TU WIEN Informatics

Towards a Reliable System Architecture for Autonomous Vehicles

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

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

Vienna, 5th March, 2020

Tobias Kain	Hans Tompits

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

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, 5. März 2020

Tobias Kain

Acknowledgements

I would like to thank my advisor Ao.Univ.-Prof. Hans Tompits for his academic support and the many hours we spent discussing the topic. Furthermore, I would like to thank my family and friends for their great moral support while working on my thesis.

Kurzfassung

Gegenwärtige Fahrzeuge sind mit einer Vielzahl an Fahrerassistenzsystemen ausgestattet, die den Fahrer während des Betriebs des Fahrzeugs unterstützen. Moderne Assistenzsysteme sind beispielsweise in der Lage einen festgelegten Abstand zu einem vorausfahrenden Fahrzeug einzuhalten, selbständig einzuparken sowie die Fahrspur auf Autobahnen zu wechseln. Alle diese Funktionen verfügen über eine hohe Zuverlässigkeit und wurden ausführlich getestet. Dennoch ist der Fahrer dazu angehalten, das Verhalten der Assistenzsysteme zu überwachen und bei Bedarf die Kontrolle zu übernehmen.

Bei vollautonomen Fahrzeugen, welche als *SAE Level 5-Fahrzeuge* bezeichnet werden, ist jedoch eine Übernahme der Kontrolle von Fahrgästen ausgeschlossen. Um Fahrzeuge vollautonom zu betreiben, müssen zahlreiche Softwareanwendungen, wie zum Beispiel Wahrnehmungs-, Planungs- und Fahrzeugssteuerungsdienste miteinander interagieren, wobei viele dieser Anwendungen als sicherheitskritisch einzustufen sind. Folgedessen kann ein Ausfall einer solchen Applikation zu einer für Personen gefährlichen Situation führen.

Um die Sicherheit der Fahrgäste und anderer Verkehrsteilnehmer im Falle eines auftretenden Fehlers gewährleisten zu können müssen daher Maßnahmen ergriffen werden, welche einen sicheren Betrieb in solchen Situationen ermöglichen.

Insbesondere am Beginn der Einführung von autonomen Fahrzeugen ist es wichtig, das Vertrauen der Kunden zu stärken. So ist, zum Beispiel, die Behandlung von Fehlern durch das Ausführen einen Nothalts nicht zielführend, da ein solches Verhalten Kunden verunsichern würde. Folglich muss das autonome Fahrsystem in der Lage sein, Ausfälle selbstständig zu erkennen und adäquat zu behandeln.

In dieser Arbeit wird ein Ansatz zur schrittweisen Behandlung von Fehlern vorgestellt. Dieser Ansatz basiert auf den aus der Luft- und Raumfahrt bekannten FDIR ("Fault Detection, Isolation, and Recovery") Methodologie, wobei die von FDIR definierten Schritte an den Anwendungsfall des autonomen Fahrens angepasst wurden. Darüber hinaus wurde der FDIR-Ansatz um einen Systemoptimierungsschritt erweitert, welcher die Stabilität und Effizienz des Gesamtsystems nach einer sicherheitskritischen Rekonfiguration optimiert. In diesem Sinne nennen wir unseren Ansatz FDIRO ("Fault Detection, Isolation, Recovery, and Optimization").

Da eine schnelle Rekonfiguration als eine wesentliche Anforderung vieler sicherheitskritischer Softwareanwendungen angesehen wird, ist der Erkennungs- und Isolierungsschritt

des FDIRO-Ansatzes von geringer Komplexität. Daher können diese Schritte innerhalb von Millisekunden durchgeführt werden.

Um zu zeigen, dass der Erkennungs- und Isolierungsschritt in kurzer Zeit durchgeführt werden kann wird eine Proof-of-Concept-Implementierung vorgestellt, welche dieses Verhalten veranschaulicht. Weiters wird eine Implementierung basierend auf linearer Programmierung vorgestellt, welche Wiederherstellungsmaßnahmen berechnet. Schlußendlich werden Konzepte zur Optimierung des Systems basierend auf Kontextbeobachtungen diskutiert.

Abstract

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

As far as fully autonomous vehicles are concerned, i.e., so-called *SAE Level 5 vehicles*, any takeover actions by passengers are excluded. To operate such autonomous vehicles, numerous software applications, including, for example, perception, planning, and vehicle control services, have to interact with each other. Many of these applications are safety-critical, i.e., a failure might result in a hazardous situation.

Therefore, to guarantee the safety of the passengers and other road users in case an occurring failure causes a safety-critical application to misbehave, measures have to be implemented to ensure a safe operation in such situations.

Especially in the initial phase of the development of autonomous vehicles, building up consumer confidence is essential. Therefore, in this regard, handling failures by performing an emergency stop, i.e., stopping the vehicle whenever a failure is detected, is not desirable as such behavior may reduce customer confidence. Consequently, the system responsible for operating the autonomous vehicle has to detect and handle failures autonomously, i.e., the system has to be *fail-operational*.

In this thesis, we introduce a fail-operational approach for handling failures in a stepwise fashion by adapting the FDIR ("Fault Detection, Isolation, and Recovery") approach known from the aerospace domain, whereby we reimplemented the steps defined by FDIR to fit the area of autonomous driving. Moreover, we extended the FDIR approach by a system optimization procedure that improves the system stability and efficiency after a safety-critical reconfiguration. Accordingly, we call our approach FDIRO, standing for "Fault Detection, Isolation, Recovery, and Optimization".

Since a fast reconfiguration time is considered an essential requirement of many safety-critical software applications, the detection and isolation steps of the FDIRO approach are designed to be of low complexity. Therefore, these steps can be performed within milliseconds.

To show that the detection and isolation steps can be performed in a short time, we provide a proof-of-concept implementation. We further present an implementation based on linear integer programming for determining recovery actions and introduce concepts for optimizing the system based on context observations.

Contents

CHAPTER 1

Introduction

Since the introduction of motorized vehicles in the early 20th century, the task of operating those machines was the responsibility of the driver. In recent years, however, vehicles have been gradually equipped with an increasing number of driver-assistance systems, which support the driver. Modern vehicles are, for instance, capable of keeping the distance to a preceding vehicle, switching lanes on highways, and reversing into a parking space autonomously.

Due to the rapid technological development, it is feasible to automate the task of driving in the near future entirely. Such autonomous vehicles, which are also referred to as *SAE Level 5 vehicles* [54], may yield many positive aspects. For instance, they can induce an improved traffic flow [17], fewer emissions [45], and provide mobility to the elderly or disabled individuals [15]. Furthermore, it is assumed that autonomous vehicles lead to safer roads [43].

However, since autonomous vehicles are complex distributed systems that consist of numerous safety-critical software applications [32], i.e., a failure of those applications might result in hazardous situations, strong efforts are necessary to ensure the safety of autonomous vehicles. As SAE Level 5 vehicles exclude any takeover actions by passengers, measures have to be implemented to handle failures autonomously in a safe manner.

Although handling failures by performing an emergency stop is feasible, such behavior is not desirable since this may cause customer satisfaction to decline [39]. 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 failure-acceptance rate might cause vehicles to operate unintentionally, leading to a loss of customer confidence. Therefore, solutions to handle this are required.

In this thesis, we introduce FDIRO ("Fault Detection, Isolation, Recovery, and Optimization"), a fail-operational approach for handling failures in a stepwise fashion, extended with a system optimization procedure that improves the system stability and efficiency

after the safety-critical reconfiguration. This approach is based on the Fdir ("Fault Detection, Isolation, and Recovery") approach known from the aerospace domain [68], which is also applied in other areas such as chemical [62] and nuclear engineering [67].

An essential feature of Fdir is the redundant design of applications. Fdiro adopts this characteristic by defining an active and several passive operation modes. The idea is that each safety-critical application is executed redundantly, whereby one application instance is executing in active operation mode and thus interacting with other applications. The other redundant application instances are executing a passive operation mode. Therefore, these instances are not interacting with other applications and can perform a degraded set of operations. However, passive instances are capable of quickly taking over the responsibilities of the active instance if required.

Due to the redundant design of safety-critical applications, the execution of the first three steps of the Fdiro approach, which correspond to the detection, isolation, and recovery actions as defined by Fdir, results in an error-free configuration system, which satisfies the demanded safety requirements. However, since the resulting configuration might not be optimal, we enhanced the Fdir strategy by an additional optimization step. The goal of this step is to optimize the application placement, i.e., the assignment between application instances and computing nodes, according to context observations, including, for example, environmental conditions and passenger preferences.

Besides enabling a context-based configuration optimization, Fdiro also ensures a fast reconfiguration time, which is considered an essential requirement of many safety-critical software applications. For instance, the maximum acceptable reconfiguration time of the application that controls the steering is 90 ms [48]. Fdiro meets this requirement due to keeping the complexity of the detection and isolation step low. Consequently, those steps can be performed within milliseconds.

Compared to the detection and isolation step of Fdiro, the subsequent recovery and optimization steps are not as time-critical. Therefore, the actions performed during the recovering and optimization procedure can be more complex and, therefore, more time-consuming.

The thesis is organized as follows: In Chapter 2, we provide background information about autonomous driving, the system architecture of autonomous vehicles, and functional safety. In Chapter 3, we introduce our stepwise reconfiguration approach, i.e., Fdiro. Chapter 4 focuses on the detection and isolation steps of Fdiro. In Chapter 5, we explain the redundancy recovery step in more detail. Chapter 6 presents ideas concerning the application-placement optimization. In Chapter 7, we compare our work to approaches that aim for similar goals. Finally, in Chapter 8, we conclude the thesis and discuss possible future work.

Parts of this thesis have been accepted in the proceedings of international conferences and workshops. To wit, the results of Chapter 3 and Chapter 4 appear in the proceedings of the 30$^{\text{th}}$ European Safety and Reliability Conference (ESREL 2020) [35]. Moreover, the methods introduced in Chapter 6 will be published in the proceedings

of the 2nd Autonomous System Design Workshop (ASD 2020) [34]. Finally, a paper summarizing results of Chapter 5 is submitted to the 31st IEEE Intelligent Vehicles Symposium (IV 2020) [36].

CHAPTER 2

Background

In what follows, we provide some general information about autonomous driving. Furthermore, we discuss the system architecture of autonomous vehicles as well as the functional safety concerns of such vehicles.

2.1 Autonomous Driving

In the past, vehicles were purely mechanical systems. In recent years, however, manufacturers gradually computerized their vehicles. Due to the rapid technological development, it is even feasible to (partially) automate the task of driving. To categorize the level of automaton, the Society of Automotive Engineers (SAE) introduced a classification, as illustrated in Table 2.1, ranging from no automation to full automation [54].

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 [61]. Nowadays, many automotive manufacturers, including, for example, Tesla, Daimler, Volkswagen, Ford, and Toyota as well as technology companies such as Google, Amazon, Nvidia, Intel, and Uber push the research of autonomous driving forward [29].

The development towards fully autonomous vehicles is to be welcomed since such vehicles can lead to safer roads [43], and an improved traffic flow [17]. Autonomous vehicles can also cause a social impact by, for example, providing mobility to the elderly or disabled individuals [15]. Furthermore, vehicle autonomy can lead to reduced energy consumption and fewer emissions [45].

Many of those benefits become effective once autonomous vehicles are accessible to the general public. According to the Victoria Transport Policy Institute [42], it is likely that this will be the case in about 20 to 30 years.

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 situations.
Level 5	Full Automation	The system performs the driving task autonomously. No driver is required.

