the example vehicle in the third stage. Consequently, the configuration C_2 is used as the initial configuration.

Table 6.9: Fault distribution parameters of applications.

Application	Failure Probability Distribution
loc1	$T_{loc1,failure} \sim exp(t; \lambda = 7 \cdot 10^{-5})$
loc2	$T_{loc2,failure} \sim exp(t; \lambda = 9 \cdot 10^{-5})$
fus2	$T_{fus2,failure} \sim norm(t; \mu = 10^{4}, \sigma = 9 \cdot 10^{3})$
fus3	$T_{fus3,failure} \sim exp(t; \lambda = 6 \cdot 10^{-5})$
amm1	$T_{amm1,failure} \sim exp(t; \lambda = 3 \cdot 10^{-6})$
amm2	$T_{amm2,failure} \sim exp(t; \lambda = 2 \cdot 10^{-6})$
int_pred1	$T_{int\_pred1,failure} \sim norm(t; \mu = 7 \cdot 10^{3}, \sigma = 8 \cdot 10^{4})$
int_pred2	$T_{int\_pred2,failure} \sim exp(t; \lambda = 4 \cdot 10^{-5})$
dr_plan2	$T_{dr\_plan2,failure} \sim exp(t; \lambda = 10^{-6})$
dr_plan3	$T_{dr\_plan3,failure} \sim exp(t; \lambda = 5 \cdot 10^{-5})$
m_cont1	$T_{m\_cont1,failure} \sim norm(t; \mu = 10^{4}, \sigma = 9 \cdot 10^{3})$
m_cont2	$T_{m\_cont2,failure} \sim exp(t; \lambda = 2 \cdot 10^{-6})$
b_cont1	$T_{b\_cont1,failure} \sim exp(t; \lambda = 10^{-6})$
b_cont2	$T_{b\_cont2,failure} \sim norm(t; \mu = 3 \cdot 10^{4}, \sigma = 8 \cdot 10^{4})$
tfc_opt1	$T_{tfc\_opt1,failure} \sim exp(t; \lambda = 2 \cdot 10^{-3})$
rd_mgmt1	$T_{rd\_mgmt1,failure} \sim exp(t; \lambda = 4 \cdot 10^{-3})$
rd_vis1	$T_{rd\_vis1,failure} \sim norm(t; \mu = 5 \cdot 10^{3}, \sigma = 2 \cdot 10^{3})$
sh_ev_rec1	$T_{sh\_ev\_rec1,failure} \sim exp(t; \lambda = 2 \cdot 10^{-3})$

Table 6.10: Fault distribution parameters of sensors.

Sensor	Failure Probability Distribution
radar_fl	$T_{radar\_fl,failure} \sim wei(t; \lambda = 2 \cdot 10^{5}, k = 1)$
radar_fc	$T_{radar\_fc,failure} \sim wei(t; \lambda = 2 \cdot 10^{6}, k = 1.2)$
radar_fr	$T_{radar\_fr,failure} \sim wei(t; \lambda = 2 \cdot 10^{5}, k = 1)$
radar_rl	$T_{radar\_rl,failure} \sim wei(t; \lambda = 2 \cdot 10^{5}, k = 1)$
radar_rr	$T_{radar\_rr,failure} \sim wei(t; \lambda = 2 \cdot 10^{5}, k = 1)$
camera_fl	$T_{camera\_fl,failure} \sim norm(t; \mu = 6 \cdot 10^{4}, \sigma = 5 \cdot 10^{4})$
camera_fc	$T_{camera\_fc,failure} \sim norm(t; \mu = 8 \cdot 10^{3}, \sigma = 9 \cdot 10^{4})$
camera_fr	$T_{camera\_fr,failure} \sim norm(t; \mu = 6 \cdot 10^{4}, \sigma = 5 \cdot 10^{4})$
camera_rl	$T_{camera\_rl,failure} \sim norm(t; \mu = 10^{4}, \sigma = 2 \cdot 10^{5})$
camera_rc	$T_{camera\_rc,failure} \sim norm(t; \mu = 3 \cdot 10^{4}, \sigma = 3 \cdot 10^{5})$
camera_rr	$T_{camera\_rr,failure} \sim norm(t; \mu = 10^{4}, \sigma = 2 \cdot 10^{5})$
lidar_tc	$T_{lidar\_tc,failure} \sim wei(t; \lambda = 5 \cdot 10^{5}, k = 1.3)$
imu_tc	$T_{imu\_tc,failure} \sim wei(t; \lambda = 4 \cdot 10^{6}, k = 1.2)$
imu_rc	$T_{imu\_rc,failure} \sim wei(t; \lambda = 9 \cdot 10^{5}, k = 1.1)$

```
[
   {
      "id": "cn1",
      "type": "computing_node",
      "memory": 32000,
      "performance": 1000,
      "supporting_software": ["cuda"],
      "failure": { "model": "weibull", "lambda": 2E5, "k": 1.5 },
      "recovery": "(t<=9000) * ((0.6-1)/9000*t+1) + (t>9000) * 0.6"
   },
   {
      "id": "radar_fl",
      "type": "sensor",
      "sensor_group": "camera_radar_lidar",
      "failure": { "model": "weibull", "lambda": 2E5, "k": 1 },
      "recovery": "0.5 + (1 - 0.5) * exp(-0.0002 * t)"
   },
   {
      "id": "localization",
      "type": "function",
      "priority": "HIGH"
   },
   {
      "id": "loc1",
      "type": "application",
      "function": "localization",
      "memory": 2000,
      "performance": 70,
      "redundancy": 2,
      "separation": 2,
      "diversity": 1,
      "supporting_software": ["cuda"],
      "failure": { "model": "exponential", "lambda": 7E-5 },
      "recovery": "0.3 + (1 - 0.3) * exp(-0.005 * t)"
   }, ...
]
```

Figure 6.12: Extract of the JSON file J_1.

The simulation parameters that underlie our analysis are defined in the JSON file J_3, which is depicted in Figure 6.14. In particular, we set the number of iterations to 1,000. Furthermore, we set the simulation time to 10,000 hours, which approximately corresponds to an autonomous taxi that is operated 14 hours per day for two years. Note that we do not consider a change in the environmental context, the operational context, and the user context during the simulation time.

Besides the number of iterations and the maximum simulation time, the JSON file J_3 also defines the fault-tolerance mechanism that is applied by AT-CARS. In order to compare

Table 6.11: Recovery probability of applications.

Applicaiton	Recovery Probability
loc1	$P_{loc1,recovery}(t_{loc1,failure}) = 0.3 + (1-0.3) \cdot e^{-(0.005 \cdot t)}$
loc2	$P_{loc2,recovery}(t_{loc2,failure}) = 0.4$
fus2	$P_{fus2,recovery}(t_{fus2,failure}) = \begin{cases} \frac{(0.2-1)}{7000} \cdot t + 1, & t \leq 7000 \\ 0.2, & t > 7000 \end{cases}$
fus3	$P_{fus3,recovery}(t_{fus3,failure}) = \begin{cases} 0.4, & t \leq 6000 \\ 0.2, & t > 6000 \end{cases}$
amm1	$P_{amm1,recovery}(t_{amm1,failure}) = 0.5$
amm2	$P_{amm2,recovery}(t_{amm2,failure}) = \begin{cases} \frac{(0.4-1)}{6000} \cdot t + 1, & t \leq 6000 \\ 0.4, & t > 6000 \end{cases}$
int_pred1	$P_{int\_pred1,recovery}(t_{int\_pred1,failure}) = \begin{cases} 0.6, & t \leq 6000 \\ 0.4, & t > 6000 \end{cases}$
int_pred2	$P_{int\_pred2,recovery}(t_{int\_pred2,failure}) = 0.3$
dr_plan2	$P_{dr\_plan2,recovery}(t_{dr\_plan2,failure}) = \begin{cases} \frac{(0.3-1)}{7000} \cdot t + 1, & t \leq 7000 \\ 0.3, & t > 7000 \end{cases}$
dr_plan3	$P_{dr\_plan3,recovery}(t_{dr\_plan3,failure}) = 0.3 + (1-0.3) \cdot e^{-(0.0003 \cdot t)}$
m_cont1	$P_{m\_cont1,recovery}(t_{m\_cont1,failure}) = \begin{cases} 0.4, & t \leq 4000 \\ 0.2, & t > 4000 \end{cases}$
m_cont2	$P_{m\_cont2,recovery}(t_{m\_cont2,failure}) = 0.4$
b_cont1	$P_{b\_cont1,recovery}(t_{b\_cont1,failure}) = \begin{cases} 0.6, & t \leq 4000 \\ 0.3, & t > 4000 \end{cases}$
b_cont2	$P_{b\_cont2,recovery}(t_{b\_cont2,failure3}) = \begin{cases} \frac{(0.2-1)}{5000} \cdot t + 1, & t \leq 5000 \\ 0.2, & t > 5000 \end{cases}$
tfc_opt1	$P_{tfc\_opt1,recovery}(t_{tfc\_opt1,failure}) = 0.3 + (1-0.3) \cdot e^{-(0.00009 \cdot t)}$
rd_mgmt1	$P_{rd\_mgmt1,recovery}(t_{rd\_mgmt1,failure}) = 0.5$
rd_vis1	$P_{rd\_vis1,recovery}(t_{rd\_vis1,failure}) = \begin{cases} \frac{(0.3-1)}{6000} \cdot t + 1, & t \leq 6000 \\ 0.3, & t > 6000 \end{cases}$
sh_ev_rec1	$P_{sh\_ev\_rec1,recovery}(t_{sh\_ev\_rec1,failure}) = 0.2 + (1-0.2) \cdot e^{-(0.005 \cdot t)}$

the different fault-tolerance mechanisms, i.e., m_1, m_2, and m_3, regarding their influence on the vehicle's reliability, we run three individual analyzes applying each mechanism once.

Furthermore, the JSON file J_3 defines which sensors need to be functional so that the vehicle is considered functional. We introduce so-called *sensor groups* to define how many sensors of a specific group need to be at least functional. For this example, we define that at least 10 of the 12 radars, cameras, and the lidar must be functional. Furthermore,

Table 6.12: Recovery Probability of computing nodes.

Computing Node	Recovery Probability
cn1	$P_{cn1,recovery}(t_{cn1,failure}) = \begin{cases} \frac{(0.6-1)}{9000} \cdot t + 1, & t \leq 9000 \\ 0.6, & t > 9000 \end{cases}$
cn2	$P_{cn2,recovery}(t_{cn2,failure}) = \begin{cases} 0.8, & t \leq 7000 \\ 0.6, & t > 7000 \end{cases}$
cn3	$P_{cn3,recovery}(t_{cn3,failure}) = \begin{cases} 0.8, & t \leq 6000 \\ 0.5, & t > 6000 \end{cases}$

Table 6.13: Recovery probability of sensors.

Sensor	Recovery Probability
radar_fl	$P_{radar\_fl,recovery}(t_{radar\_fl,failure}) = 0.5 + (1 - 0.5) \cdot e^{-(0.0002 \cdot t)}$
radar_fc	$P_{radar\_fc,recovery}(t_{radar\_fc,failure}) = \begin{cases} \frac{(0.4-1)}{5000} \cdot t + 1, & t \leq 5000 \\ 0.4, & t > 5000 \end{cases}$
radar_fr	$P_{radar\_fr,recovery}(t_{radar\_fr,failure}) = 0.5 + (1 - 0.5) \cdot e^{-(0.0002 \cdot t)}$
radar_rl	$P_{radar\_rl,recovery}(t_{radar\_rl,failure}) = 0.5 + (1 - 0.5) \cdot e^{-(0.0002 \cdot t)}$
radar_rr	$P_{radar\_rr,recovery}(t_{radar\_rr,failure}) = 0.5 + (1 - 0.5) \cdot e^{-(0.0002 \cdot t)}$
camera_fl	$P_{camera\_fl,recovery}(t_{camera\_fl,failure}) = \begin{cases} 0.6, & t \leq 3000 \\ 0.4, & t > 3000 \end{cases}$
camera_fc	$P_{camera\_fc,recovery}(t_{camera\_fc,failure}) = 0.4 + (1 - 0.4) \cdot e^{-(0.0005 \cdot t)}$
camera_fr	$P_{camera\_fr,recovery}(t_{camera\_fr,failure}) = \begin{cases} 0.6, & t \leq 3000 \\ 0.4, & t > 3000 \end{cases}$
camera_rl	$P_{camera\_rl,recovery}(t_{camera\_rl,failure}) = \begin{cases} 0.7, & t \leq 8000 \\ 0.5, & t > 8000 \end{cases}$
camera_rc	$P_{camera\_rc,recovery}(t_{camera\_rc,failure}) = \begin{cases} 0.8, & t \leq 7000 \\ 0.4, & t > 7000 \end{cases}$
camera_rr	$P_{camera\_rr,recovery}(t_{camera\_rr,failure}) = \begin{cases} 0.7, & t \leq 8000 \\ 0.5, & t > 8000 \end{cases}$
lidar_tc	$P_{lidar\_tc,recovery}(t_{lidar\_tc,failure}) = 0.3 + (1 - 0.3) \cdot e^{-(0.0007 \cdot t)}$
imu_tc	$P_{imu\_tc,recovery}(t_{imu\_tc,failure}) = 0.2 + (1 - 0.2) \cdot e^{-(0.0004 \cdot t)}$
imu_rc	$P_{imu\_rc,recovery}(t_{imu\_rc,failure}) = 0.5$

```
[
   {
      "computing_node": "cn1",
      "application_instances": [
         {
            "application_id": "loc2",
            "application_instance_id": "loc2_0",
            "operation_mode": "active"
         }, ...
      ]
   },
   {
      "computing_node": "cn2",
      "application_instances": [
         {
            "application_id": "loc2",
            "application_instance_id": "1",
            "operation_mode": "active-hot"
         }, ...
      ]
   },
   {
      "computing_node": "cn3",
      "application_instances": [
         {
            "application_id": "loc1",
            "application_instance_id": "2",
            "operation_mode": "active-hot"
         }, ...
      ]
   }
]
```

Figure 6.13: Extract of the of the JSON file J_2.

J_3 defines that at least one IMU must be operational. Note that the user defines the corresponding sensor groups for each sensor in the JSON file J_1.

Figure 6.15 illustrates the results of the conducted reliability analysis. We denote the resulting reliabilities of the three simulation runs as $R_{m_1}(t)$, $R_{m_2}(t)$, and $R_{m_3}(t)$, whereby the index corresponds to the fault-tolerance mechanism that was used in that particular run. The computer used to compute $R_{m_1}(t)$, $R_{m_2}(t)$, and $R_{m_3}(t)$ was equipped with an Apple M2 chip with 8 cores and 8 GB memory, whereby the execution time was between 8 and 10 minutes.

This analysis shows that the reliability is decreasing over time in all three conducted simulation runs, i.e., $R_{m_x}(t) \leq R_{m_x}(t+1)$ holds, for $x \in \{1, 2, 3\}$. Additionally, it

```json
{
   "number_of_iterations": 1000,
   "max_simulation_time": 10000,
   "fault_tolerance_mechanism": "m1",
   "sensor_groups" : [
      { "id": "camera_radar_lidar", "needed_sensors": 10},
      { "id": "imus", "needed_sensors": 1}
   ]
}
```

Figure 6.14: The JSON file J_3.

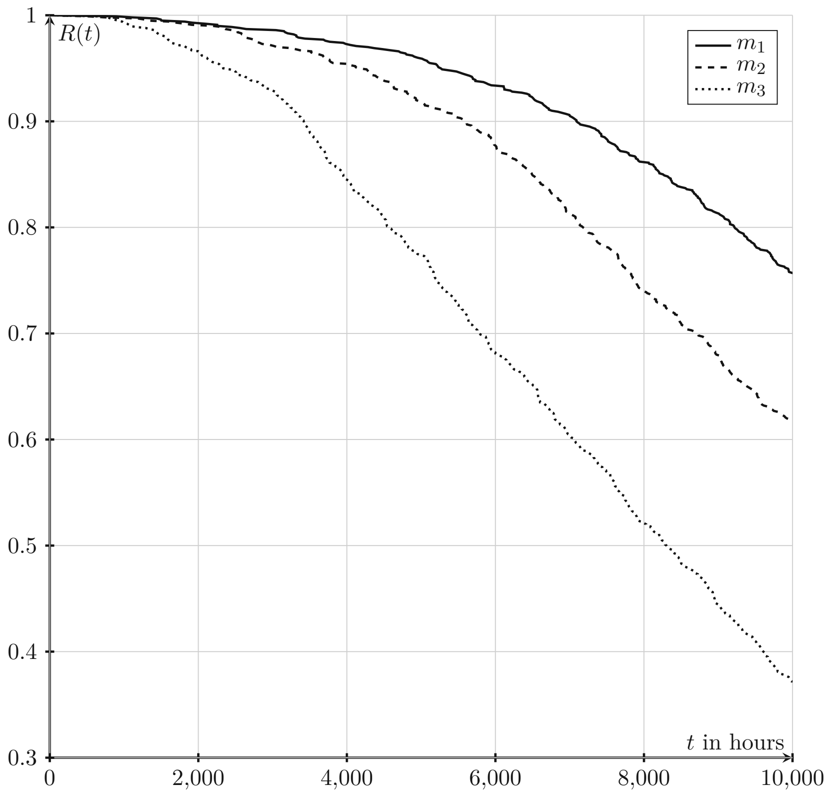


Figure 6.15: Reliability analysis of example vehicle in stage three using the fault-tolerance method m_1, m_2, and m_3.

demonstrates that

$$\forall t : R_{m_3}(t) \leq R_{m_2}(t) \leq R_{m_1}(t)$$

holds. Therefore, we can conclude that the fault-tolerance mechanism m_1, i.e., FDIRO together with HRR, maintains the reliability of the autonomous vehicle the best, followed by the fault-tolerance mechanism m_2, i.e., FDIRO. On the other hand, applying the fault-tolerance mechanism m_3, i.e., execution of only the fault detection and isolation step of FDIRO, causes, compared to m_1 and m_2, the lowest reliability.

6.5 Summary of the Use-Case Scenario

In this chapter, we evaluated APTUS on a use-case scenario. The evaluation showed that APTUS efficiently adapts the autonomous vehicle to changing conditions.

Note that the stages of the use-case scenario were designed to demonstrate a broad range of the features of APTUS while aiming for a realistic setup. A key assumption underlying the scenario is the system architecture of the autonomous vehicle, i.e., the functions, applications, computing nodes, and sensors, as well as their respective properties. To ensure realistic assumptions, we based, e.g., the functional architecture and the sensor set on state-of-the-art publications [13, 42]. Additionally, we assumed that the autonomous vehicle can determine the current context and detect faults that occur.

In order to fully incorporate APTUS in a concrete autonomous vehicle, several key aspects need to be addressed:

- the current-context provider, i.e., the component that is responsible for acquiring the current context and providing it to C-SAR, needs to be implemented;
- GCDB, the database used by C-SAR for determining software architecture requirements, needs to be populated with real-world data;
- ConfEx, the component that applies the configurations, needs to be implemented;
- monitors that detect faults in hardware and software components need to be implemented;
- for the implementation of HRR, the fault categories, the faults, and the corresponding recovery approaches need to be defined; and
- communication mechanisms that are required by D-DEG and CGM to communicate with other vehicles and backend systems need to be implemented.

To successfully deploy APTUS in an autonomous vehicle, its functionality and performance must be ensured. This necessitates the creation and execution of an exhaustive verification and validation process. Moreover, compatibility with applicable laws and standards must be guaranteed.

CHAPTER 7

Summary and Future Work

7.1 Summary

In this dissertation, we presented APTUS, a framework that aims to integrate self-managing capabilities into the system architecture of autonomous vehicles. Our framework is built upon the well-established three-layered architecture initially proposed by Gat [70]. The three layers of APTUS are the context layer, the reconfiguration layer, and the component layer, each responsible for specific self-managing properties.

In the first part of the dissertation, we introduced the general architecture of APTUS and provided background information on autonomous driving. Following this, we address the individual layers of APTUS.

The context layer, which is the top layer in our framework, focuses on implementing context-awareness by deriving software-architecture requirements from contextual observations. To accomplish this task, we introduced the two components C-SAR and D-DEG.

C-SAR, implemented in Python, uses answer-set programming (ASP) to identify software requirements suited to the current context. The presented implementation demonstrates that answer-set programming is a viable knowledge-representation language, as the programs are compact, comprehensible, and efficient in their performance.

The requirements determined by C-SAR are in the next step passed to D-DEG, which aims to degrade the resource needs of the software-architecture requirements based on the information received by other vehicles. We described two degradation modes, whereby one focuses on reducing the redundancy of applications, and the other seeks to lower the resource requirements by degrading active applications.

The resource-optimized software-architecture requirements determined by D-DEG serve as input for the reconfiguration layer. This layer is responsible for implementing the self-configuration property, the self-optimization property, and anticipativeness.

Self-configuration is achieved via two application-placement problem solvers: `logAP²S` and `linAP²S`. Those application-placement problem solvers are responsible for computing new configurations, whereby `logAP²S` can perform complex non-linear optimization functions, while `linAP²S`, which is based on prior work [116, 123], can quickly compute configurations after the fault of an application or hardware component.

To implement `logAP²S`, we employed ASP due to its ability to express non-linear optimization statements in a compact and comprehensible fashion. Furthermore, a performance evaluation showed that the solving time of `logAP²S` is only marginally longer than that of `linAP²S`.

The optimization statements employed by `logAP²S` are determined by C-po$_{\text{Aptus}}$, which implements the self-optimization property. C-po$_{\text{Aptus}}$ selects optimization statements based on the current context, whereby it aims to gradually optimize the safety and reliability properties of the configurations.

To implement anticipativeness in the configuration layer, we introduced the configuration-graph manager `CGM`. The idea of `CGM` is to reuse configurations by sharing configurations with other vehicles. Furthermore, `CGM` allows precomputing configurations in order to expand the configuration graph.

The bottom layer, the component layer, focuses on implementing the self-healing property. This layer incorporates Fdiro, a fault-tolerance approach that was also part of my Master's thesis [116, 122]. Fdiro, consists of four successive steps: fault detection, fault isolation, recovery, and optimization. These steps are designed to bring the system back to a safe and optimized state after encountering software or hardware faults. Furthermore, to address the recovery of failed hardware components, such as computing nodes or sensors, we introduced Hrr, which complements the capabilities of Fdiro.

The interplay of the individual layers was finally presented in a use-case scenario. In this example, we assume an autonomous vehicle that successively operates in three stages, whereby the vehicle experiences a different current context in each stage. The use-case scenario shows that Aptus computes different configurations for the individual stages.

In addition, the presented use-case scenario considers a fault of a computing node. We show that Aptus can handle this fault and transfer the system using multiple reconfiguration steps into a safe and optimized state.

Based on this use-case scenario, we also analyzed the reliability of a configuration in order to determine the impact of Fdiro and Hrr. To conduct this reliability analysis, we implemented `AT-CARS`, a Monte Carlo simulation-based tool that continuously injects faults into the vehicle's system architecture and analyzes their impacts. The conducted reliability analysis showed that Fdiro and Hrr have a beneficial effect on the reliability of the autonomous vehicles in the considered use case.

7.2 Future Work

In future work, we anticipate to enhance the individual layers of Aptus. In what follows, we elaborate on some of our planned activities.