2.2 Functional System Architecture of Autonomous Vehicles

As the research on autonomous vehicles is still in an early phase, none of the proposed software and hardware architecture concepts for autonomous vehicles is widely accepted. However, we can identify some functional components that can be found in most autonomous vehicles [63, 33, 9, 69], viz. such a functional system architecture consists of the following three components:

- perception,
- planning, and
- vehicle control.

The input for this architecture is the information provided by a set of various sensors. This information is used by the components to generate a set of actions that are performed by the actuators of the vehicle. Those actuators control, amongst other parameters, the longitudinal as well as lateral movement of the vehicle. The interplay between the functional components, sensors, and actuators is illustrated in Figure 2.1.

The perception unit is responsible for locating the vehicle with respect to a map. Since some use cases require accuracy in the range of centimeters [4], using only the position data provided by a GPS sensor is not sufficient as the accuracy of GPS ranges above 20

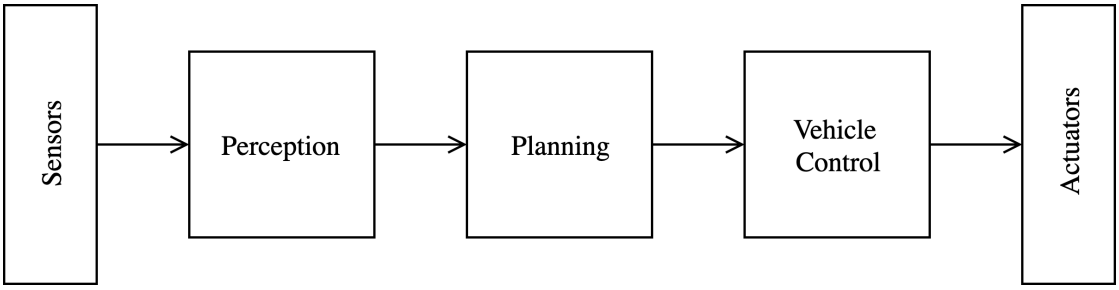

Figure 2.1: Functional system architecture of autonomous vehicles.

meters [57]. However, the location unit can increase its accuracy by considering additional information provided by, e.g., (i) cameras, (ii) the internal measurement unit, which measures the linear accelerations and the vehicle angular, (iii) radar, and (iv) LiDAR (light detection and ranging) sensors [41]. Furthermore, the perception unit can also use information shared by other vehicles [19] and the infrastructure [16]. Therefore, vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) information is required.

Besides determining the location of the vehicle, the perception unit is also responsible for observing the driving environment. Modern approaches mainly use the data received from LiDAR sensors and cameras to perform the environmental perception [50].

One task of observing the driving environment is, for instance, detecting obstacles such as other cars, passengers, and cyclists. Currently used perception units mainly apply deep learning approaches for detecting obstacles due to their superior performance [12]. Further tasks required for observing the driving environment include, for example, detecting traffic signs [3], recognizing road markings as well as extracting road surfaces [5].

The location information and environment observations determined by the perception unit are used as input for the planning component. Based on these data, the route, behavior, and trajectory of the vehicle are planned.

For determining the route a vehicle shall follow, applying classic search algorithms like A* [27] or Dijkstra's [14] is feasible. However, those approaches turn out to be inefficient in the case of large road networks. Thus, to improve the performance of route planning in such environments, more sophisticated algorithms have to be applied [6].

The behavioral planner is responsible for obeying the traffic rules. It determines, for example, the maximum speed, the minimum distance to the preceding vehicle, or whether the vehicle has to stop at an intersection or not [64]. To model the desired behaviors, finite-state machines can be used [44].

The output from the route planner and the behavioral planner are used by the trajectory planner to compute the path that describes how the vehicle shall traverse the environment with respect to time. Graph search and interpolating curve algorithms are currently the two main approaches used for implementing trajectory planners [21].

In the next step, the trajectory determined by the planning unit is used as an input for the vehicle control unit. This unit executes the planned trajectory by controlling the actuators responsible for the longitudinal and lateral motion of the vehicle.

2.3 Functional Safety

Even though the vehicles currently available on the consumer market are far from being considered highly automated, they are already equipped with various functions that are safety-critical, i.e., their misbehavior can cause hazardous situations. The progressive introduction of safety-critical functions, like brake-by-wire or electronic stability control, increases the probability that a failure occurs, which causes a safety-critical application to fail [37].

To reduce this risk, the International Organization for Standardization (ISO) introduced a standard concerning the functional safety of road vehicles, viz., ISO 26262 [30]. This standard defines functional safety as the absence of unreasonable risks due to hazards caused by malfunctioning behavior of electrical and electronic systems (so-called *E/E systems*) [30, Part 1]. To achieve this goal, ISO 26262 introduces, for example, a safety-driven development process as well as an approach for the risk classification of functions.

A common method for increasing functional safety is to monitor the behavior of safety-critical applications and to deactivate them in case a failure is detected. Such systems are referred to as *fail-safe systems*. Typically, traction control [20], power steering [40], as well as adaptive cruise control systems [49] are designed to be fail-safe. In case, for example, the adaptive course control system fails, the driver is warned that this function is disabled. Therefore, the tasks of keeping the speed and distance to the preceding vehicle are handed over to the driver.

Although a fail-safe behavior is acceptable in today's vehicles, for fully autonomous vehicles, more sophisticated approaches such as *fail-operational systems* are required since any takeover actions by the passengers are excluded.

Compared to fail-safe systems, fail-operational systems ensure a correct and safe operation even if the system is affected by failures [8]. In the following chapter, we introduce an approach that aims for a fail-operational behavior.

CHAPTER 3

An Approach for a Fail-Operational System Design

In this chapter, we introduce a concept for the implementation of a fail-operational system. The idea of our approach is to handle failures in a stepwise fashion to ensure a safe system state as fast as possible.

Since our idea of a fail-operational system is based on a redundant design of the software applications, which are required to operate the autonomous vehicle, we first introduce the redundancy concept adopted by our approach.

Afterwards, we introduce FDIRO, a stepwise reconfiguration approach standing for "Fault Detection, Isolation, Recovery, and Optimization", that is the core of our concept for a fail-operational system.

3.1 Software Redundancy Concept

As already mentioned before, the fundamental basis of our approach for a fail-operational system is a redundant design of the software applications. The idea thereby is that multiple instances of the same application are executed by the system, whereby the individual instances run in different operation modes. Generally speaking, we distinguish between an *active operation mode*, multiple *passive operation modes*, and one *isolated mode*.

An application instance that is executed in active operation mode is referred to as an *active instance*. These instances are actively interacting with the system, i.e., they provide data to other applications or are operating an actuator.

On the other hand, *passive instances*, i.e., application instances that are executed in one of the passive operation modes, do not interact with the system. They, however,

might perform the same operations as the corresponding active instances. A passive trajectory planner instance, for example, does not send commands to the controller that is responsible for steering the car.

Since passive instances are not interacting with the system, per application, multiple instances that run in passive operation mode are permitted. On the other hand, only one active instance per application is allowed because otherwise conflicts between multiple active instances of the same application might arise.

As mentioned before, our redundancy concept defines multiple passive operation modes. In total, we distinguish between three different passive modes, namely:

- *active-hot*,
- *passive-warm*, and
- *passive-cold*.

The difference between those three passive operation modes is the level of workload degradation. While active-hot instances perform the same operations as the corresponding active instances, passive-warm instances perform only a reduced set of actions. Therefore, active-hot instances can quickly take over the actions of active instances. Passive-cold instances do not utilize any computing resources. Their primary purpose is to block required resources so that in case it is required they can quickly upgrade to the active-hot operation mode. The idea behind defining three passive operation modes is to increase overall safety while reducing the consumption of computing resources.

Note that in the following we sometimes do not distinguish between the active-hot, passive-warm, and passive-cold operation mode. We then refer to all application instances executing one of these operation modes as *passive instances* and *redundant instances*, respectively.

Besides the active and the passive operation modes, our redundancy concept also defines an *isolated mode*. This operation mode prevents an instance from communicating with other instances or controlling actuators. The main purpose of this operation mode is to prevent further damage after detecting the malfunctioning of an application.

To illustrate the basic idea of our redundancy concept, consider the configuration shown in Figure 3.1. Each software application is executed by exactly one computing node. Assuming that all the applications are mission-critical, i.e., necessary for operating the autonomous vehicle safely, any failure that causes one of the applications to fail might result in an accident.

Figure 3.2 shows a redundant design of the system illustrated in Figure 3.1. For each application, one active and one passive instance is executed, whereby these two instances are located on different computing nodes, i.e., the level of hardware segregation is two.

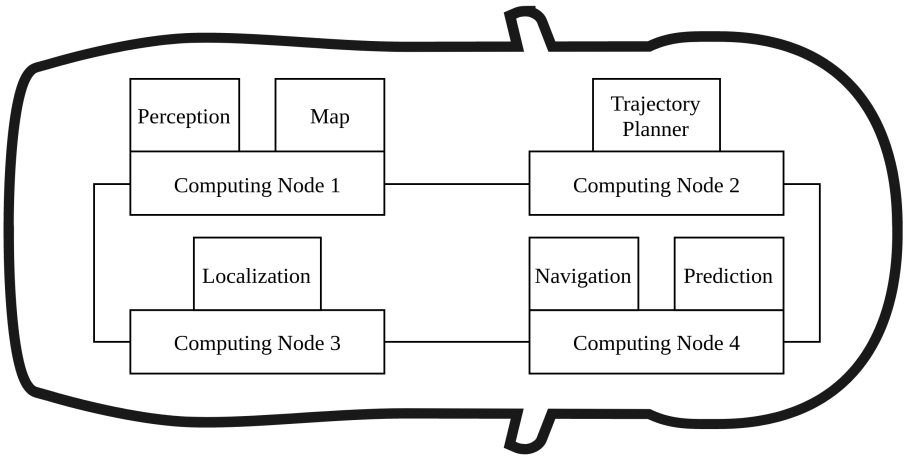

Figure 3.1: Configuration without redundant application instances.

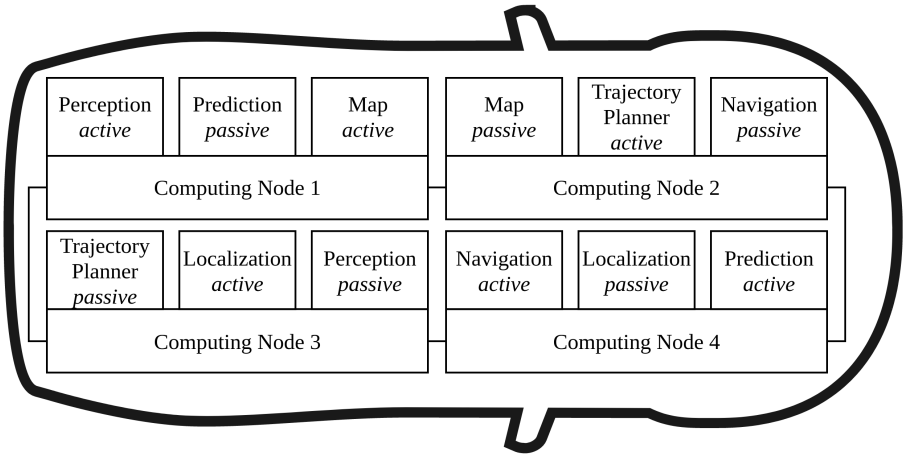

Figure 3.2: Configuration with redundant application instances.

Assuming that *Computing Node 2* fails, therefore, as shown in Figure 3.3, the active instance of the trajectory planner, as well as the passive instances of the map and the navigation application, fail. Since the passive instances are not interacting with the system, their failure does not endanger the driving mission. The failure of the active instance of the trajectory planner, on the other hand, affects the steering of the vehicle and therefore endangers the safety of the passengers as well as other road users.

However, because of the redundant design of the trajectory planner, an accident can be circumvented by changing the operation mode of the passive trajectory planner instance to active.

Since handling such failures is not a trivial task, we introduce next our novel approach realizing the idea of a stepwise reconfiguration.

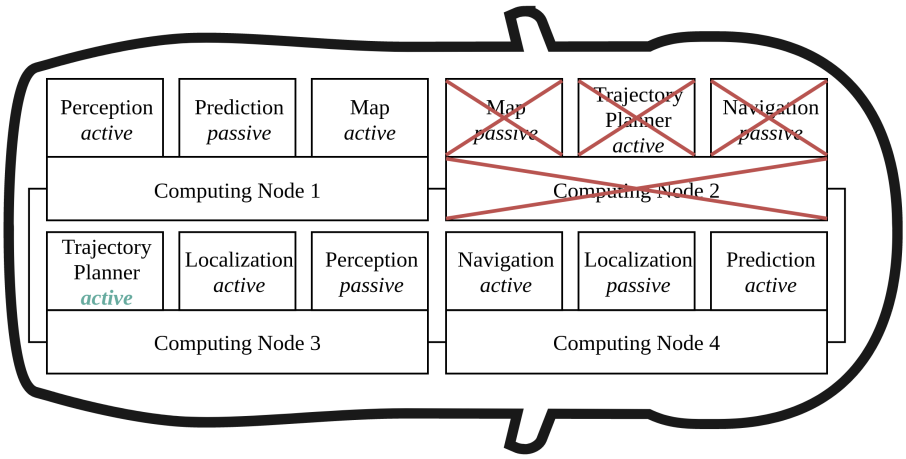

Figure 3.3: Configuration after the failure of computing node 2.

3.2 A Stepwise Reconfiguration Approach

Our approach for handling occurring failures is based on the Fdir ("Fault Detection, Isolation, and Recovery") approach known from the aerospace domain [68]. The idea is to handle failures by executing the following steps:

Step 1: detection of the failure,

Step 2: isolation of the failure by a switchover between redundant instances,

Step 3: recovery of the redundancy requirements, and

Step 4: optimizing the placement of the application instances.

The first three steps correspond to the detection, isolation, and recovery actions specified by Fdir. The execution of those steps results in an error-free configuration, which satisfies the required safety requirements. However, since the resulting configuration might not be optimal, we extended the Fdir strategy by an additional optimization step. Consequently, we refer to this extended Fdir approach as Fdiro.

The newly introduced optimization step aims to optimize the system according to goals that depend on the current situation. In the case of an autonomous electric vehicle, a conceivable goal is, for example, the extension of the range. Assuming that the energy efficiency of an electric vehicle can be improved by shutting down computing nodes, the optimization step might aim for finding an application placement that allows shutting down as many computing nodes as possible.

The motivation for implementing a stepwise reconfiguration is that for some failures, the reconfiguration time is critical. In some cases, the time until a failure is isolated, and a redundant instance takes over the actions of the faulty instance is required to be within milliseconds. The maximum acceptable reconfiguration time of an application that controls the steering is, for example, 90 ms [48].

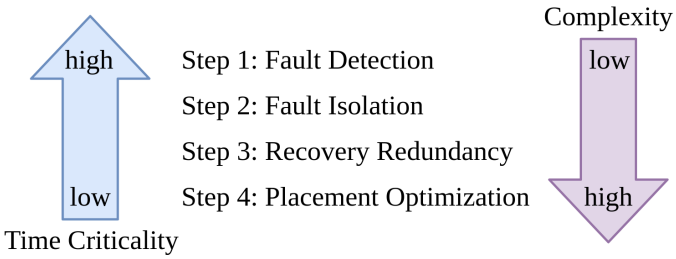

Figure 3.4: Comparison of time criticality and computational complexity of the different FDIRO steps.

The stepwise design of our reconfiguration approach allows us to meet this requirement since the complexity of the detection and isolation step is low. Therefore, those steps can be performed within a guaranteed time.

The complexity of the redundancy recovery step and the optimization step is, on the other hand, significantly higher. However, since the time criticality of the later steps is lower than the time criticality of the earlier steps, such behavior is acceptable. The complementary effect of time criticality and complexity is displayed in Figure 3.4.

Another valid approach to ensure short reconfiguration times is to precompute successor configurations for all possible failures that can occur. However, as we prove in what follows, such an approach is not feasible since today's storage capacities are not sufficient enough.

Assume that N is the set of computing nodes and I is the set of application instances, i.e., of all active and passive instances that are executed by the system.

Moreover, we define a *configuration*, C, as a mapping between the set of computing nodes and application instances. We encode configurations as $|N| \times |I|$ matrices, whereby an element is 1 in case the computing node executes the corresponding application instance or 0 otherwise.

Assuming that we consider only failures of computing nodes and application instances, in total there are $|N| + |I|$ possible failures that can occur. Since the reconfiguration actions, after a failure occurred, depend on all failures that happened previously, we have to consider all possible permutations of failures.

In total, there are $(|N| + |I|)!$ different sequences of failures. Since each failure results in a new configuration, per failure permutation $|N| + |I|$ configurations, apart from the initial configuration, are required. Therefore,

$$(|N| + |I|)! \cdot (|N| + |I|) + 1$$

configurations have to be precomputed. Since we defined that a configuration is represented by a $|N| \times |I|$ matrix, and the single elements adopt one of two possible

values, $|N| \cdot |I|$ bits of storage are required per configuration. Therefore, to store all $(|N| + |I|)! \cdot (|N| + |I|) + 1$ configurations,

$$((|N| + |I|)! \cdot (|N| + |I|) + 1) \cdot |N| \cdot |I|$$

bits are required.

Assuming, a car is equipped with four computing nodes, i.e., $|N| = 4$, which execute 100 application instance, i.e., $|I| = 100$, about $5 \cdot 10^{169}$ bytes of storage are required.

Even though configurations can be probably represented more efficiently and some failure sequences can be shortened we can assume that it is not feasible to store reconfiguration actions for all possible failures that can occur.

3.2.1 Workflow of FDIRO

Due to the stepwise design of FDIRO, it is evident to use a service-oriented architecture to implement this approach. In total, we define the following five different components, which collectively are responsible for performing the FDIRO approach:

- *monitors*,
- a *switchover component*,
- a *failover controller*,
- a *redundancy-recovery component*, and
- a *placement optimizer*.

In what follows, we describe the responsibilities of the different components as well as their interplay.

The monitors are the components that detect failures and consequently trigger a reconfiguration. We can distinguish between three types of monitors:

- *application-instance monitors*,
- *operating-system monitors*, and
- *hardware monitors*.

Application-instance monitors detect failures of application instances. To detect that an application instance is malfunctioning, the desired behavior of the application instance has to be known. Therefore, each application has its own application instance monitor that is monitoring all instances of that application.

Furthermore, each operating system shall be monitored by an operating system monitor, and each computing node shall be monitored by a hardware monitor.

In case that any of those different types of monitors detect a failure, they inform the switchover component about the affected application instances.

The switchover component then instructs the faulty instance to switch to the isolated operation mode. Recall, that this operation mode prevents the faulty instance communicating with other applications or controlling actuators.

Next, the switchover component checks whether for a faulty application instance redundant instances, i.e., application instances that run in a lower operation mode, exist. In case that redundant instances exist, the switchover component selects the instance with the highest operation mode that is lower than the operation mode of the faulty application instance. Note that we refer to the active operation mode as the *highest operation mode*, whereby passive-cold is considered as the *lowest operation mode*.

The switchover component then instructs the selected instance to switch to the operation mode of the faulty instance and subsequently hands the control to the redundancy-recovery component. In case that no redundant instance exists, the switchover component has to evaluate if the failure of the instance caused the safety level to drop below an acceptable threshold. If so, the switchover component instructs the failover controller to take over the control and safely stop the vehicle. Otherwise, the switchover component hands over the control to the redundancy-recovery component.

The redundancy-recovery component is responsible for restoring the lost redundancy. Therefore, this component obtains the current resource utilization of all available computing nodes. Using this information, the redundancy-recovery component determines whether one of the computing nodes offers enough free resources to run an instance that implements the same functionality as the faulty instance. If such a computing node exists, the redundancy-recovery component instructs that node to start a redundant instance that runs in the same operation mode as the instance that was selected by the switchover component.

In case that no computing node has sufficient resources left, the redundancy-recovery component is allowed to displace application instances and stop application instance of lower priority.

The actions performed by the redundancy-recovery component can be seen as a fast way to increase the safety of the system. However, the placement of the new redundant instance might not be optimal. Therefore, in the next step, the placement optimizer tries to find an application placement that is better suited for the current driving situation.

Therefore, the placement optimizer first determines the goals the application placement shall meet. Then it computes a placement that satisfies the previously defined goals. Finally, the placement optimizer rolls out the new placement plan to all computing nodes, which then perform the actions defined in the placement plan.

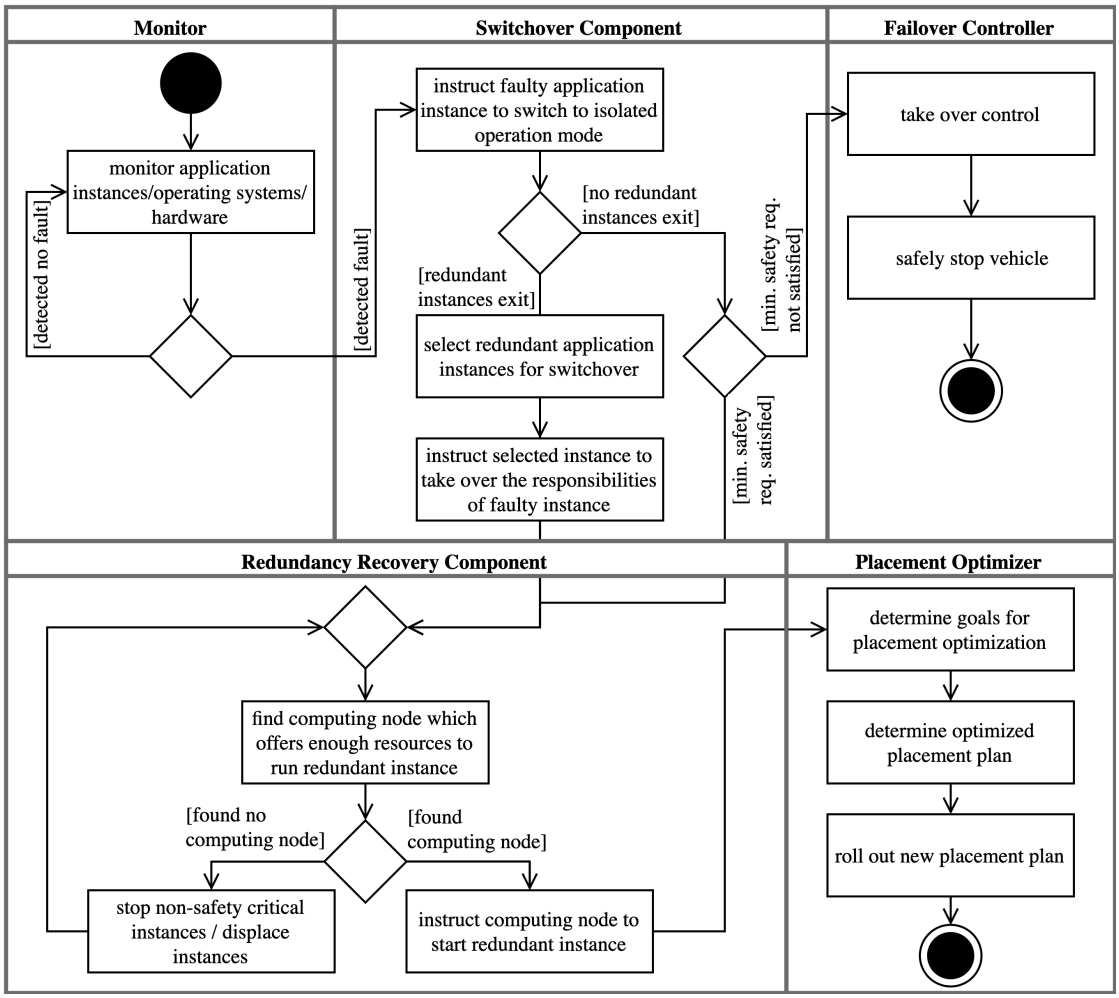

Figure 3.5: Activity diagram of the stepwise reconfiguration process.