The context layer currently lacks a mechanism to determine the current context the vehicle is experiencing. Recall that we assume that the current context is provided by the vehicle platform. In a future version of Aptus, we plan to include an approach for extracting contextual observations from various sensor data.

Furthermore, we plan to enhance C-sar. Particularly, we aim to extend the data model implemented by C-sar, for instance, to enable modeling different context-based applications and sensor degradation options. In addition, since the contextual observations that are used as input for C-sar can be considered as streams, we plan to investigate the use of ASP-based stream reasoning [24, 71].

The configuration layer currently includes two application-placement problem solvers, i.e., `linAP`^{2}`S`, which is based on integer linear programming, and `logAP`^{2}`S`, which employs ASP. In our future work, we plan to implement additional application-placement problem solvers that are, e.g., based on approaches as discussed in Subsection 4.1.2. These solvers can be integrated in Aptus as new options or they can be used alongside the existing ones, i.e., as redundant instances, to further improve the reliability of the framework.

Moreover, we aim to implement a test environment for `CGM`. This test environment shall enable the comparison of different precomputation strategies. Note that it is also conceivable to integrate this test environment into `AT-CARS`.

The current version of Aptus does not implement the self-protection property. Therefore, in a future version of Aptus, we aim to introduce measures in order to fulfill this property. Our idea is to extend Fdiro to detect security attacks and to consequently isolate the attack. To validate the security extension of Fdiro we plan to adapt `AT-CARS`. A preview of this idea was already presented by Horeis et al. [98].

In this dissertation we used `AT-CARS` to determine the reliability of concrete configurations. However, `AT-CARS` could be also used to determine the applications and computing nodes a vehicle shall be equipped with in order to maximize the reliability, safety, and security. Furthermore, `AT-CARS` can help to determine which failure probabilities of the individual components are required to achieve a certain level of reliability, safety, and security. In order to perform such reverse engineering activities it is, however, necessary to improve to solving time of `AT-CARS`. Rinaldo et al. [185, 186] presented the idea of a hybrid modeling approach which could serve as a basis for enhancing the performance of `AT-CARS`.

APPENDIX A

Implementation Details

A.1 Implementation of the Context-Reasoning Component **CRC**

This section presents the implementation of CRC. Recall that, as specified in Subsection 3.3.3, CRC is implemented by the answer-set program

$$P_{\text{CRC}} = P_{\text{CRC\_1}} \cup P_{\text{CRC\_2}} \cup P_{\text{CRC\_3}} \cup P_{\text{CRC\_4}},$$

where

- $P_{\text{CRC\_1}}$ is responsible for choosing the optimization function that is requested by the current context,
- $P_{\text{CRC\_2}}$ is responsible for choosing the tasks that are requested by the current context,
- $P_{\text{CRC\_3}}$ takes care of selecting for each selected task one active application, and
- $P_{\text{CRC\_4}}$ identifies the feasible diverse applications for each task.

The program P_{CRC} is given as follows:

```
% P_CRC_1: Optimization Function Selection
selected_optimization_function(OPT_FUNC, PRIO) :-
   optimization_function(OPT_FUNC), current_context(CUR_CX),
   def_optimization_function(CUR_CX, OPT_FUNC, PRIO).

% P_CRC_2: Functions Selection
selected_function(FUNC, PRIO) :-
   function(FUNC), current_context(CUR_CX),
   req_function(CUR_CX,FUNC,PRIO),
   PRIO = #max{P: req_function(C, FUNC, P),
               current_context(C)}.

% P_CRC_3: Active Application Selection
specialization_level(APP, SET, SPEC_L) :-
   env_cx_implementation(APP, SET, _),
   SPEC_L = #count{V: env_cx_set_member(V, SET)},
   MATCHES = #count{C: environmental_context_set(SET),
                    env_cx_set_member(C, SET),
                    current_context(C)},
   SPEC_L = MATCHES.

specialization_level(APP, CUR_CX, 1) :-
   current_context(CUR_CX),
   env_cx_implementation(APP, CUR_CX, _),
   environmental_context_value(CUR_CX, _).

max_special_env_cx(APP, ENV_CX) :-
   specialization_level(APP, ENV_CX, SPEC_L),
   SPEC_L = #max{SV:specialization_level(APP, _, SV)}.

cx_application_rating(APP, RATING) :-
   application(APP, FUNC),
   max_special_env_cx(APP, ENV_CX),
   env_cx_implementation(APP, ENV_CX, RATING),
   RATING = #min{R: max_special_env_cx(APP, E_C),
                 env_cx_implementation(APP, E_C, R)}.

feasible_rated_application(APP, FUNC) :-
   application(APP, FUNC),
   max_special_env_cx(APP, ENV_CX),
   specialization_level(APP, ENV_CX, SPEC_L),
   SPEC_L = #max{S_L: application(A, FUNC),
                   max_special_env_cx(A,E_C),
                   specialization_level(A, E_C, S_L)}.

feasible_non_rated_application(APP) :-
   application(APP, FUNC), selected_function(FUNC, _),
   not env_cx_implementation(APP, _, _),
   not feasible_rated_application(_, FUNC).

max_rated_application(APP) :-
   selected_function(FUNC, _),
   application(APP,FUNC),
   feasible_rated_application(APP, FUNC),
   cx_application_rating(APP, RATING),
   RATING = #max{R: feasible_rated_application(A, FUNC),
                 cx_application_rating(A, R)}.

active_application(APP,0) :-
   max_rated_application(APP),
   application(APP,FUNC),
   APP = #min{A: max_rated_application(A),
              application(A, FUNC)}.

active_application(APP,0) :-
   selected_function(FUNC, _),
   application(APP,FUNC),
   feasible_non_rated_application(APP),
   APP = #min{A: feasible_non_rated_application(A),
              application(A, FUNC)}.

:- selected_function(FUNC, _),
   #count{A: active_application(A, _),
          application(A, FUNC)} != 1.

% P_CRC_4: Diverse Application Determination
feasible_diverse_application(APP, SPEC_L, RATING) :-
   application(APP, FUNC),
   selected_function(FUNC, _),
   max_special_env_cx(APP, ENV_CX),
   cx_application_rating(APP, RATING),
   specialization_level(APP, ENV_CX, SPEC_L),
   SPEC_L = #max{S_L: specialization_level(APP, _, S_L)},
   not active_application(APP, _).

feasible_diverse_application(APP, 0, 0):-
   application(APP, FUNC),
   selected_function(FUNC, _),
   not env_cx_implementation(APP, _, _),
   not active_application(APP, _).
```

A.2 Implementation of the Linear Application-Placement Problem Solver

This section focuses on the implementation of `linAP`^{2}`S`. As mentioned in Section 4.4 `linAP`^{2}`S` is a Python program that uses OR-Tools to express and solve the application-placement problem. The function `compute_configuration` is considered as the core of this component. The source code of this function is presented in the following listing and discussed in the remainder of this section:

```python
def compute_configuration(current_configuration, functions_apps,
                          separation, performance_capacities,
                          memory_capacities, performance_demands,
                          memory_demands, software_restrictions):

   solver = ortools.linear_solver.pywraplp.Solver('Solver',
                     pywraplp.Solver.BOP_INTEGER_PROGRAMMING)

   number_of_functions = len(functions_apps)
   number_of_applications = len(current_configuration)
```

```
number_of_computing_nodes = len(current_configuration[0])

# Creates the variables that shall be determined by the solver.
new_placement = [[solver.BoolVar("i%i n%i" % (i, n))
                  for n in range(number_of_computing_nodes)]
                 for i in range(number_of_applications)]

new_placement_trans = [[new_placement[i][j]
                        for i in range(number_of_applications)]
                       for j in range(number_of_computing_nodes)]

separation_helper = [[solver.BoolVar("a%i n%i" % (a, n))
                      for n in range(number_of_computing_nodes)]
                     for a in range(number_of_functions)]

# Condition 1
for i in range(number_of_applications):
   solver.Add(solver.Sum(new_placement[i]) == 1)

# Condition 2
for n in range(number_of_computing_nodes):
   solver.Add(
      solver.Sum([new_placement_trans[n][i] * memory_demands[i]
                  for i in range(number_of_applications)]) <=
      memory_capacities[n])

# Condition 3
for n in range(number_of_computing_nodes):
   solver.Add(
      solver.Sum([new_placement_trans[n][i] * performance_demands[i]
                  for i in range(number_of_applications)]) <=
      performance_capacities[n])

# Condition 4
for i in range(number_of_applications):
   for n in range(number_of_computing_nodes):
      if software_restrictions[i][n] == 0:
         solver.Add(new_placement[i][n] == 0)

# Condition 5
for a in range(number_of_functions):
   for n in range(number_of_computing_nodes):

      for i in functions_apps[a]:
         solver.Add(new_placement[i][n] <=
            separation_helper[a][n])

      solver.Add(
         solver.Sum([
            new_placement[i][n]
            for i in functions_apps[a]
         ]) >= separation_helper[a][n]
      )

      solver.Add(
         solver.Sum(separation_helper[a]) >=
                     separation[a]
      )

   # Problem Optimization
   solver.Maximize(solver.Sum([
      new_placement[i][n] * current_configuration[i][n]
      for i in range(number_of_applications)
      for n in range(number_of_computing_nodes)]))

   result_status = solver.Solve()

   if result_status == 2:
      return None # No new configuration found

   new_configuration = [[new_placement[i][n].new_configuration()
                  for n in range(number_of_computing_nodes)]
                  for i in range(number_of_applications)]

   return new_configuration
```

The `compute_configuration` function requires as input the eight parameters:

- `current_configuration`,
- `functions_apps`,
- `separation`,
- `memory_capacities`,
- `performance_capacities`,
- `memory_demands`,
- `performance_demands`, and
- `software_restrictions`.

The `current_configuration` parameter is an $|A| \times |N|$ matrix, whereby $|A|$ is the number of applications that shall be executed and $|N|$ is the number of computing nodes. Each element of the matrix specifies whether an application instance $a \in A$ is executed by a computing node $n \in N$, whereby a value of 1 indicates that computing node n executes a, while a value of 0 indicates that it does not.

The `functions_apps` parameter is a list of lists, whereby each list represents a selected function. The elements of the individual lists are the so-called *applications indices* of the applications that are selected to execute the specific function. The application index of an application defines its position in the `current_configuration` parameter and the `new_configuration` variable. Note that the application indices are defined in the procedure for setting up the application-placement problem.

The `separation` parameter is a list of length $|F|$, whereby $|F|$ is the number of selected functions. The elements of this list define the desired separation of the individual functions.

The provided memory and performance of the computing nodes are defined by the `memory_capacities` and the `performance_capacities` parameter, respectively, which are lists of length $|N|$. Corresponding to those parameters, the memory and performance demands of the applications are given by the `memory_demands` and the `performance_demands` parameter, which are lists of length $|A|$.

The `software_restrictions` parameter is a $|A| \times |N|$ whereby the elements indicate whether a computing node provides all the required supporting software for an application. A value of 1 indicates that computing node n provides all the required supporting software of a, while a value of 0 indicates that it does not.

After defining all required input variables, the `compute_configuration` function declares in lines 6 to 11 of the code the solver responsible for finding a valid solution as well as some helper variables.

Next, in lines 14 to 24, we define the variables that shall be determined by the solver. The `new_placement` parameter is of the same dimension as the `current_configuration` parameter and declares the variables that, after solving the application problem, correspond to the determined solution. We also define `current_configuration_trans`, which is the transpose of the `current_configuration` parameter.

Besides these variables, the solver also has to determine the variables defined by the `separation_helper` parameter, whereby those variables indicate whether any instance of an application is executed by a specific computing node. Due to those helper variables, condition (C_5) of the application-placement problem can be expressed linearly.

Note that the sum of the number of applications and the number of functions multiplied by the number of computing nodes corresponds to the total number of variables t that the solver has to determine:

$$t = |N| \cdot (|A| + |F|).$$

Since all these variables are Boolean, the size of the solution space is 2^t.

To restrict the solution space, we incorporate the conditions of the application-placement problem in lines 27 to 68. Note that we only use linear operations to define those constraints.

Next, in lines 71 to 74, we specify the optimization goal. The goal implemented by `linAP`^{2}`S` is to find a configuration that differs the least from the current configuration.

Finally, in lines 76 to 85, we instruct the solver to solve the specified application-placement problem and return the solution if one exists.

A.3 Implementation of the ASP Application-Placement Problem Solver

This section presents the implementation of `logAP`^{2}`S` as described in Section 4.5. In particular, we outline the version of `logAP`^{2}`S` used to compare its performance with `linAP`^{2}`S`. The code for this version is illustrated in the following listing:

```
% Condition 1
{assignment(APP, R_INST, CN): computing_node(CN)} = 1 :-
   active_application(APP, R_INST).
{assignment(APP, R_INST, CN): computing_node(CN)} = 1 :-
   active_low_application(APP, R_INST).
{assignment(APP, R_INST, CN): computing_node(CN)} = 1 :-
   active_hot_application(APP, R_INST).

% Condition 2
:- computing_node(CN), memory(CN, MEM),
   #sum{M, APP, R_INST: memory(APP, M),
                    application(APP, R_INST),
   assignment(APP, _, CN)} > MEM.

% Condition 3
:- computing_node(CN), performance(CN, PERF),
   #sum{C, APP, R_INST: performance(APP, C),
                    application(APP, R_INST),
   assignment(APP, _, CN)} > PERF.

% Condition 4
:- assignment(APP, _, CN), req_supporting_software(APP, SW),
   not provided_supporting_software(CN, SW).

% Condition 5
:- active_application(A_APP, _), application(A_APP, FUNC),
   separation(A_APP, SEP), #count{CN: application(APP, FUNC),
   assignment(APP, _, CN)} < SEP.

:- active_low_application(AL_APP, _), application(AL_APP, FUNC),
   separation(AL_APP, SEP), #count{CN: application(APP, FUNC),
   assignment(APP, _, CN)} < SEP.

% Problem Optimization
displacement_count(CNT) :-
   CNT = #count{APP, R_INST: assignment(APP, R_INST, CN),
                      current_assignment(APP_CUR,
                                     R_INST_CUR,
                                     CN_CUR),
                      APP = APP_CUR,
                      R_INST = R_INST_CUR,
                        CN != CN_CUR}.

#minimize{CNT:displacement_count(CNT)}.
```

A.4 Implementation of **AT-CARS**

This section illustrates the implementation of AT-CARS. As mentioned in Section 5.5, AT-CARS is implemented in Matlab. However, for the sake of clarity and easier presentation, we use pseudo-code instead of the full MATLAB code to illustrate the core ideas of AT-CARS. The following listing presents the pseudo-code for AT-CARS, written using Python-like syntax:

```
def main(J_1, J_2, J_3):
   # Parse input parameters from JSON files
   computing_nodes = get_computing_nodes(J_1)
   sensors = get_sensors(J_1)
   applications = get_applications(J_1)
   functions = get_functions(J_1)
   initial_config = get_initial_config(J_2)
   sim_params = get_simulation_parameters(J_3)

   components = computing_nodes + sensors + applications
   iterations = []

   # Start simulation
   for i in range(sim_params.number_of_iterations):
      iteration = new Iteration()
      current_simulation_time = 0

      # Apply initial config
      current_config = initial_config

      # Determine fault times of components
      component_faults = determine_component_faults(components,
          current_simulation_time)

      # Start iteration
      while current_simulation_time <= sim_params.max_sim_time:

         # Select the fault that occurs chronologically first
         next_fault = get_next_fault(component_faults)

         # Check if the fault time exceeds the maximum simulation time
         if next_fault.component.fault_time > sim_params.max_sim_time:
            iteration.fault_time = None # The iteration did not fail
            iterations.append(iteration)
            break # End the current iteration

         current_simulation_time = next_fault.component.fault_time
```

```

            # Inject the selected fault into the corresponding component
            next_fault.component.failed = True

            # Execute fault-tolerance method based on selected mode
            current_config = run_fault_tolerance_mechanism(current_config,
               next_fault, sim_params.tolerance_mechanism, functions)

            # Check if the system failed
            if not (all_HIGH_prio_functions_executed(current_config,
               functions) and all_sensor_group_reqs_fulfilled(sensors,
               sim_params.sensor_groups)):
               iteration.fault_time = next_fault.component.fault_time
               iterations.append(iteration)
               break

            # Determine fault times of new applications
            new_applications = current_config.get_new_applications()
            component_faults += determine_component_faults(new_applications,
               current_simulation_time)

            # Update fault times recovered components
            rec_comps = get_recovered_components(computing_nodes, sensors)
            component_faults.update_recovered_components(rec_comps,
               current_simulation_time)


   # Determine the reliability and visualize it
   plot_reliability_over_time(iterations, sim_params.number_of_iterations,
      sim_params.max_sim_time)


def determine_component_faults(components, time):
   component_faults = []
   for c in components:
      cf = new ComponentFault()
      cf.component = c
      cf.fault_time = time + c.fault_distribution.draw_sample()
      component_faults.append(cf)
   return component_faults


def get_next_fault(component_faults):
   next_fault = component_faults[0]

   for cf in component_faults:
      if cf.fault_time < next_fault.fault_time:
         next_fault = cf

   return next_fault


def run_fault_tolerance_mechanism(config, fault, ft_mechanism, functions):
    if ft_mechanism == "m1":
        new_config = execute_FDIRO(config, fault)
        execute_HRR(fault)
        return new_config
    elif ft_mechanism == "m2":
        return execute_FDIRO(config, fault)
    elif ft_mechanism == "m3":
        return execute_fault_isolation(config, fault, functions)

def execute_HRR(fault):
    if fault.component.type == "computing_node" or
        fault.component.type == "sensor":

        fault.component.recovery_possible =
            recovery_decision(fault.component)
        fault.component.failed = not fault.component.recovery_possible


def recovery_decision(component):
    r = random.uniform(0, 1) # Create random number between 0 and 1

    # Get the recovery probability at the time of the fault,i.e., at
    # time fault.component.fault_time
    rec_prop = fault.component.get_recovery_probability()

    if r <= rec_prop:
        return True # Component can be recovered

    return False

def execute_fdiro(config, fault, functions):
    # Step 1: Fault Detection
    # In this simulation, the fault detection is trivial
    # since the failed components are marked.

    # Step 2: Fault Isolation
    new_config = execute_fault_isolation(config, fault, functions)
    if new_config is None:
        return # Fault isolation step was not successful

    # Step 3: Redundancy Recovery
    failed_applications = new_config.get_failed_applications()

    for a in failed_applications:
        a.recovery_possible = recovery_decision(a)

    new_config = linAPPS(new_config, failed_applications)

    # Step 4: Configuration Optimization
    new_config = cpo(new_config)

    new_config = logAPPs(optimization_functions, new_config)
```

```

   return new_config


def execute_fault_isolation(config, fault, functions):
   apps = identify_affected_applications(config, fault)
   for a in apps:
      a.set_operation_mode("isolated")

      if a.operation_mode == "active" or a.operation_mode == "active-low":
         # Switch to a redundant instance if available
         redundant_instance = config.get_redundant_instance(a)

         if redundant_instance is not None:
            redundant_instance.set_operation_mode("active")
         else:
            # No redundant instance available. Check that the minimum
            # safety requirement according to Lambda_prio is fulfilled
            if not all_HIGH_prio_functions_executed(config, functions):
               return None

   return config


def identify_affected_applications(config, fault):
   apps = []

   # Check if applications failed due to the fault of a computing node
   if fault.component.type == "computing_node":
      apps = config.get_apps_assigned_to_computing_node(fault.component)
      for a in apps:
         a.failed = True
   elif fault.component.type == "application":
      apps = [fault.component]

   return apps


def all_HIGH_prio_functions_executed(config, functions):
   # Check if for functions of priority HIGH at least one application
   # that implements this function is executed
   for f in functions.get_HIGH_prio_functions():
      function_executed = False
      for a in config.get_executed_applications():
         if a.function == f:
            function_executed = True
            break
      if not function_executed:
         return False

   return True

```

```
def all_sensor_group_reqs_fulfilled(sensors, sensor_groups):
   # Check for sensor group if the required number of functional
   # sensors are fulfilled
   for sg in sensor_groups:
      functional_sensors_count = 0
      for s in sensors:
         if s.sensor_group == sg.id and not s.failed:
            functional_sensors_count += 1

      if functional_sensors_count < sg.needed_sensors:
         return False

   return True

def plot_reliability_over_time(iterations, num_of_iterations, max_time):
   times = iterations.get_all_fault_times()
   times = sorted(times) # Sort times ascending order
   functional_iteration_count = num_of_iterations

   reliability = 1
   plot.add_data_point(0, 1) # At time 0 the reliability is set to 1

   for t in times:
      if t is not None:
         reliability = functional_iteration_count / num_of_iterations
         plot.add_data_point(t, reliability)
         functional_iteration_count -= 1

   plot.add_data_point(max_time, reliability)
```

Bibliography

[1] Mohamed Abdel-Basset, Laila Abdel-Fatah, and Arun Kumar Sangaiah. Metaheuristic Algorithms: A Comprehensive Review. In *Computational Intelligence for Multimedia Big Data on the Cloud with Engineering Applications*, Intelligent Data-Centric Systems, pages 185–231. Academic Press, 2018.

[2] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a Better Understanding of Context and Context-Awareness. In *Proceedings of the First International Symposium of Handheld and Ubiquitous Computing (HUC'99)*, pages 304–307. Springer, 1999.

[3] Jean-Raymond Abrial. Data Semantics. In *Proceedings of the IFIP Working Conference on Data Base Management*, pages 1–60. North-Holland, 1974.

[4] Aeronautical Radio Incorporated. *ARINC Specification 653 P1-2*. ARINC, 2005.

[5] M. Nadeem Ahangar, Qasim Z. Ahmed, Fahd A. Khan, and Maryam Hafeez. A Survey of Autonomous Vehicles: Enabling Communication Technologies and Challenges. *Sensors*, 21(3):1–33, 2021.

[6] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 01(1):11–33, 2004.

[7] Mansoor Alicherry and T.V. Lakshman. Network Aware Resource Allocation in Distributed Clouds. In *Proceedings of the 2012 IEEE International Conference on Computer Communications (INFOCOM 2012)*, pages 963–971. IEEE, 2012.

[8] Mario Alviano and Wolfgang Faber. Aggregates in Answer Set Programming. *KI - Künstliche Intelligenz*, 32:119–124, 2018.

[9] Pedro Andreo. Monte Carlo Simulations in Radiotherapy Dosimetry. *Radiation Oncology*, 13(1), 2018.

[10] Pasquale Antonante, Heath G. Nilsen, and Luca Carlone. Monitoring of Perception Systems: Deterministic, Probabilistic, and Learning-based Fault Detection and Identification. *Artificial Intelligence*, 325, 2023.

[11] Shunsuke Aoki, Takamasa Higuchi, and Onur Altintas. Cooperative Perception with Deep Reinforcement Learning for Connected Vehicles. In *Proceedings of the 2020 IEEE Intelligent Vehicles Symposium (IV 2020)*, pages 328–334. IEEE, 2020.

[12] Krzysztof R Apt and Mark Wallace. *Constraint Logic Programming using ECLiPSe*. Cambridge University Press, 2006.

[13] APTIV, Audi, Baidu, BMW, Continental, Daimler, FCA, HERE, Infineon, Intel, and Volkswagen. Safety First for Automated Driving, 2019. White paper.