Figure 3.5 illustrates the just described workflow in an activity diagram. To further clarify the tasks performed during the stepwise reconfiguration, we next provide a concrete use case.

3.2.2 Use Case: Failure of a Trajectory Planner

Assume a system, as illustrated in Figure 3.6, which executes five applications, whereby for each application, one redundant instance exists. In this example, we consider a failure of the active instance of the trajectory planner, which is executed by *Computing Node 2*.

After the failure occurred, the application instance monitor that is specifically designed to detect failures of trajectory planners has to become aware of the malfunctioning of the active instance of the trajectory planner. This part of the FDIRO process corresponds to Step 1, the detection of the failure.

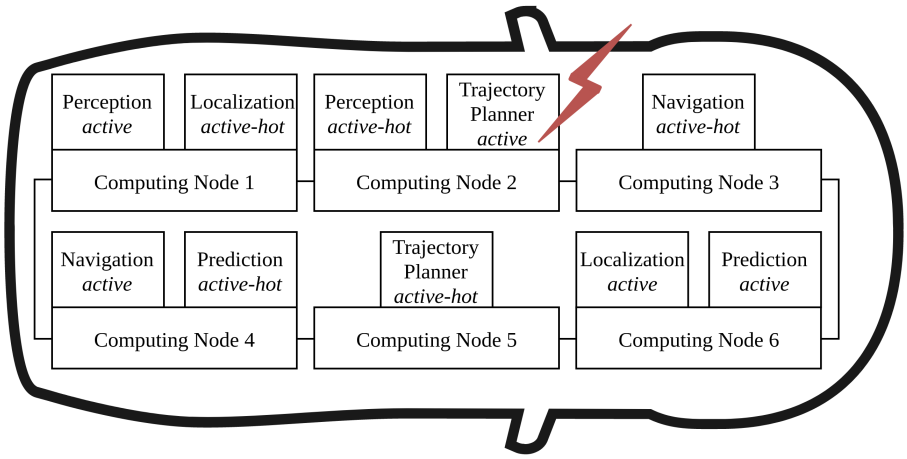

Figure 3.6: Configuration that illustrates a failure of the active instance of the active trajectory planner instance.

Next, the detected failure has to be isolated to avoid further damages. Therefore, the application instance monitor instructs the switchover component to isolate the faulty trajectory planner instance. Once the prior active instance of the trajectory planner receives the command to switch to the isolated operation mode, it stops interacting with the system.

After the failure has been isolated, the switchover component determines whether a redundant trajectory planner instance is available. In our scenario, this is the case because *Computing Node 5* executes a trajectory planner instance which runs in active-hot operation mode, so the switchover component selects this instance to upgrade its operation mode to active. The actions performed by the switchover component correspond to the second step of the FDIRO procedure. The results of the executed actions are illustrated in Figure 3.7.

Due to the isolation and the switchover, the trajectory planner is not executed redundantly anymore. Therefore, in the third step of the FDIRO process, the redundancy-recovery component tries to recover the lost redundancy. This component examines whether any of the computing nodes offers enough free resources to execute a trajectory planner instance. Assuming *Computing Node 3* and *Computing Node 5* are the only two nodes offering enough resources for an additional trajectory planner instance, the redundancy-recovery component will select *Computing Node 3* for executing the redundant instance of the trajectory planner. *Computing Node 3* is preferred over *Computing Node 5* since the latter already executes the active instance of the trajectory planner. Therefore, selecting *Computing Node 3* increases the hardware segregation, which results in an improvement of reliability.

The operation mode selected for the new trajectory planner instance is the same as the previous operation mode of the now active instance. Therefore, the redundancy

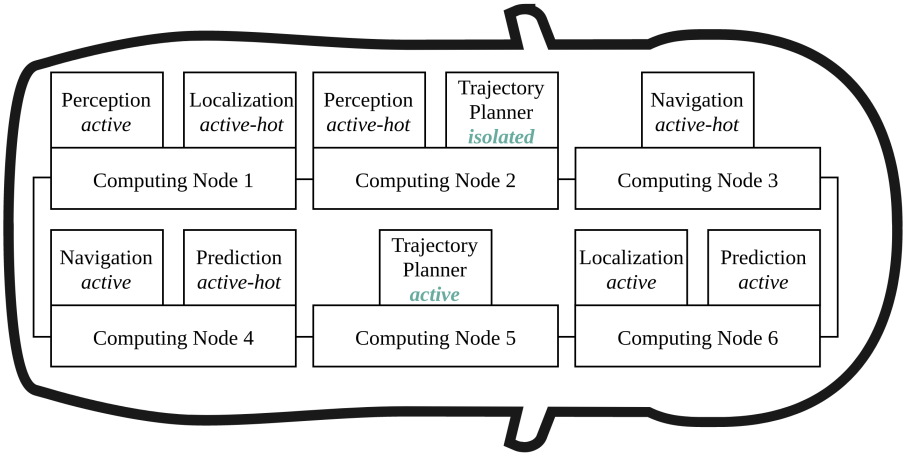

Figure 3.7: Configuration after the isolation/switchover step of FDIRO was performed.

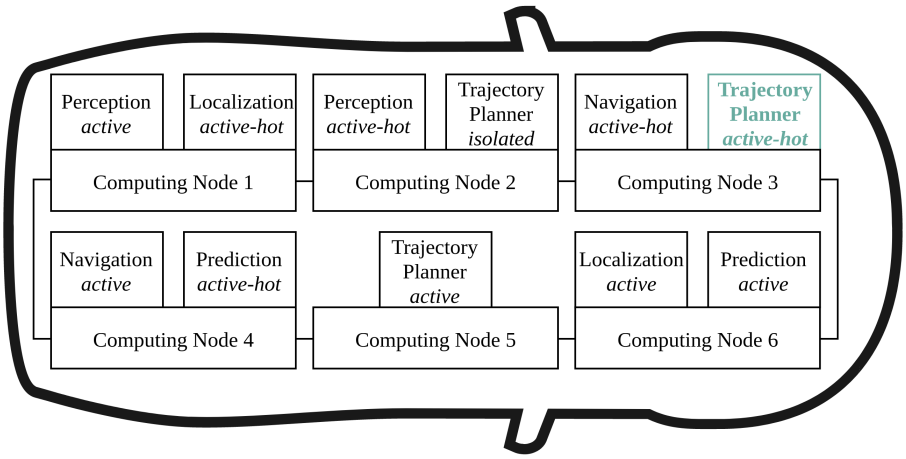

Figure 3.8: Configuration after the redundancy recovery step of FDIRO was performed.

conditions comply with the conditions obtained before the occurrence of the failure. The resulting configuration is illustrated in Figure 3.8.

In the last step of the FDIRO process, control is handed over to the placement optimizer. The task of this unit is to optimize the mapping between computing nodes and application instances according to specified goals. Assuming that the goal in the currently considered use case is to utilize as few as possible computing nodes in order to save energy, the placement optimizer might decide to stop the isolated trajectory planner instance and migrate the active-hot instance of the perception application to *Computing Node 5*. As a result, *Computing Node 2* can be turned off, since this node executes no applications anymore. The optimized application placement is displayed in Figure 3.9.

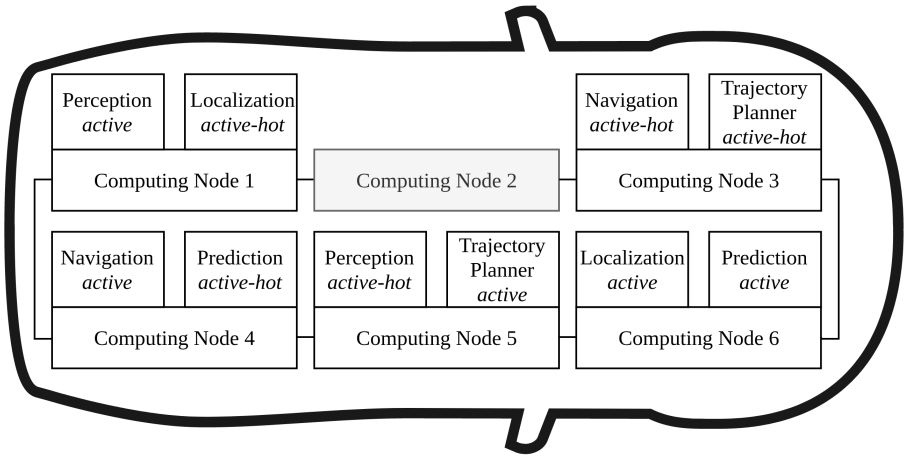

Figure 3.9: Configuration after the optimization step of the FDIRO was performed.

CHAPTER 4

Isolation and Switchover Procedure

After a monitor has detected a failure, the FDIRO procedure defines that the operation mode of the faulty application instance is set to isolated. Therefore, the effects of the failure can be limited. Furthermore, to recover the lost functionality, FDIRO specifies that a redundant instance upgrades its operation to active. After this switchover, the system state is again considered to be safe. Therefore, the isolation and switchover procedure has to be performed fast. To show that it is feasible to perform the isolation and switchover procedure within milliseconds, we present a proof-of-concept implementation for it.[1]

4.1 Switchover Strategies

As mentioned before, the switchover procedure instructs a redundant instance, if existing, to take over the responsibilities of the failed instance. Therefore, the following events can be treated:

- a failure of an active instance, whereby at least one active-hot or passive-warm instance has to exist,
- a failure of a passive-hot instance, whereby at least one passive-warm instance has to exist,
- a failure of a runtime environment, whereby for each instance running on the faulty runtime environment, there has to be at least one instance running on another runtime environment, and

[1] The source codes of all implementations mentioned in the sequel are available upon request from the author (email: `tobias.kain@outlook.com`).