[14] Jakub Arm, Zdenek Bradac, and Radek Štohl. Increasing Safety and Reliability of Roll-back and Roll-forward Lockstep Technique for Use in Real-time Systems. *IFAC-PapersOnLine*, 49(25):413–418, 2016.

[15] Arno Meyna and Bernhard Pauli. *Zuverlässigkeitstechnik - Quantitative Bewertungsverfahren*. Hanser, second edition, 2010.

[16] 5G Infrastructure Association. 5G Automotive Vision, 2015. White paper.

[17] Jyotika Athavale, Andrea Baldovin, and Michael Paulitsch. Trends and Functional Safety Certification Strategies for Advanced Railway Automation Systems. In *Proceedings of the IEEE International Reliability Physics Symposium (IRPS 2020)*. IEEE, 2020.

[18] Algirdas Avizienis and John P. J. Kelly. Fault Tolerance by Design Diversity: Concepts and Experiments. *Computer*, 17(8):67–80, 1984.

[19] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[20] Chitta Baral, Juraj Dzifcak, and Tran Cao Son. Using Answer Set Programming and Lambda Calculus to Characterize Natural Language Sentences with Normatives and Exceptions. In *Proceedings of the 23rd Conference on Artificial Intelligence (AAAI 2008)*, pages 818–823. AAAI Press, 2008.

[21] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 1267–1329. IOS Press, 2009. Second edition, 2021.

[22] Matthew Barth, Kanok Boriboonsomsin, and Guoyuan Wu. Vehicle Automation and Its Potential Impacts on Energy and Emissions. In *Road Vehicle Automation*, pages 103–112. Springer, 2014.

[23] H. Basher and J. S. Neal. Autonomous Control of Nuclear Power Plants. Report, Nuclear Science and Technology Division, 2003.

[24] Harald Beck, Thomas Eiter, and Christian Folie. Ticker: A System for Incremental ASP-based Stream Reasoning. *Theory and Practice of Logic Programming*, 17(5–6):744–763, 2017.

[25] Sagar Behere and Martin Törngren. A Functional Reference Architecture for Autonomous Driving. *Information and Software Technology*, 73(C):136–150, 2016.

[26] Pietro Belotti, Christian Kirches, Sven Leyffer, Jeff Linderoth, James Luedtke, and Ashutosh Mahajan. Mixed-Integer Nonlinear Optimization. *Acta Numerica*, 22:1–131, 2013.

[27] F. Ben Jemaa, G. Pujolle, and M. Pariente. QoS-Aware VNF Placement Optimization in Edge-Central Carrier Cloud Architecture. In *Proceedings of the 2016 IEEE Global Communications Conference (GLOBECOM 2016)*. IEEE, 2016.

[28] R.C. Berkan, Belle R. Upadhyaya, Lefteri H. Tsoukalas, Roger A. Kisner, and R.L. Bywater. Advanced Automation Concepts for Large-Scale Systems. *IEEE Control Systems Magazine*, 11(6):4–12, 1991.

[29] Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan, and Daniele Riboni. A Survey of Context Modelling and Reasoning Techniques. *Pervasive and Mobile Computing*, 6(2):161–180, 2010.

[30] Joseph John Bevelacqua. *Basic Health Physics: Problems and Solutions*, volume 1. John Wiley & Sons, 2010.

[31] Joseph John Bevelacqua. Applicability of Health Physics Lessons Learned from the Three Mile Island Unit 2 Accident to the Fukushima Daiichi Accident. *Journal of Environmental Radioactivity*, 105:6–10, 2012.

[32] Alexander Beyer, Gerhard Grunwald, Martin Heumos, Manfred Schedl, Ralph Bayer, Wieland Bertleff, Bernhard Brunner, Robert Burger, Jörg Butterfaß, Robin Gruber, Thomas Gumpert, Franz Hacker, Erich Krämer, Maximilian Maier, Sascha Moser, Josef Reill, Máximo Alejandro Roa Garzon, Hans-Jürgen Sedlmayr, Nikolaus Seitz, Martin Stelzer, Andreas Stemmer, Gabriel Tubio Manteiga, Tilman Wimmer, Markus Grebenstein, Christian Ott, and Alin Olimpiu Albu-Schäffer. CAESAR: Space Robotics Technology for Assembly, Maintenance, and Repair. In *Proceedings of the 69th International Astronautical Congress (IAC 2018)*, 2018.

[33] Pierre Bieber, Eric Noulard, Claire Pagetti, Thierry Planche, and Francois Vialard. Preliminary Design of Future Reconfigurable IMA Platforms. *AMC Special Interest Group on Embedded Systems (SIGBED) Review*, 6(3), 2009.

[34] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336. IOS Press, 2009. Second edition, 2021.

[35] Alessandro Birolini. *Reliability Engineering: Theory and Practice*. Springer, 2013.

[36] Nikolaj Bjørner, Anh Dung Phan, and Lars Fleckenstein. νZ - An Optimizing SMT Solver. In *Proceedings of the 21st International ConferenceLecture on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*, pages 194–199. Springer, 2015.

[37] Hugh Blair-Smith. Space Shuttle Fault Tolerance: Analog and Digital Teamwork. In *Proceedings of the 28th IEEE/AIAA Digital Avionics Systems Conference (DASC 2009)*, pages 6.B.1–1–6.B.1–11. IEEE, 2009.

[38] Hermann Boehnhardt, Jean Pierre Bibring, Istvan Apathy, Hans Ulrich Auster, Amalia Ercoli Finzi, Fred Goesmann, Göstar Klingelhöfer, Martin Knapmeyer, Wlodek Kofman, Harald Krüger, Stefano Mottola, Walter Schmidt, Klaus Seidensticker, Tilman Spohn, and Ian Wright. The Philae Lander Mission and Science Overview. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2097), 2017.

[39] Martin Böhm and Alexander Frötscher. Data-Flow and Processing for Mobile In-Vehicle Weather Information Services COOPERS Service Chain for Co-operative Traffic Management. In *Proceedings of the 69th IEEE Vehicular Technology Conference (VTC Spring 2009)*, 2009.

[40] Rudolf Brockhaus, Wolfgang Alles, and Robert Luckner. *Flugregelung*. Springer, 2011.

[41] Peter J. Brown, John D. Bovey, and Xian Chen. Context-Aware Applications: From the Laboratory to the Marketplace. *IEEE Personal Communications*, 4(5):58–64, 1997.

[42] Holger Caesar, Varun Bankiti, Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuScenes: A Multimodal Dataset for Autonomous Driving. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2020)*, pages 11618–11628, 2020.

[43] Matthieu Carré, Ernesto Exposito, and Javier Ibañez-Guzmán. Framework for Safety in Autonomous Vehicles. In *Proceedings of the 2019 Conférence Francophone sur les Architectures Logicielles (CAL 2019)*. Éditions RNTI, 2019.

[44] Guanling Chen and David Kotz. A Survey of Context-Aware Mobile Computing Research. *Computer Science Technical Report*, 381, 2000.

[45] Harry Chen, Filip Perich, Tim Finin, and Anupam Joshi. SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications. In *Proceeding of the First Annual International Conference on Mobile and Ubiquitous Systems (MobiQuitous 2004)*, pages 258–267. IEEE, 2004.

[46] Peter Pin-Shan Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[47] Tao Chen, Rami Bahsoon, and Xin Yao. A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems. *ACM Computing Surveys*, 51(3):1–40, 2018.

[48] Antonio Chialastri. Automation in Aviation. In *Automation*. IntechOpen, 2012.

[49] William F. Clocksin and Christopher S. Mellish. *Programming in PROLOG*. Springer, fifth edition, 2003.

[50] Ian Colwell, Buu Phan, Shahwar Saleem, Rick Salay, and Krzysztof Czarnecki. An Automated Vehicle Safety Concept Based on Runtime Restriction of the Operational Design Domain. In *Proceedings of the 2018 IEEE Intelligent Vehicles Symposium (IV 2018)*, pages 1910–1917. IEEE, 2018.

[51] United States Nuclear Regulatory Commission. *NRC Response to Lessons Learned from Fukushima*. U.S. NRC, 2018.

[52] Deshan Cooray, Sam Malek, Roshanak Roshandel, and David Kilgore. RESISTing Reliability Degradation Through Proactive Reconfiguration. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, pages 83–92. AMC, 2010.

[53] Douglas Crockford. *RFC 4627: The Application/JSON Media Type for JavaScript Object Notation (JSON)*. IETF, 2006.

[54] Clemens Dannheim, Markus Mader, C. Icking, Jan Loewenau, and Kay Massow. A Novel Approach for the Enhancement of Cooperative ACC by Deriving Real Time Weather Information. In *Proceedings of the 16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*, pages 2207–2211. IEEE, 2013.

[55] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Göschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Dennis B. Smith, João Pedro Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In *Software Engineering for Self-Adaptive Systems II*. Springer, 2013.

[56] Miguel Ángel de Miguel, Francisco Miguel Moreno, Fernando García, Jose María Armingol, and Rodrigo Encinar Martin. Autonomous Vehicle Architecture for High

Automation. In *Proceedings of the International Conference on Computer Aided Systems Theory (EUROCAST 2019)*, pages 145–152. Springer, 2020.

[57] Salvatore Distefano and Antonio Puliafito. Dynamic Reliability Block Diagrams: Overview of a Methodology. In *Proceedings of the European Safety and Reliability Conference 2007 (ESREL 2007)*, pages 1059–1063. Taylor & Francis, 2007.

[58] Joanne Bechta Dugan, Salvatore J. Bavuso, and Mark A. Boyd. Dynamic Fault-Tree Models for Fault-Tolerant Computer Systems. *IEEE Transactions on Reliability*, 41(3):363–377, 1992.

[59] Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining Answer Set Programming with Description Logics for the Semantic Web. *Artificial Intelligence*, 172(12–13):1495–1539, 2008.

[60] Esra Erdem, Erdi Aker, and Volkan Patoglu. Answer Set Programming for Collaborative Housekeeping Robotics: Representation, Reasoning, and Execution. *Intelligent Service Robotics*, 5(4):275–291, 2012.

[61] European Cooperation for Space Standardization. *ECSS-Q-ST-40C Rev.1*. ECSS, 2017.

[62] Daniel J. Fagnant and Kara Kockelman. Preparing a Nation for Autonomous Vehicles: Opportunities, Barriers and Policy Recommendations. *Transportation Research Part A: Policy and Practice*, 77:167 – 181, 2015.

[63] Jamil Fayyad, Mohammad A. Jaradat, Dominique Gruyer, and Homayoun Najjaran. Deep Learning Sensor Fusion for Autonomous Vehicle Perception and Localization: A Review. *Sensors*, 20(15), 2020.

[64] Pedro Fernandes and Urbano Nunes. Platooning With IVC-Enabled Autonomous Vehicles: Strategies to Mitigate Communication Delays, Improve Safety and Traffic Flow. *IEEE Transactions on Intelligent Transportation Systems*, 13(1):91–106, 2012.

[65] Stefan Figedy and Gabriel Oksa. Modern Methods of Signal Processing in the Loose Part Monitoring System. *Progress in Nuclear Energy*, 46(3–4):253–267, 2005.

[66] Holger Flühr. *Avionik und Flugsicherungstechnik*. Springer, 2012.

[67] Jeremy Frank, David Iverson, Christopher Knight, Sriram Narasimhan, Keith Swanson, Michael Scott, May Windrem, Kara Pohlkamp, Jeffery Mauldin, Kerry McGuire, and Haifa Moses. Demonstrating Autonomous Mission Operations Onboard the International Space Station. In *Proceedings of the 2015 AIAA SPACE Conference and Exposition*. IAAA, 2015.

[68] Simone Fuchs, Stefan Rass, Bernhard Lamprecht, and Kyandoghere Kyamakya. Context-Awareness and Collaborative Driving for Intelligent Vehicles and Smart Roads. In *Proceedings of the 1st International Workshop on ITS for Ubiquitous Roads (UBIROADS'2007)*, 2007.

[69] Alan G. Ganek and Thomas A. Corbi. The Dawning of the Autonomic Computing Era. *IBM Systems Journal*, 42(1):5–18, 2003.

[70] Erann Gat. Three-Layer Architectures. In *Artificial Intelligence and Mobile Robots*, page 195–210. AAAI Press, 1998.

[71] Martin Gebser, Torsten Grote, Roland Kaminski, Philipp Obermeier, Orkunt Sabuncu, and Torsten Schaub. Stream Reasoning with Answer Set Programming: Preliminary Report. In *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR 2012)*, page 613–617. AAAI Press, 2012.

[72] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, Sven Thiele, and Philipp Wanko. *Potassco User Guide.* University of Potsdam, second edition, 2019.

[73] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[74] Martin Gebser, Roland Kaminski, and Torsten Schaub. Complex Optimization in Answer Set Programming. *Theory and Practice of Logic Programming*, 11(4–5):821–839, 2011.

[75] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-Driven Answer Set Solving: From Theory to Practice. *Artificial Intelligence*, 187–188:52–89, 2012.

[76] Martin Gebser, Torsten Schaub, and Sven Thiele. GrinGo: A New Grounder for Answer Set Programming. In *Proceedings of the 9th International Conference on Logic Programming (LPNMR 2007)*, pages 266–271. Springer, 2007.

[77] Michael Gelfond and Yulia Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.* Cambridge University Press, 2014.

[78] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICLP/SLP '88)*, pages 1070–1080. MIT Press, 1988.

[79] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[80] Eric Bernd Gil, Ricardo Caldas, Arthur Rodrigues, Gabriel Levi Gomes Da Silva, Genaina Nunes Rodrigues, and Patrizio Pelliccione. Body Sensor Network: A Self-Adaptive System Exemplar in the Healthcare Domain. In *Proceedings of the 2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2021)*, pages 224–230. IEEE, 2021.

[81] Fred Glover and Manuel Laguna. Tabu Search. In *Handbook of Combinatorial Optimization*, pages 2093–2229. Springer, 1998.

[82] Thomas Goelles, Birgit Schlager, and Stefan Muckenhuber. Fault Detection, Isolation, Identification and Recovery (FDIIR) Methods for Automotive Perception Sensors Including a Detailed Literature Survey for Lidar. *Sensors*, 20(13):1–21, 2020.

[83] Franciszek Grabski. *Semi-Markov Processes: Applications in System Reliability and Maintenance*. Elsevier, 2015.

[84] Tao Gu, Xiao Hang Wang, Hung Keng Pung, and Da Qing Zhang. An Ontology-based Context Model in Intelligent Environments. *arXiv preprint arXiv:2003.05055*, 2020.

[85] Hendrik-Jörn Günther. *Collective Perception in Vehicular Ad-hoc Networks*. Dissertation, Technische Universität Carolo-Wilhelmina zu Braunschweig, 2017.

[86] Hao Guo, Ehsan Meamari, and Chien Chung Shen. Blockchain-inspired Event Recording System for Autonomous Vehicles. In *Proceedings of the 1st IEEE International Conference on Hot Information-Centric Networking (HotICN 2018)*, pages 218–222. IEEE, 2018.

[87] Terry Halpin and Tony Morgan. *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design*. Morgan Kaufman, 2001.

[88] Hashem M. Hashemian. *Maintenance of Process Instrumentation in Nuclear Power Plants*. Springer, 2006.

[89] Maryam Hemmati, Morteza Biglari-Abhari, and Smail Niar. Adaptive Vehicle Detection for Real-time Autonomous Driving System. In *Proceedings of the 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE 2019)*, pages 1034–1039. IEEE, 2019.

[90] Maryam Hemmati, Morteza Biglari-Abhari, and Smail Niar. Adaptive Real-Time Object Detection for Autonomous Driving Systems. *Journal of Imaging*, 8, 2022.

[91] Mojtaba Hemmati, Maghsoud Amiri, and Mostafa Zandieh. Optimization Redundancy Allocation Problem with Nonexponential Repairable Components using Simulation Approach and Artificial Neural Network. *Quality and Reliability Engineering International*, 34:278–297, 2018.

[92] Karen Henricksen and Jadwiga Indulska. Developing Context-Aware Pervasive Computing Applications: Models and Approach. *Pervasive and Mobile Computing*, 2(1):37–64, 2006.

[93] Henry Hexmoor and Kailash Yelasani. Economized Sensor Data Processing with Vehicle Platooning. *International Journal of Computer and Information Engineering*, 12(6):428–433, 2018.

[94] J. Wesley Hines and Brandon Rasmussen. Online Sensor Calibration Monitoring Uncertainty Estimation. *Nuclear Technology*, 151(3):281–288, 2005.

[95] John H Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[96] José Miguel Horcas, Julien Monteil, Mélanie Bouroche, Mónica Pinto, Lidia Fuentes, and Siobhán Clarke. Context-Dependent Reconfiguration of Autonomous Vehicles in Mixed Traffic. *Journal of Software: Evolution and Process*, 30(4), 2018.

[97] Timo Frederik Horeis, Tobias Kain, Julian-Steffen Müller, Fabian Plinke, Johannes Heinrich, Maximilian Wesche, and Hendrik Decke. A Reliability Engineering Based Approach to Model Complex and Dynamic Autonomous Systems. In *Proceedings of the 3rd IEEE International Conference on Connected and Autonomous Driving* (*MetroCAD 2020*). IEEE, 2020.

[98] Timo Frederik Horeis, Tobias Kain, Rhea C. Rinaldo, and Aaron Blickle. A Modeling Approach to Consider the Effects of Security Attacks on the Safety Assessment of Autonomous Vehicles - An AT-CARS Extension and Case-Study. In *Proceedings of the 31th European Safety and Reliability Conference* (*ESREL 2021*). Research Publishing, 2021.

[99] Timo Frederik Horeis, Tobias Kain, Rhea C. Rinaldo, Hans Tompits, Fabian Plinke, and Johannes Heinrich. Cross-Industry Overview of Fault-Tolerant Approaches used in Autonomous Systems. In *Proceedings of the 35th VDI-Fachtagung Fahrerassistenzsysteme und Automatisiertes Fahren* (*FAS 2022*), pages 89–108. VDI, 2022.

[100] Paul Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology, 2001. White paper.

[101] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From SHIQ and RDF to OWL: The Making of a Web Ontology Language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[102] Markus C. Huebscher and Julie A. McCann. A Survey of Autonomic Computing - Degrees, Models, and Applications. *ACM Computing Surveys*, 40(3), 2008.

[103] Rasheed Hussain and Sherali Zeadally. Autonomous Cars: Research Results, Issues, and Future Challenges. *IEEE Communications Surveys and Tutorials*, 21(2):1275–1313, 2019.

[104] IBM. An Architectural Blueprint for Autonomic Computing, 2005. White paper.

[105] Institute of Electrical and Electronics Engineers. *610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology*. IEEE, 1990.

[106] International Atomic Energy Agency. *SSR-2/1 (Rev. 1):Safety of Nuclear Power Plants: Design*. IAEA, 2016.

[107] International Electrotechnical Commission. *IEC 61025: Fault Tree Analysis (FTA)*. IEC, 2006.

[108] International Electrotechnical Commission. *IEC 61078: Reliability block diagrams*. IEC, 2016.

[109] International Organization for Standardization. *ISO/IEC 2382:2015(en) Information technology — Vocabulary*. ISO, 2015.

[110] Rolf Isermann and Peter Ballé. Trends in the Application of Model-based Fault Detection and Diagnosis of Technical Processes. *Control Engineering Practice*, 5(5):709–719, 1997.

[111] Steffen Jaekel and Bastian Scholz. Utilizing Artificial Intelligence to Achieve a Robust Architecture for Future Robotic Spacecraft. In *Proceedings of the 2015 IEEE Aerospace Conference (AeroConf 2015)*. IEEE, 2015.

[112] Dongyao Jia, Kejie Lu, Jianping Wang, Xiang Zhang, and Xuemin Shen. A Survey on Platoon-Based Vehicular Cyber-Physical Systems. *IEEE Communications Surveys Tutorials*, 18(1):263–284, 2016.

[113] Hao Jing, Yang Gao, Sepeedeh Shahbeigi, and Mehrdad Dianati. Integrity Monitoring of GNSS/INS Based Positioning Systems for Autonomous Vehicles: State-of-the-Art and Open Challenges. *IEEE Transactions on Intelligent Transportation Systems*, 2022.

[114] Kichun Jo, Junsoo Kim, Dongchul Kim, Chulhoon Jang, and Myoungho Sunwoo. Development of Autonomous Car - Part II: A Case Study on the Implementation of an Autonomous Driving System Based on Distributed Architecture. *IEEE Transactions on Industrial Electronics*, 62(8):5119–5132, 2015.

[115] Johannes Heinrich and Julian-Steffen Müller and Fabian Plinke and Timo Frederik Horeis and Henrik Decke. State-based Availability Analysis of Hard- and Software Architectures using Monte Carlo Simulation under Consideration of Different Failure Modes and Degradation Models. In *Proceedings of the 29th European Safety and Reliability Conference (ESREL 2019)*, pages 249–254. Research Publishing, 2019.

[116] Tobias Kain. Towards a Reliable System Architecture for Autonomous Vehicles, 2020. Master's Thesis. Institute of Logic and Computation, Technische Universität Wien.

[117] Tobias Kain, Timo Frederik Horeis, Johannes Heinrich, Marcel Aguirre Mehlhorn, Hans Tompits, Fabian Plinke, and Julian-Steffen Müller. HRR: A Hardware-Redundancy Recovery Approach. In *Proceedings of the 30th VDI-Fachtagung Technische Zuverlässigkeit 2021* (*TTZ 2021*). VDI, 2021.

[118] Tobias Kain, Marcel Aguirre Mehlhorn, Hans Tompits, and Julian-Steffen Müller. Dynamic Cooperation-Based Approach for Reducing Resource Consumption in Autonomous Vehicles. In *Proceedings of the 31th European Safety and Reliability Conference* (*ESREL 2021*). Research Publishing, 2021.

[119] Tobias Kain, Julian-Steffen Müller, Philipp Mundhenk, Hans Tompits, Maximilan Wesche, and Hendrik Decke. Towards a Reliable and Context-Based System Architecture for Autonomous Vehicles. In *Proceedings of the 2nd Workshop on Autonomous Systems Design* (*ASD 2020*), pages 1:1–1:7. Dagstuhl, 2020.

[120] Tobias Kain and Hans Tompits. ReConf: An Automatic Context-based Software Reconfiguration Tool for Autonomous Vehicles using Answer-Set Programming. In *Proceedings of the 22nd International Conference of the Italian Association for Artificial Intelligence* (*AIxIA 2023*). Springer, 2023.

[121] Tobias Kain, Hans Tompits, Timo Frederik Horeis, Johannes Heinrich, Julian-Steffen Müller, Fabian Plink, Hendrik Decke, and Marcel Aguirre Mehlhorn. C-po: A Context-Based Application-Placement Optimization for Autonomous Vehicle. In *Proceedings of the 3nd Workshop on Autonomous Systems Design* (*ASD 2021*). IEEE, 2021.

[122] Tobias Kain, Hans Tompits, Julian-Steffen Müller, Philipp Mundhenk, Maximilan Wesche, and Hendrik Decke. Fdiro: A General Approach for a Fail-Operational System Design. In *Proceedings of the 30th European Safety and Reliability Conference* (*ESREL 2020*). Research Publishing, 2020.

[123] Tobias Kain, Hans Tompits, Julian-Steffen Müller, Maximilan Wesche, Yael Abelardo Martinez Flores, and Hendrik Decke. Optimizing the Placement of Applications in Autonomous Vehicles. In *Proceedings of the 30th European Safety and Reliability Conference* (*ESREL 2020*). Research Publishing, 2020.