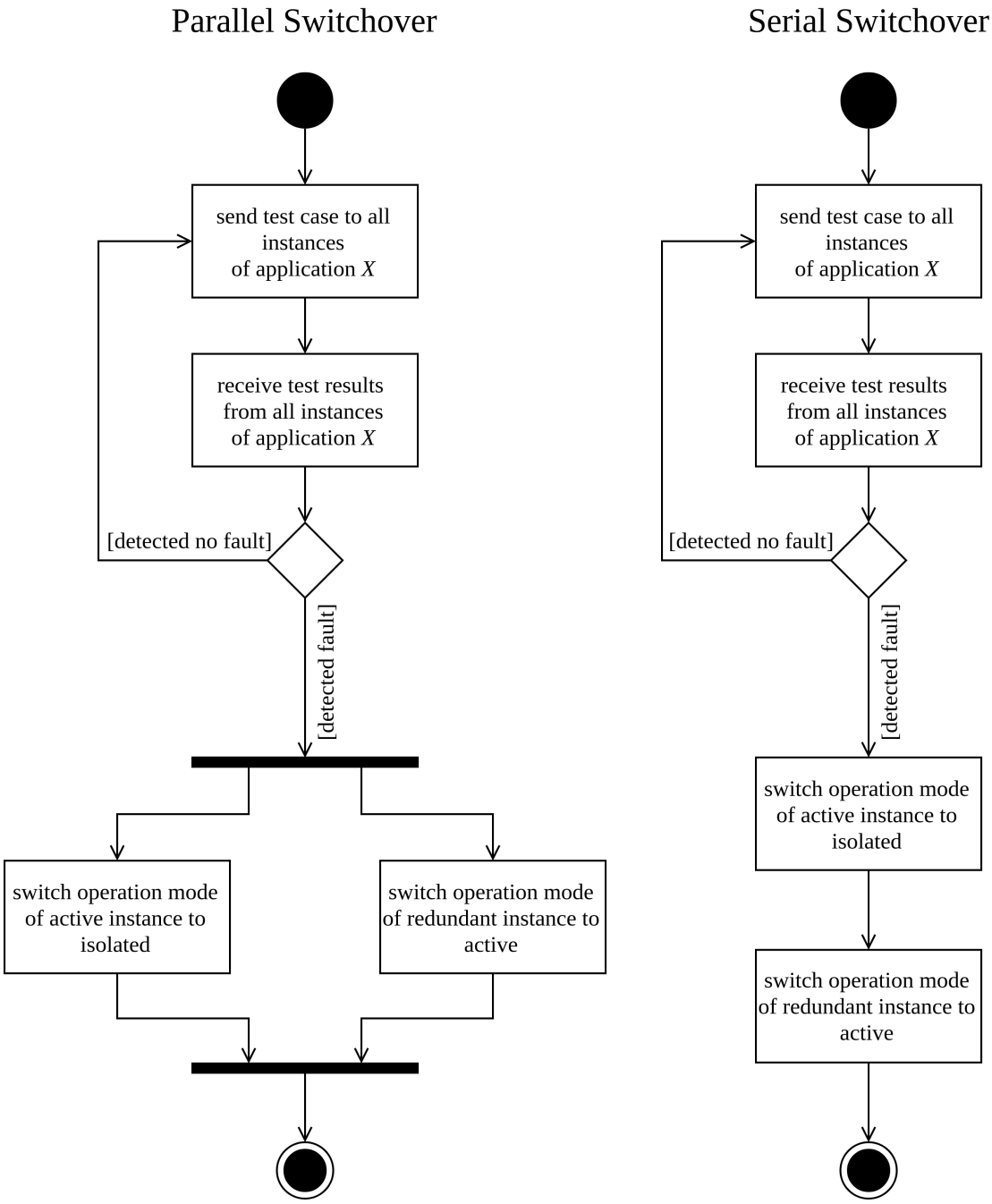

Figure 4.1: Activity diagram of the parallel and the serial switchover procedure.

- a failure of a computing node, whereby for each instance running on the faulty computing node, there has to be at least one instance running on another computing node.

In case that the switchover component receives a request to perform a switchover procedure for an application for which no redundant instance exists, the failure can be isolated by updating the operation mode of the active instance to isolated. However, since no redundant instances exist, the switchover component cannot recover the functionality that was lost due to the failure. In such a case, the switchover component has to determine the safety critically of that application and decide whether a takeover by the fail-safe system is necessary.

On the other hand, in case a redundant instance exits and the active instance fails, we can distinguish between a *parallel* and a *serial switchover strategy*, as illustrated in Figure 4.1.

The benefit of the parallel switchover procedure is that the takeover time can be reduced since the switchover instructions sent to the active instance and commands sent to one of the redundant instances are executed in parallel. However, the problem is that, for a short time, two active instances of the same application might exist. As a result, other components of the system, e.g., actuators, may receive correct as well as corrupted information.

On the other hand, the advantage of the serial switchover procedure is that at each point in time, at most one active instance of the same application is executed. However, the takeover time might be longer since additional communication between the switchover component and the faulty active component is required.

Each switchover strategy has its own benefits and drawbacks, but it depends on a specific situation which strategy may be preferred. It is also conceivable that the preferred switchover procedure might depend on the application and shall, therefore, be defined at the application level.

Note that for all other events apart from failures of active instances, the parallel switchover procedure is the one to prefer because active-hot and passive-warm instances do not interact with the system. Therefore, it is not a problem if two active-hot or two passive-warm instances are executed simultaneously.

In what follows, we only consider the parallel switchover procedure.

4.2 Switchover Test Implementation

To prove our hypothesis that the switchover procedure can be performed within milliseconds, we performed an experiment using a proof-of-concept implementation.

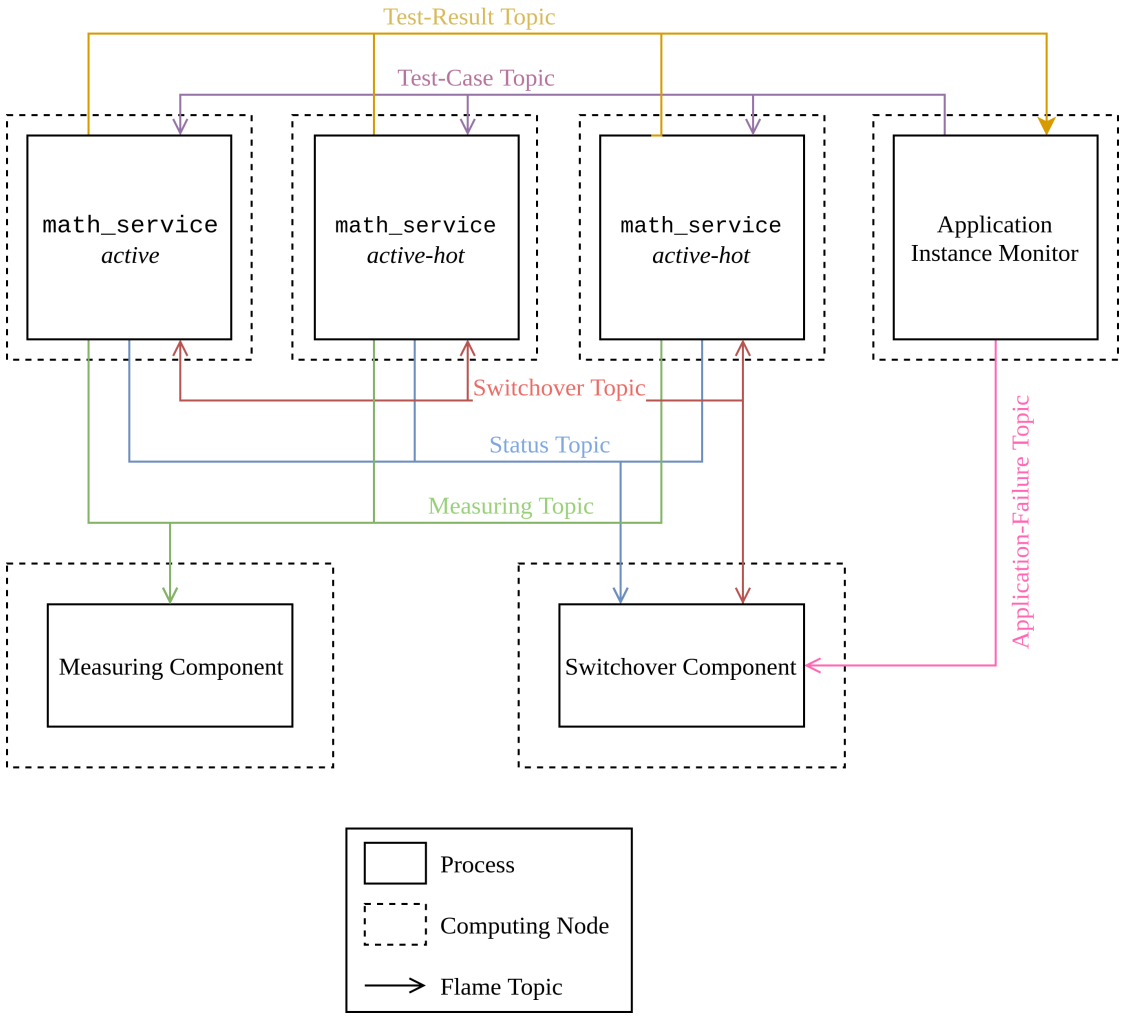

Figure 4.2: Setup of the switchover experiment.

4.2.1 Test Setup

For the test setup, we used six identical computing nodes (with OS Ubuntu 18.04.1 LTS, CPU Intel Pentium Silver J5005, and 8 GB memory). Three computing nodes execute an application that simulates a safety-critical function, whereby one computing node runs the active instance, and the other two execute an active-hot instance. The remaining three computing nodes execute an application-instance monitor, a switchover component, and a *measuring component*. The setup of our experiment is illustrated in Figure 4.2.

4.2.2 Component Communication

Physically, the computing nodes are connected using an Ethernet with 1 gigabit/s bandwidth. FLAME ("Flexible Automotive Communication Middleware"), a communication

middleware developed by Volkswagen Group Research [59], is used for the communication between the different components. Using FLAME, we define communication channels, referred to as *topics*, between multiple components.