[124] Alexandru Kampmann, Bassam Alrifaee, Markus Kohout, Andreas Wustenberg, Timo Woopen, Marcus Nolte, Lutz Eckstein, and Stefan Kowalewski. A Dynamic Service-Oriented Software Architecture for Highly Automated Vehicles. In *Proceedings of the 2019 IEEE Intelligent Transportation Systems Conference* (*ITSC 2019*), pages 2101–2108. IEEE, 2019.

[125] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi. Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems* (*ICCPS 2018*), pages 287–296. IEEE, 2018.

[126] Tanwee Kausar, Priyanka Gupta, Deepesh Arora, and Rishabh Kumar. A VANET based Cooperative Collision Avoidance System for a 4-Lane Highway. *Notes on Engineering Research and Development, IIT Kanpur Technical Journal*, 4, 2012.

[127] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[128] Jari Kettunen, Teemu Reiman, and Björn Wahlström. Safety Management Challenges and Tensions in the European Nuclear Power Industry. *Scandinavian Journal of Management*, 23(4):424–444, 2007.

[129] Saddam Hussain Khan, Muhammad Haroon Yousaf, Fiza Murtaza, and Sergio Velastin. Passenger Detection and Counting for Public Transport System. *NED University Journal of Research*, 17(2):35–46, 2020.

[130] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[131] Philip Koopman and Frank Fratrik. How Many Operational Design Domains, Objects, and Events? In *Proceedings of the Workshop on Artificial Intelligence Safety (SafeAI 2019)*. CEUR, 2019.

[132] Philip Koopman and Michael Wagner. Challenges in Autonomous Vehicle Testing and Validation. *SAE International Journal of Transportation Safety*, 4(1):15–24, 2016.

[133] Philip Koopman and Michael Wagner. Autonomous Vehicle Safety: An Interdisciplinary Challenge. *IEEE Intelligent Transportation Systems Magazine*, 9(1):90–96, 2017.

[134] Robert Kowalski. Logic Programming. In *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 523–569. North-Holland, 2014.

[135] Jeff Kramer and Jeff Magee. Self-Managed Systems: An Architectural Challenge. In *Proceedings of the Future of Software Engineering Symposium (FOSE 2007)*, pages 259–268. IEEE, 2007.

[136] Pushpendra Kumar, Rochdi Merzouki, Blaise Conrard, and Belkacem Ould-Bouamama. Multilevel Reconfiguration Strategy for the System of Systems Engineering: Application to Platoon of Vehicles. In *Proceedings of the 19th International Federation of Automatic Control World Congress (IFAC 2014)*, pages 8103–8109. Elsevier, 2014.

[137] Sampo Kuutti, Saber Fallah, Konstantinos Katsaros, Mehrdad Dianati, Francis Mccullough, and Alexandros Mouzakitis. A Survey of the State-of-the-Art Localization Techniques and Their Potentials for Autonomous Vehicle Applications. *IEEE Internet of Things Journal*, 5(2):829–846, 2018.

[138] Anastasios Kyrillidis, Anshumali Shrivastava, Moshe Y. Vardi, and Zhiwei Zhang. Solving hybrid Boolean constraints in continuous space via multilinear Fourier expansions. *Artificial Intelligence*, 299, 2021.

[139] Jaynarayan H. Lala and Richard E. Harper. Architectural Principles for Safety-Critical Real-Time Applications. *Proceedings of the IEEE*, 82(1):25–40, 1994.

[140] Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Yijun Yu. Towards Requirements-Driven Autonomic Systems Design. *ACM SIGSOFT Software Engineering Notes*, 30(4), 2005.

[141] 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.

[142] Gregory Levitin and Anatoly Lisnianski. Optimal Separation of Elements in Vulnerable Multi-State Systems. *Reliability Engineering and System Safety*, 73(1):55–66, 2001.

[143] 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. IEEE, 2009.

[144] Chu Min Li and Felip Manya. MaxSAT, Hard and Soft Constraints. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336, pages 903–927. IOS Press, 2009. Second edition, 2021.

[145] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. Symbolic optimization with SMT solvers. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*, page 607–618. ACM, 2014.

[146] Zhixia Li, Madhav V. Chitturi, Lang Yu, Andrea R. Bill, and David A. Noyce. Sustainability Effects of Next-Generation Intersection Control for Autonomous Vehicles. *Transport*, 30:342–352, 2015.

[147] Mark H. Liffiton and Jordyn C. Maglalang. A Cardinality Solver: More Expressive Constraints for Free. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT 2012)*, pages 485–486. Springer, 2012.

[148] Nikolaos Limnios. *Fault Trees*. John Wiley & Sons, 2013.

[149] Todd Litman. Autonomous Vehicle Implementation Predictions: Implications for Transport Planning. Technical report, Victoria Transport Policy Institute, 2024.

[150] Jinguo Liu, Qing Gao, Zhiwei Liu, and Yangmin Li. Attitude Control for Astronaut Assisted Robot in the Space Station. *International Journal of Control, Automation and Systems*, 14(4):1082–1095, 2016.

[151] Shaoshan Liu, Liyun Li, Jie Tang, Shuang Wu, and Jean-Luc Gaudiot. *Creating Autonomous Vehicle Systems*. Morgan & Claypool Publishers, 2020.

[152] Yang Liu, Yu Peng, Bailing Wang, Sirui Yao, and Zihe Liu. Review on Cyber-physical Systems. *IEEE/CAA Journal of Automatica Sinica*, 4(1):27–40, 2017.

[153] Igor Lopez, Marina Aguado, Denis Ugarte, Alaitz Mendiola, and Marivi Higuero. Exploiting Redundancy and Path Diversity for Railway Signalling Resiliency. In *Proceedings of the 2016 IEEE International Conference on Intelligent Rail Transportation* (*ICIRT 2016*), pages 432–439. IEEE, 2016.

[154] Victor López-Jaquero, Francisco Montero, Elena Navarro, Antonio Esparcia, and José Antonio Catalán. Supporting ARINC 653-based Dynamic Reconfiguration. In *Proceedings of the 2012 Joint Working Conference on Software Architecture and 6th European Conference on Software Architecture* (*WICSA/ECSA 2012*), pages 11–20. IEEE, 2012.

[155] Rui Loureiro, Samir Benmoussa, Youcef Touati, Rochdi Merzouki, and Belkacem Ould Bouamama. Integration of Fault Diagnosis and Fault-Tolerant Control for Health Monitoring of a Class of MIMO Intelligent Autonomous Vehicles. *IEEE Transactions on Vehicular Technology*, 63(1):30–39, 2014.

[156] Bill Lozanovski, David Downing, Phuong Tran, Darpan Shidid, Ma Qian, Peter Choong, Milan Brandt, and Martin Leary. A Monte Carlo Simulation-based Approach to Realistic Modelling of Additively Manufactured Lattice Structures. *Additive Manufacturing*, 32, 2020.

[157] Pietro Lungaro, Konrad Tollmar, and Thomas Beelen. Human-to-AI Interfaces for Enabling Future Onboard Experiences. In *Proceedings of the 9th International Conference on Automotive User Interfaces and Interactive Vehicular Applications Adjunct* (*AutomotiveUI 2017*), page 94–98. ACM, 2017.

[158] Benjamin Lussier, Alexandre Lampe, Raja Chatila, Jérémie Guiochet, Félix Ingrand, Marc-Olivier Killijian, and David Powell. Fault Tolerance in Autonomous Systems: How and How Much? In *Proceedings of the 4th IARP-IEEE/RAS-EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments* (*DRHE 2005*). IEEE, 2005.

[159] Jianping Ma and Jin Jiang. Applications of Fault Detection and Diagnosis Methods in Nuclear Power Plants: A Review. *Progress in Nuclear Energy*, 53(3):255–266, 2011.

[160] Frank D. Macías-Escrivá, Rodolfo Haber, Raul Del Toro, and Vicente Hernandez. Self-Adaptive Systems: A Survey of Current Approaches, Research Challenges and Applications. *Expert Systems with Applications*, 40(18):7267–7279, 2013.

[161] Piergiuseppe Mallozzi, Patrizio Pelliccione, Alessia Knauss, Christian Berger, and Nassar Mohammadiha. Autonomous Vehicles: State of the Art, Future Trends, and Challenges. In *Automotive Systems and Software Engineering: State of the Art and Future Trends*, pages 347–367. Springer, 2019.

[162] Ragavan Manian, J Bechta Dugan, David Coppit, and Kevin J Sullivan. Combining Various Solution Techniques for Dynamic Fault Tree Analysis of Computer Systems. In *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium (HASE '98)*, pages 21–28. IEEE, 1998.

[163] Markus Maurer, J. Christian Gerdes, Barbara Lenz, and Hermann Winner. *Autonomous Driving: Technical, Legal and Social Aspects*. Springer, 2016.

[164] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for Combinatorial Optimization: A Survey. *Computers & Operations Research*, 134, 2021.

[165] Don L. McLeish. *Monte Carlo Simulation & Finance*. John Wiley & Sons, 2011.

[166] Nicholas Metropolis and Stanislaw Ulam. The Monte Carlo Method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.

[167] Mohammad Modarres, Mark P. Kaminskiy, and Vasiliy Krivtsov. *Reliability Engineering and Risk Analysis: A Practical Guide*. CRC Press, third edition, 2016.

[168] Michitsugu Mori, Masaku Kaino, Shigeru Kanemoto, Mitsuhiro Enomoto, Shigeo Ebata, and Shigeaki Tsunoyama. Development of Advanced Core Noise Monitoring System for BWRs. *Progress in Nuclear Energy*, 43(1):43–49, 2003.

[169] William R Morrow, Jeffery B Greenblatt, Andrew Sturges, Samveg Saxena, Anand Gopal, Dev Millstein, Nihar Shah, and Elisabeth A Gilmore. Key Factors Influencing Autonomous Vehicles' Energy and Environmental Outcome. In *Road Vehicle Automation*, pages 127–135. Springer, 2014.

[170] Farzeen Munir, Shoaib Azam, Muhammad Ishfaq Hussain, Ahmed Muqeem Sheri, and Moongu Jeon. Autonomous Vehicle: The Architecture Aspect of Self Driving Car. In *Proceedings of the 2018 International Conference on Sensors, Signal and Image Processing (SSIP 2018)*, pages 1–5. ACM, 2018.

[171] Victor P. Nelson. Fault-Tolerant Computing: Fundamental Concepts. *Computer*, 23(7):19–25, 1990.

[172] Philipp Nenninger. *Vernetzung verteilter sicherheitsrelevanter Systeme im Kraftfahrzeug*. PhD thesis, Universitätsverlag Karlsruhe, 2007.

[173] Daniela Nicklas and Bernhard Mitschang. On Building Location aware Applications using an Open Platform based on the NEXUS Augmented World Model. *Software and Systems Modeling*, 3(4):303–313, 2004.

[174] Zhaolong Ning, Jun Huang, and Xiaojie Wang. Vehicular Fog Computing: Enabling Real-Time Traffic Management for Smart Cities. *IEEE Wireless Communications*, 26(1):87–93, 2019.

[175] Robert P. Ocampo. Limitations of Spacecraft Redundancy: A Case Study Analysis. In *Proceedings of the 44th International Conference on Environmental Systems (ICES 2014)*. AIAA, 2014.

[176] Xavier Olive. FDI(R) for Satellite at Thales Alenia Space: How to Deal with High Availability and Robustness in Space Domain? In *Proceedings of the 2010 Conference on Control and Fault-Tolerant Systems (SysTol 2010)*, pages 837–842. IEEE, 2010.

[177] Sergey Orlov, Matthias Korte, Florian Oszwald, and Pascal Vollmer. Automatically Reconfigurable Actuator Control for Reliable Autonomous Driving Functions (AutoKonf). In *Proceedings of the 10th International Munich Chassis Symposium (CHASSIS.TECH PLUS 2019)*. Springer, 2019.