The messages, referred to as *objects*, that are exchanged via the topics are defined in a GIGO file, whereby GIGO ("Generic Interface Language Compiler") is a middleware-independent interface definition language (IDL), which allows a formal description of programming interfaces [59]. The following listing shows the GIGO file that defines the objects exchanged in the switchover experiment:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Service name="SwitchoverService" version="0.0"
 xmlns="http://www.volkswagenag.com/6160/service/v1">
  <Object name="TestCase">
    <Elements>
      <Integer name="Id" width="16"/>
      <Integer name="FirstSummand" width="16"/>
      <Integer name="SecondSummand" width="16"/>
    </Elements>
  </Object>
  <Object name="TestResult">
    <Elements>
      <Integer name="TestCaseId" width="16"/>
      <Integer name="Sum" width="16"/>
      <Struct name="ApplicationInstanceInformation"
       type="ApplicationInstanceInformation"/>
    </Elements>
  </Object>
  <Object name="ApplicationInstanceInformation">
    <Elements>
      <Enumeration name="OperationMode" type="OperationMode"/>
      <String name="ApplicationInstanceId"/>
      <String name="ApplicationId"/>
    </Elements>
  </Object>
  <Object name="SwitchOverCommand">
    <Elements>
      <Struct name="ApplicationInstanceInformation"
       type="ApplicationInstanceInformation"/>
      <Enumeration name="NewOperationMode" type="OperationMode"/>
    </Elements>
  </Object>
  <Object name="ApplicationInstanceFail">
    <Elements>
      <Struct name="ApplicationInstanceInformation"
       type="ApplicationInstanceInformation"/>
    </Elements>
  </Object>
  <Enumeration name="OperationMode">
    <Members>
      <Member id="0" name="ACTIVE"/>
      <Member id="1" name="ACTIVE_HOT"/>
      <Member id="2" name="PASSIVE_HOT"/>
```

```
      <Member id="3" name="ISOLATED"/>                    44
    </Members>                                            45
  </Enumeration>                                          46
</Service>                                                47
```

4.2.3 Test Workflow

The application we used for our experiment is a simple service that can add two integers called `math_service`. To test the functionality of this application, the application-instance monitor sends every n milliseconds a test case containing two random integers to all three application instances via the *test-case topic*. Once an application instance receives a test case, it adds the two integers and returns the result via the *test-result topic* to the application-instance monitor. The latter then waits until it has received the test results of all application instances. In case that the test result returned by the active instance is different from the results returned by the active-hot instances and the results returned by the two active-hot instances are equal to each other, the application-instance monitor reports to the switchover component using the *application-failure topic* that the active instance of the `math_service` is not working correctly anymore. As soon as the switchover component receives such a message, it instructs the faulty active instance via the *switchover topic* to switch to the isolated operation mode. Since the application instances continuously publish the status information, including the operation mode, every second on the *status topic*, the switchover component has knowledge about all available application instances. Therefore, the switchover component can quickly select an active-hot instance that shall switch to active mode. Since the goal of this experiment is to perform the switchover as fast as possible, the switchover component randomly selects an active-hot instance for the switchover. It then instructs the selected active-hot instance via the switchover topic to switch to the active operation mode.

The measuring component simulates the interaction of the `math_service` with the rest of the system. The active instance of the `math_service` sends every millisecond a message to the measuring component. This message also includes the instance indicator of the currently active instance as well as a flag that signals whether the active instance is working correctly or not. Using the receiving times of these messages, the measuring component determines how long the switchover took. Note that the measuring component actually measures the takeover time. However, since the active instance of the `math_service` sends a message every millisecond to the measuring component, the *switchover time* and the *takeover time* can be considered as equal.

An overview of the performed procedure is illustrated in Figure 4.3.

All components involved in this experiment are implemented in C++.

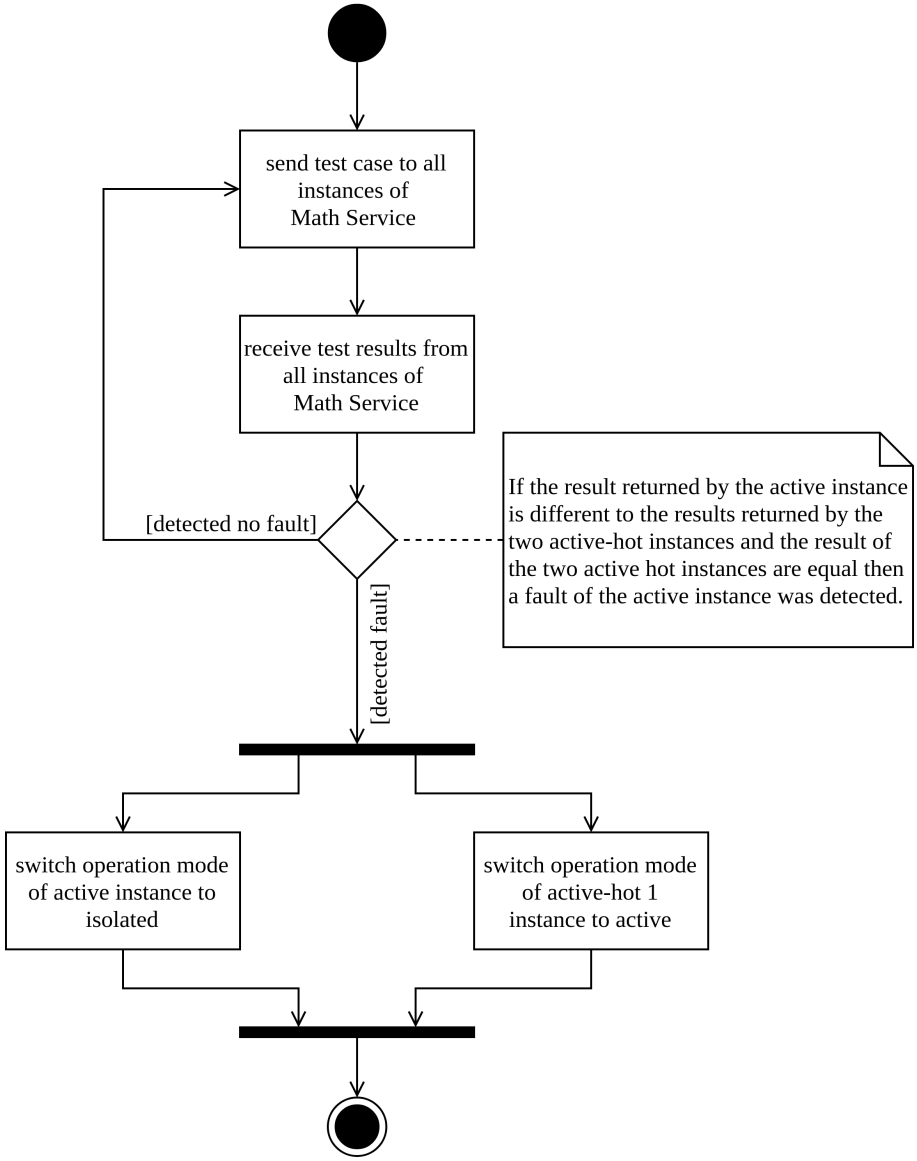

Figure 4.3: Activity diagram of the switchover lab experiment.

4.2.4 Test Results

As the goal of this experiment is to show that a switchover can be performed within milliseconds, we equipped the `math_service` with a function that allows us to corrupt the active instance on purpose. We then measure the switchover time as well as the *switchover time of the faulty instance*. The difference between those two times is illustrated in Figure 4.4.

Since the switchover time strongly depends on the *failure detection time*, we varied the frequency at which the application-instance monitor sends test cases, referred to as *failure*

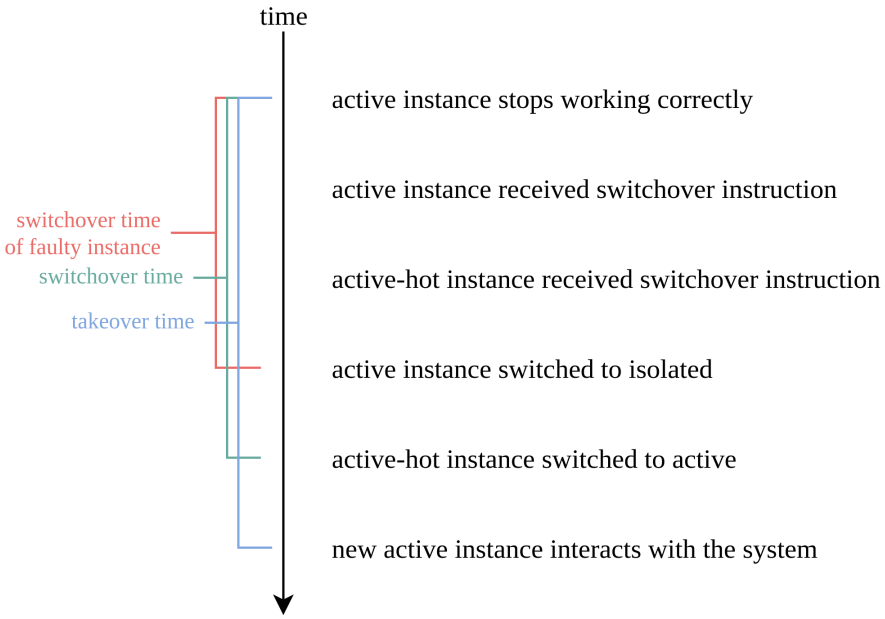

Figure 4.4: Visualization of the order of the events during the switchover experiment.

Table 4.1: The test results of the switchover experiment.

Failure Detection Sampling Rate	Average Switchover Time of Faulty Instance	Average Takeover Time
5 ms	6.8 ms	8.5 ms
10 ms	9.4 ms	14.4 ms
15 ms	11.8 ms	16.2 ms
20 ms	14.5 ms	16.3 ms
25 ms	17.7 ms	19.3 ms
30 ms	21.8 ms	22.4 ms
35 ms	19.4 ms	21.2 ms
40 ms	25.7 ms	30.0 ms
45 ms	26.8 ms	27.2 ms
50 ms	34.8 ms	36.4 ms

detection sampling rate, to the application instances. Table 4.1, shows the results of the performed tests. The average switchover time and the average switchover time of the faulty instance are determined based on 100 test iterations.

From the measurements shown in Table 4.1, we can infer that the latency between the average switchover time of the faulty instance and the average takeover time is approximately 2 ms.

Furthermore, we can observe that, on average, the switchover time of the faulty instance

is about half of the failure detection sampling rate plus an offset of approximately 5 ms. Assuming that time consumed by the application-instance monitor and the time consumed by the switchover controller as well as the latency of the network is 0 ms, the average switchover time of the failed instance will converge to half of the failure detection sampling rate as the number of test runs increases. Therefore we can conclude that the time consumed by the application-instance monitor plus the time consumed by the switchover controller plus the latency of the network sums up to is approximately 5 ms.

CHAPTER 5

Redundancy Recovery

The redundancy recovery procedure is the third step of the FDIRO process. After the switchover component has isolated the failure that was detected by a monitor, and a switchover to a redundant instance recovered the lost functionality, the control is passed on to the redundancy-recovery component. The task of this component is to recover the redundancy condition which held before the occurrence of the failure. To recover the redundancy, the redundancy-recovery component has to find a computing node capable of running an instance of the application affected by the failure. Depending on the parameters considered and the current application placement, this can be a challenging task.

The first section of this chapter is dedicated to defining the problem of finding a mapping between application instances and computing nodes. Afterwards, we introduce a software architecture as well as an implementation for performing the redundancy recovery. Furthermore, we present a testing tool for validating the results of our implementation.

5.1 Application-Placement Problem

The task of determining the assignment between application instances and computing nodes is referred to as the *application-placement problem*. In our setting, the input of this problem is a set I of application instances and a set N of computing nodes and the task is to find a function C, called *configuration*, that maps each instance $i \in I$ to exactly one node $n \in N$ such that certain constraints, 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., maximizing the number of computing nodes that execute no application instances, maximizing the redundancy of a specific class of applications, or minimizing the number of displacements after the occurrence of a failure.

The constraint conditions of the application-placement problem that we consider here are defined in terms of the following parameters for each application:

- the memory demand,
- the CPU demand,
- the software requirements,
- the level of hardware segregation,
- the level of redundancy, and
- the priority.

Furthermore, for each computing node, we specify

- the memory capacity,
- the CPU capacity, and
- the installed software.

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

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

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

(C_3) The sum of CPU demands of all the application instances running on a computing node cannot exceed the CPU capacity of that node.

(C_4) An application instance only runs on a computing node, which offers the software required by the instance.

(C_5) The instances that belong to the same application have to run on at least a certain number of distinct computing nodes, i.e., the level of hardware segregation has to be satisfied for each application.

In general, solving the application-placement problem is a non-trivial task since, as follows from results of Tang, Steinder, Spreitzer, and Pacifici [58], the *class constrained multiple-knapsack problem* [56], which is NP-hard, can be reduced to the application-placement problem. Therefore, determining whether the application-placement problem has a solution is NP-hard too.

5.2 Software Architecture

For implementing the redundancy recovery step of the FDIRO procedure, we designed a software architecture consisting of several individual components. It allows for dividing the responsibilities of the different components and thus keeps their complexity reasonable. Furthermore, adaptations of the individual components are in that way easier to maintain.

Overall, we define five different components:

- a *current-state reporter*,
- a *current-state determiner*,
- an *application-placement determiner*,
- a *reconfiguration-plan advertiser*, and
- a *reconfiguration-plan executor*.

These components can be divided into two subsystems: The current-state reporter and the reconfiguration-plan executor belong to the *reconfiguration unit*, which is part of each computing node installed in a vehicle, while the remaining components build up the *redundancy-recovery component*. Figure 5.1 illustrates these two subsystems as well as the subsystem that is responsible for performing the isolation step of the FDIRO procedure.

As mentioned before, the current-state reporter is part of the reconfiguration unit and is therefore executed by each computing node. The task of this component is to report the current state of the node to the redundancy-recovery component. This includes information about the available resources and information about the applications that are currently executed by this computing node.

The information provided by the current-state reporter is used by the current-state determiner to obtain the current state of the entire system, containing knowledge about all available computing nodes, the placement of the application instances, as well as the requirements of the applications.

The current system state is required by the application-placement determiner to compute a new application placement, which restores the redundancy condition that held before the occurrence of the fault. The application-placement determiner is the main component and the most complex involved in the redundancy recovery step. It is also the point of entry for the redundancy recovery procedure, i.e., the switchover component instructs the application-placement determiner to start a set of application instances so that the redundancy condition is recovered.

Once the application-placement determiner computed a new placement, it forwards this plan, as well as the system state that was the basis for the determination, to the reconfiguration-plan advertiser. This component computes, based on the received inputs,

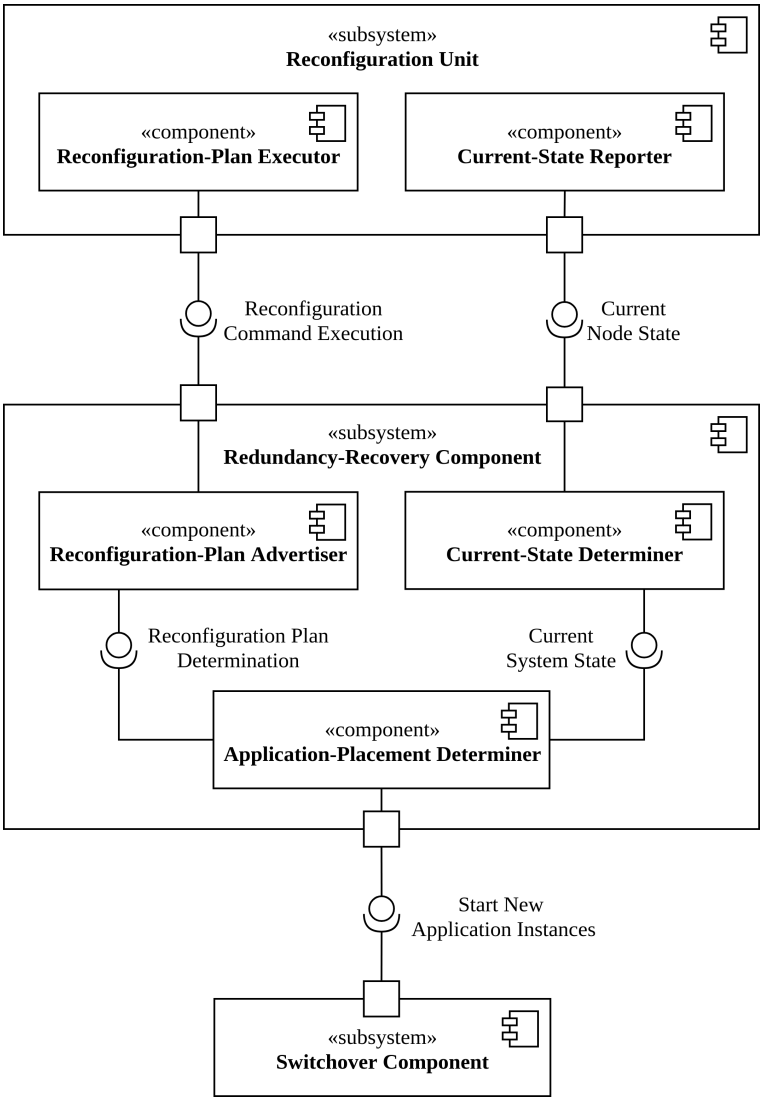

Figure 5.1: Component diagram, showing the system responsible for the redundancy recovery step.

a reconfiguration plan, i.e., a set of commands that transfers the current state to a new system state computed by the application-placement determiner. These commands instruct a computing node to either start or stop an application instance.

The components receiving those commands are the reconfiguration-plan executors. Those components are responsible for executing the reconfiguration commands computed by the reconfiguration-plan advertiser.

In the following section, we describe an implementation for the application-placement determiner.

5.3 Application-Placement Determiner