[178] Ilias Panagiotopoulos and George Dimitrakopoulos. Intelligent, In-Vehicle Autonomous Decision-Making Functionality for Driving Style Reconfigurations. *Electronics*, 12, 3 2023.

[179] Manish Parashar and Salim Hariri. Autonomic Computing: An Overview. In *Proceedings of the International Workshop on Unconventional Programming Paradigms (UPP 2004)*, volume 3566, pages 257–269. Springer, 2005.

[180] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context Aware Computing for the Internet of Things: A Survey. *IEEE Communications Surveys and Tutorials*, 16(1):414–454, 2014.

[181] Jian Qin, Ying Liu, and Roger Grosvenor. A Categorical Framework of Manufacturing for Industry 4.0 and Beyond. *Procedia CIRP*, 52:173–178, 2016.

[182] Miguel Realpe, Boris X. Vintimilla, and Ljubo Vlacic. A Fault Tolerant Perception System for Autonomous Vehicles. In *35th Chinese Control Conference (CCC 2016)*, pages 6531–6536. IEEE, 2016.

[183] Abdellah Rezoug, Mohamed Bader-el-den, and Dalila Boughaci. Application of Supervised Machine Learning Methods on the Multidimensional Knapsack Problem. *Neural Processing Letters*, 54:871–890, 2022.

[184] Timo Rieker. *Modellierung der Zuverlässigkeit technischer Systeme mit stochastischen Netzverfahren*. PhD thesis, University of Stuttgart, 2018.

[185] Rhea C. Rinaldo and Timo F. Horeis. A Hybrid Model for Safety and Security Assessment of Autonomous Vehicles. In *Proceedings of the 4th ACM Computer Science in Cars Symposium (CSCS '20)*. ACM, 2020.

[186] Rhea C. Rinaldo, Timo F. Horeis, and Tobias Kain. A Hybrid Modeling Approach for Analyzing Complex Autonomous Systems - A Reliability Assessment Case Study. In *Proceedings of the 31th European Safety and Reliability Conference (ESREL 2021)*. Research Publishing, 2021.

[187] Jose Roberto Vargas Rivero, Thiemo Gerbich, Valentina Teiluf, Boris Buschardt, and Jia Chen. Weather Classification Using an Automotive LIDAR Sensor Based on Detections on Asphalt and Atmosphere. *Sensors*, 20(15), 2020.

[188] Mostafa Sadighi, Saeid Setayeshi, and Ali Akbar Salehi. PWR Fuel Management Optimization using Neural Networks. *Annals of Nuclear Energy*, 29(1):41–51, 2002.

[189] SAE International. *J3016: Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. SAE, 2021.

[190] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The Context Toolkit: Aiding the Development of Context-Enabled Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*, pages 434–441. ACM, 1999.

[191] Mazeiar Salehie and Ladan Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2), 2009.

[192] Suryansh Saxena, Isaac K. Isukapati, Stephen F. Smith, and John M. Dolan. Multiagent Sensor Fusion for Connected Autonomous Vehicles to Enhance Navigation Safety. In *Proceedings of the 22nd IEEE Intelligent Transportation Systems Conference (ITSC 2019)*, pages 2490–2495. IEEE, 2019.

[193] Bill Schilit, Norman Adams, and Roy Want. Context-Aware Computing Applications. In *Proceedings of the First Workshop on Mobile Computing Systems and Applications (WMCSA '94)*, pages 85–90. IEEE, 1994.

[194] Wouter S. Schinkel, Tom P. J. Van Der Sande, and Henk Nijmeijer. Driver Intervention Detection via Real-Time Transfer Function Estimation. *IEEE Transactions on Intelligent Transportation Systems*, 22(2):772–781, 2021.

[195] Hartmut Schmeck. Organic Computing - A New Vision for Distributed Embedded Systems. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005)*, pages 201–203. IEEE, 2005.

[196] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[197] Roberto Sebastiani and Patrick Trentin. OptiMathSAT: A Tool for Optimization Modulo Theories. *Journal of Automated Reasoning*, 64(3):423–460, 2020.

[198] Hadas Shachnai and Tami Tamir. On Two Class-Constrained Versions of the Multiple Knapsack Problem. *Algorithmica*, 29(3):442–467, 2001.

[199] Johanneke Siljee, Ivor Bosloper, and Jos Nijhuis. A Quality Framework for the Storage and Retrieval of Context. *Revue d'Intelligence Artificielle*, 19(3):499–517, 2005.

[200] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138(1):181–234, 2002.

[201] Roy Sterritt and David Bustard. Autonomic Computing - A Means of Achieving Dependability? In *Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems* (*ECBS 2003*), pages 247–251. IEEE, 2003.

[202] Ronald Suich and Richard Patterson. How much Redundancy: Some Cost Considerations, including Examples for Spacecraft Systems. In *Proceeding of the AIChE Summer National Meeting Session on Space Power Systems Technology*. NASA, 1990.

[203] Morteza Taiebat, Austin L. Brown, Hannah R. Safford, Shen Qu, and Ming Xu. A Review on Energy, Environmental, and Sustainability Implications of Connected and Automated Vehicles. *Environmental Science & Technology*, 52(20):11449–11465, 2018.

[204] Han Shue Tan and Jihua Huang. DGPS-Based Vehicle-to-Vehicle Cooperative Collision Warning: Engineering Feasibility Viewpoints. *IEEE Transactions on Intelligent Transportation Systems*, 7(4):415–427, 2006.

[205] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. A Scalable Application Placement Controller for Enterprise Data Centers. In *Proceedings of the 16th International Conference on World Wide Web* (*WWW 2007*), pages 331–340. ACM, 2007.

[206] Sheng-Hsien Gary Teng and Shin-Yann Michael Ho. Failure Mode and Effects Analysis: An Integrated Approach for Product Design and Process Control. *International Journal of Quality & Reliability Management*, 13(5):8–26, 1996.

[207] Martin Törngren and Paul T. Grogan. How to Deal with the Complexity of Future Cyber-Physical Systems? *Designs*, 2(4), 2018.

[208] Pascal Traverse, Isabelle Lacaze, and Jean Souyris. Airbus Fly-by-Wire: A Total Approach to Dependability. In *Proceedings of the IFIP 18th World Computer Congress*, volume 156 of *IFIP International Federation for Information Processing*, pages 191–212. Springer, 2004.

[209] Sadayuki Tsugawa. Results and issues of an automated truck platoon within the energy its project. In *Poceedings of the 2014 IEEE Intelligent Vehicles Symposium (IV 2014)*, pages 642–647, 2014.

[210] Sadayuki Tsugawa, Teruo Yatabe, Takeshi Hirose, and Shuntetsu Matsumoto. An Automobile with Artificial Intelligence. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence (IJCAI '79)*, pages 893–895. Morgan Kaufmann Publishers, 1979.

[211] National Highway Traffic Safety Administration U.S. Department of Transportation. *Automated Driving Systems 2.0: A Vision for Safety*. Penny Hill Press, 2017.

[212] Hamed Vahdat-Nejad, Azam Ramazani, Tahereh Mohammadi, and Wathiq Mansoor. A Survey on Context-Aware Vehicular Network Applications. *Vehicular Communications*, 3:43–57, 2016.

[213] Ardalan Vahidi and Antonio Sciarretta. Energy Saving Potentials of Connected and Automated Vehicles. *Transportation Research Part C: Emerging Technologies*, 95:822–843, 2018.

[214] Gustavo Velasco-Hernandez, De Jong Yeong, John Barry, and Joseph Walsh. Autonomous Driving Architectures, Perception and Data Fusion: A Review. In *Proceedings of the 16th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP 2020)*, pages 315–321. IEEE, 2020.

[215] Junfeng Wang, Jungang Wang, Clive Roberts, and Lei Chen. Parallel Monitoring for the Next Generation of Train Control Systems. *IEEE Transactions on Intelligent Transportation Systems*, 16(1):330–338, 2015.

[216] Daniel Watzenig and Horn Martin. *Automated Driving*. Springer, 2017.

[217] Waloddi Weibull. A Statistical Distribution Function of Wide Applicability. *Journal of Applied Mechanics*, 18(3):293–297, 1951.

[218] 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. IEEE, 2013.

[219] Gereon Weiss, Marc Zeller, and Dirk Eilers. Towards Automotive Embedded Systems with Self-X Properties. In *New Trends and Developments in Automotive System Engineering*, chapter 21. IntechOpen, 2011.

[220] Danny Weyns, M. Usman Iftikhar, Sam Malek, and Jesper Andersson. Claims and Supporting Evidence for Self-Adaptive Systems: A Literature Study. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2012)*, pages 89–98. IEEE, 2012.

[221] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1–2):67–96, 2012.

[222] Konrad Wrona and Laurent Gomez. Context-Aware Security and Secure Context-Awareness in Ubiquitous Computing Environments. *Annales UMCS Informatica AI*, 4(1):332–348, 2006.

[223] Juan Ye, Lorcan Coyle, Simon Dobson, and Paddy Nixon. Ontology-based Models in Pervasive Computing Systems. *The Knowledge Engineering Review*, 22(4):315–347, 2007.

[224] Y. C. Yeh. Triple-Triple Redundant 777 Primary Flight Computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference (AeroConf 1996)*, pages 293–307. IEEE, 1996.

[225] Fisher Yu, Haofeng Chen, Xin Wang, Wenqi Xian, Yingying Chen, Fangchen Liu, Vashisht Madhavan, and Trevor Darrell. BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning. In *In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2020)*, pages 2636–2645. IEEE, 2020.

[226] Edith Zavala, Xavier Franch, Jordi Marco, and Christian Berger. HAFLoop: An Architecture for Supporting Highly Adaptive Feedback Loops in Self-adaptive Systems. *Future Generation Computer Systems*, 105:607–630, 2020.

[227] Edith Zavala, Xavier Franch, Jordi Marco, and Christian Berger. Adaptive Monitoring for Autonomous Vehicles using the HAFLoop Architecture. *Enterprise Information Systems*, 15(2):270–298, 2021.

[228] Marc Zeller, Gereon Weiss, Dirk Eilers, and Rudi Knorr. A Multi-layered Control Architecture for Self-Management in Adaptive Automotive Systems. In *Proceedings of the 2009 International Conference on Adaptive and Intelligent Systems (ICAIS 2009)*, pages 63–68. IEEE, 2009.

[229] Zhibin Zhang, Xinhong Li, Yanyan Li, Gangxuan Hu, Xun Wang, Guohui Zhang, and Haicheng Tao. Modularity, Reconfigurability, and Autonomy for the Future in Spacecraft: A Review. *Chinese Journal of Aeronautics*, 36:282–315, 2023.

[230] Haitao Zhao, Hai Liu, Yiu Wing Leung, and Xiaowen Chu. Self-Adaptive Collective Motion of Swarm Robots. *IEEE Transactions on Automation Science and Engineering*, 15:1533–1545, 2018.

[231] Peng Zhou, Decheng Zuo, Kun Mean Hou, Zhan Zhang, Jian Dong, Jianjin Li, and Haiying Zhou. A Comprehensive Technological Survey on the Dependable Self-Management CPS: From Self-Adaptive Architecture to Self-Management Strategies. *Sensors*, 19(5), 2019.

[232] Martin Zimmermann and Franz Wotawa. An Adaptive System for Autonomous Driving. *Software Quality Journal*, 28(3):1189–1212, 2020.

[233] Ömer Şahin Taş, Florian Kuhnt, J. Marius Zöllner, and Christoph Stille. Functional System Architectures towards Fully Automated Driving. In *Poceedings of the 2016 IEEE Intelligent Vehicles Symposium (IV 2016)*, pages 304–309. IEEE, 2016.