As already mentioned before, the application-placement determiner, which is part of the redundancy-recovery component, is responsible for determining the placement of application instances that recover the redundancy conditions that held before the occurrence of a failure. The application-placement determiner, therefore, attempts to find computing nodes that offer enough resources to execute the new application instances. In case of a resource shortage, the application-placement determiner may stop application instances of lower priority and displace application instances.

In this section, we introduce an implementation of an application-placement determiner, called APD, which solves the application-placement problem. The solving approach implemented by APD is *integer linear programming* [55]. The system used to specify our problem is an open-source software suite developed by Google called OR-Tools [24]. We use this tool since it supports multiple programming languages including Python, C++, and Java. Furthermore, OR-Tools supports a variety of commercial and open source solvers, including, for example, Gurobi [26], SCIP [1], and GLOP [25]. The source code of this implementation is available on request from the author.

5.3.1 Software Architecture of APD

As illustrated in Figure 5.2, the structural design of APD consists of the following three subcomponents:

- the *input preprocessor*,
- the *application-placement solver*, and
- the *solution processor*.

The input preprocessor is responsible for transforming the current state information into a valid application-placement problem definition. Based on that, the input preprocessor creates a thread of four versions (including the original one) of the given problem definition, which have different solution-space sizes and are processed in parallel. The application-placement solver then attempts to solve these four problems in parallel. Finally, the solution processor selects the most desired solution and forwards it to the reconfiguration-plan advertiser. A more detailed discussion of this parallel processing is given in Subsection 5.3.5.

5.3.2 Formalization of the Application-Placement Problem

The application-placement solver determines an application placement based on a given system state, S, which is represented by a 9-tuple of the form

$$(A, I, N, R, \Phi, \Omega, \phi, \omega, \pi),$$

whose elements are defined as follows:

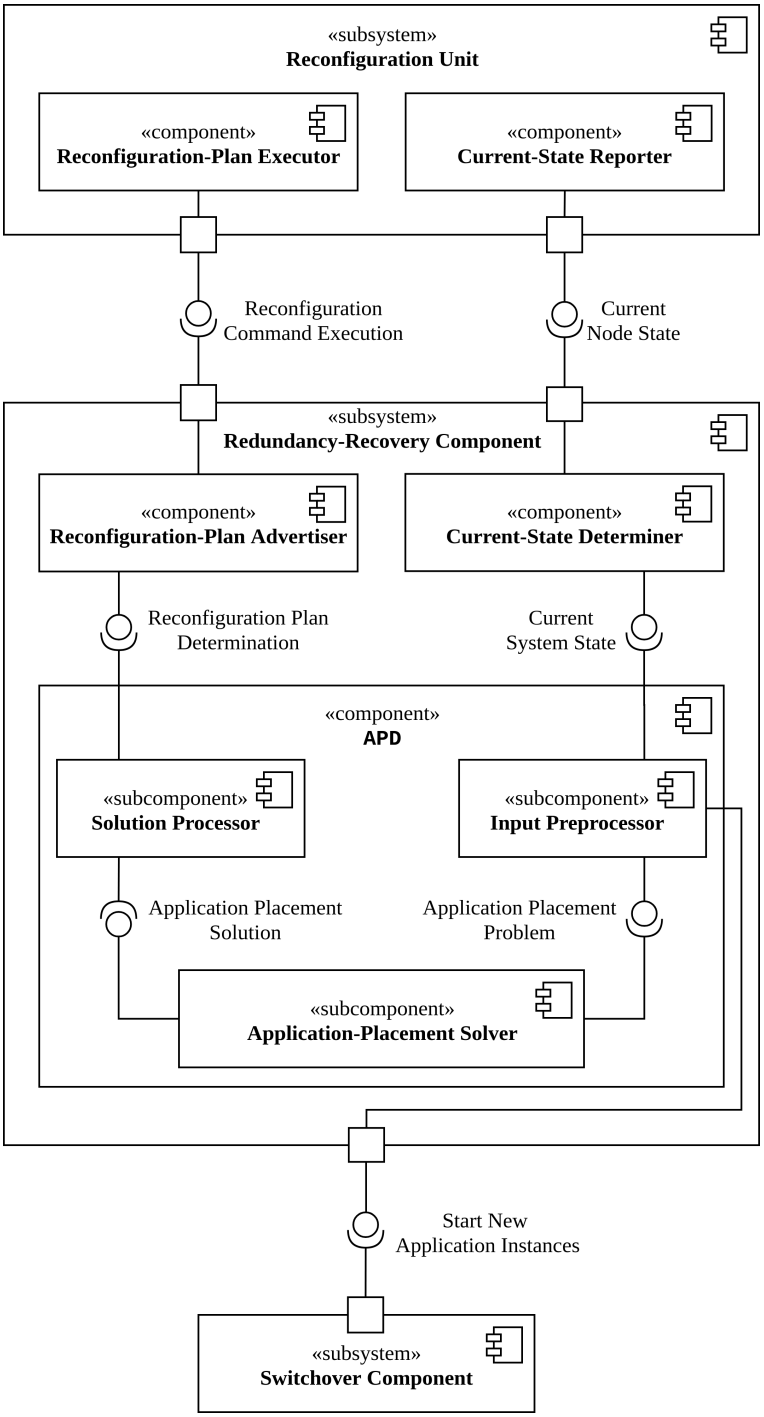

Figure 5.2: Component diagram, showing the structural design of APD.

- A is the set of applications;
- I is the set of application instances, where each application instance $i \in I$ belongs to exactly one application $a \in A$;
- 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 $i \in I$, defined by setting $R(n,i) = 1$ if n fulfills the requirements of i and $R(n,i) = 0$ otherwise (we write $R_{i,n} = R(n,i)$ 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 CPU capacity $\Omega(n) = \Omega_n$ in cycles per second;
- ϕ is the function which assigns each $i \in I$ its memory demand $\phi(i) = \phi_i$ in megabytes;
- ω is the function which assigns each $i \in I$ its CPU demand $\omega(i) = \omega_i$ in cycles per second; and
- π is the function assigning each $a \in A$ the minimum number $\pi(a) = \pi_a$ of computing nodes a has to run on, i.e., the level of hardware segregation of a.

Based on the system state S determined by the input preprocessor, the application-placement solver computes a configuration function C which specifies whether a computing node $n \in N$ executes an application instance $i \in I$ by setting $C(n,i) = 1$ if n executes i and $C(n,i) = 0$ otherwise. Similar to the representation of the placement restriction function, we will use $C_{i,n}$ to stand for $C(n,i)$.

Since all elements of the configuration function C are binary, at most $2^{|I|\cdot|N|}$ potential solutions exist. Valid solutions, however, must satisfy constraints (C_1)–(C_5) from Section 5.1, which are expressed in terms of the following linear constraints:

- Constraint (C_1):
$$\forall i \in I : \sum_{n \in N} C_{i,n} = 1. \tag{5.1}$$
- Constraint (C_2):
$$\forall n \in N : \sum_{i \in I} \phi_i \cdot C_{i,n} \leq \Phi_n. \tag{5.2}$$
- Constraint (C_3):
$$\forall n \in N : \sum_{i \in I} \omega_i \cdot C_{i,n} \leq \Omega_n. \tag{5.3}$$

- Constraint (C_4):

$$\forall i \in I, \forall n \in N: \text{ if } R_{i,n} = 0, \text{ then } C_{i,n} = 0. \tag{5.4}$$

- Constraint (C_5): To express this constraint linearly, we introduce an ancillary variable $h_{a,n}$ for each application $a \in A$ and computing node $n \in N$, which is defined as follows:

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

Using this variables, we can formalize constraint (C_5) by the following four linear expressions:

$$\forall a \in A, \forall n \in N : h_{a,n} = 0 \text{ or } h_{a,n} = 1, \tag{5.5}$$

$$\forall a \in A, \forall i \in a, \forall n \in N : C_{i,n} \leq h_{a,n}, \tag{5.6}$$

$$\forall a \in A, \forall n \in N : \sum_{i \in a} C_{i,n} \geq h_{a,n}, \tag{5.7}$$

$$\forall a \in A : \sum_{n \in N} h_{a,n} \geq \pi_a. \tag{5.8}$$

Since the modeling of (C_5) is not straightforward, we provide an example to clarify its meaning.

Example 1. Consider a system state S which defines three applications, six application instances, and four computing nodes, as represented by

$$A = \{a_1, a_2\},$$
$$I = \{i_1, i_2, i_3, i_4, i_5, i_6\}, \text{ and}$$
$$N = \{n_1, n_2, n_3, n_4\}.$$

The application instances i_1, i_2, and i_3 belong to application a_1 and the remaining instances belong to a_2, where the hardware segregation level of both applications is 3, i.e., $\pi_{a_1} = \pi_{a_2} = 3$. Furthermore, the system state S defines a configuration function C, as specified in Table 5.1.

We first consider application a_1: Since $\sum_{i \in a_1} C_{i,n_1} = 0$, (5.7) causes that $h_{a_1,n_1} = 0$. Furthermore, (5.6) yields that $h_{a_1,n_2} = h_{a_1,n_3} = h_{a_1,n_4} = 1$ since $C_{n_2,i_1} = C_{n_3,i_2} = C_{n_4,i_3} = 1$. As a result, also (5.8) holds since

$$\sum_{n \in N} h_{a_1,n} = 3 \geq \pi_a = 3.$$

Next, we show that (C_5) does not hold for application a_2: As $C_{i_4,n_1} = 1$ (note that also $C_{i_5,n_1} = C_{i_6,n_1} = 1$) and (5.6) has to hold, $h_{a_2,n_1} = 1$ holds. Since $C_{i_x,n_y} = 0$ (for $x \in$

Table 5.1: The configuration function C.

C	n_1	n_2	n_3	n_4
i_1	0	1	0	0
i_2	0	0	1	0
i_3	0	0	0	1
i_4	1	0	0	0
i_5	1	0	0	0
i_6	1	0	0	0

$\{4, 5, 6\}$ and $y \in \{2, 3, 4\}$) and (5.7) has to hold, $h_{a_2,n_2} = h_{a_2,n_3} = h_{a_2,n_4} = 0$ holds. As a result, also (5.8) does not hold since

$$\sum_{n \in N} h_{a_2,n} = 1 \not\geq \pi_a = 3.$$

□

From this example, we can conclude the following result:

Theorem 1. *Given a system state*

$$S = (I, A, N, R, \Phi, \Omega, \phi, \omega, \pi)$$

satisfying conditions (5.1)–(5.8) and a configuration function C, for any $a \in A$ and any $n \in N$, the ancillary variables $h_{a,n}$ satisfy the following two conditions:

$$h_{a,n} = 0 \text{ iff } \sum_{i \in a} C_{i,n} = 0, \tag{5.9}$$

$$h_{a,n} = 1 \text{ iff } \sum_{i \in a} C_{i,n} \geq 1. \tag{5.10}$$

Proof. We first show condition (5.9). Assume $h_{a,n} = 0$. Towards a contradiction assume $\sum_{i \in a} C_{i,n} > 0$ (since, by definition, $\sum_{i \in a} C_{i,n}$ cannot be negative, we do not have to consider the case that $\sum_{i \in a} C_{i,n} < 0$). However, $\sum_{i \in a} C_{i,n} > 0$ implies that there exists some $i_0 \in A$ and some $n_0 \in N$ such that $C_{i_0,n_0} > 0$, and therefore condition (5.6) is violated.

Conversely, assume now that $\sum_{i \in a} C_{i,n} = 0$ but $h_{a,n} = 1$ (note that condition (5.5) restricts that $h_{a,n}$ is either 0 or 1). However, in this case, condition (5.7) is violated since $0 = \sum_{i \in a} C_{i,n} < h_{a,n} = 1$.

Now we prove condition (5.10). Assume that $h_{a,n} = 1$ but $\sum_{i \in a} C_{i,n} < 1$. The latter implies that $\sum_{i \in a} C_{i,n} = 0$ must hold, which contradicts condition (5.7).

Table 5.2: Resources required by *App 1* to *App 4*.

	App 1	*App 2*	*App 3*	*App 4*
Memory Demand	500	150	150	100
CPU Demand	300	200	300	100
Required Software	x	x, y	x, z	y, z
Hardware Segregation	2	2	2	1

Table 5.3: Resources provided by *Computing Node 1* to *Computing Node 4*. Note that in this table we abbreviate "*Computing Node*" by "*CN*".

	CN 1	*CN 2*	*CN 3*	*CN 4*
Memory Demand	700	700	1200	400
CPU Demand	600	500	1000	900
Installed Software	x, z	x, y	x, y, z	x, y, z

Conversely, assume that $\sum_{i \in a} C_{i,n} \geq 1$ but $h_{a,n} = 0$ (note again that condition (5.5) restricts that $h_{a,n}$ is either 0 or 1). $\sum_{i \in a} C_{i,n} \geq 1$ implies that there is some $i_0 \in A$ such that $C_{i_0,n} > 0$. Therefore, condition (5.6) is violated. □

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:

$$\sum_{i \in I, n \in N} C_{i,n} \times \overline{C}_{i,n}.$$

Due to this optimization goal, application placements that minimize the number of application instance displacements are preferred. We aim for a low number of application instance displacements since displacements are considered to be time-consuming.

In what follows, we provide an example that illustrates the defined constraints as well as the optimization goal implemented by `APD`.

5.3.3 `APD` Use Case

Assume a system configuration comprising four applications, *App 1* to *App 4*, whereby the required resources are specified in Table 5.2, as well as four computing nodes, *Computing Node 1* to *Computing Node 4*, whereby the resources provided by these computing nodes are defined in Table 5.3.

The initial application placement that we consider for this example is illustrated in Figure 5.3. It is easy to verify that this application placement fulfills all the requirements requested by the previously defined constraints.

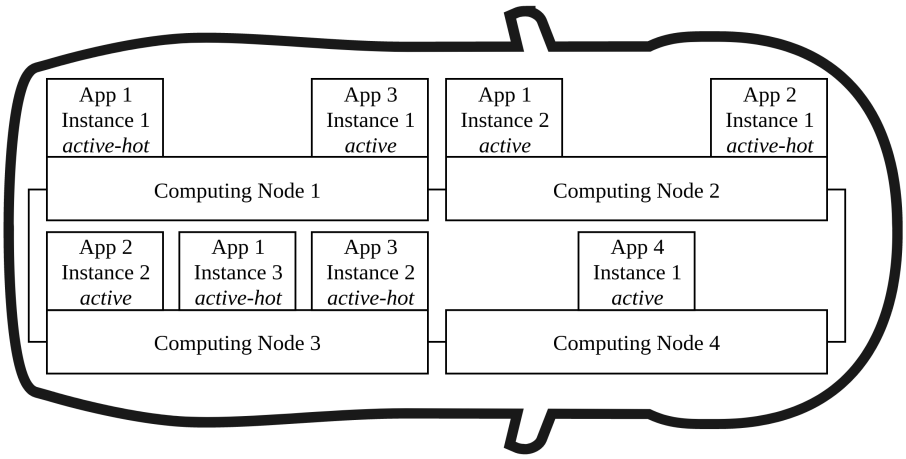

Figure 5.3: The initial application placement.

Assume now that *Computing Node 2* fails. Consequently, also the active instance of *App 1* and the active-hot instance of *App 2* fails. Due to the loss of the active instance of *App 1*, the service provided by this application is no longer available. Recall that in such a case, the switchover component instructs a redundant instance of the affected application to upgrade its operation mode to active.

Once this switchover has been performed successfully, the switchover component instructs `APD` to start a new active-hot instance of *App 1* and *App 2* in order to recover the redundancy condition that held before the crash of *Computing Node 2*.

Based on the currently active application placement as well as on the specified application and computing node parameters, `APD` computes a new application placement, which causes the fewest number of displacement of already placed application instances. For our example, it is required to displace at least one application instance to fulfill all requirements defined by the constraints.

The resulting application placement is illustrated in Figure 5.4. This application placement requires the displacement of the active-hot instance of *App 3*. As stated before, by construction, our example requires the displacement of at least one application instance. Therefore, the application placement illustrated in Figure 5.4 is considered as an optimal solution according to the optimization goal defined by `APD`.

Besides this solution, multiple other valid application placements exist. However, according to our optimization goal, they are not considered as optimal solutions. Figure 5.5 illustrates, for example, an application placement that causes the displacements of five application instances.

Although for this specific scenario, multiple solutions exist, there are many use cases for which no solution can be determined. However, in such cases, as defined before, the application-placement determiner may stop application instances of low priority.

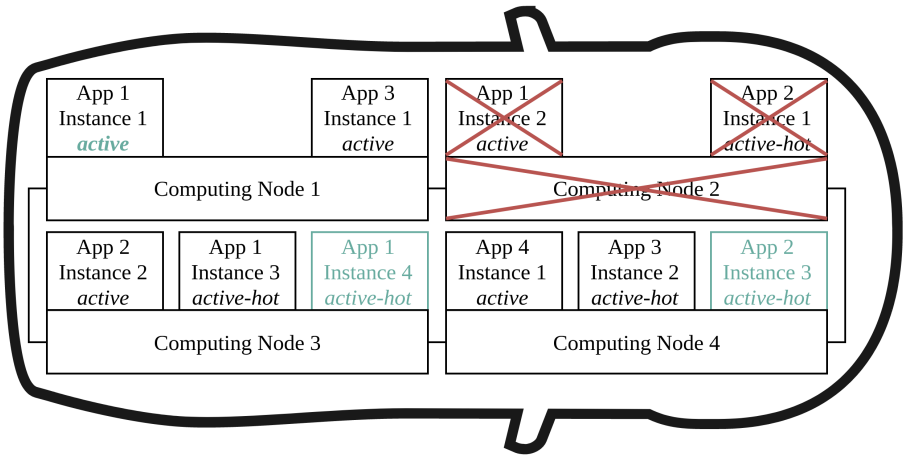

Figure 5.4: Application placement computed by `APD`.

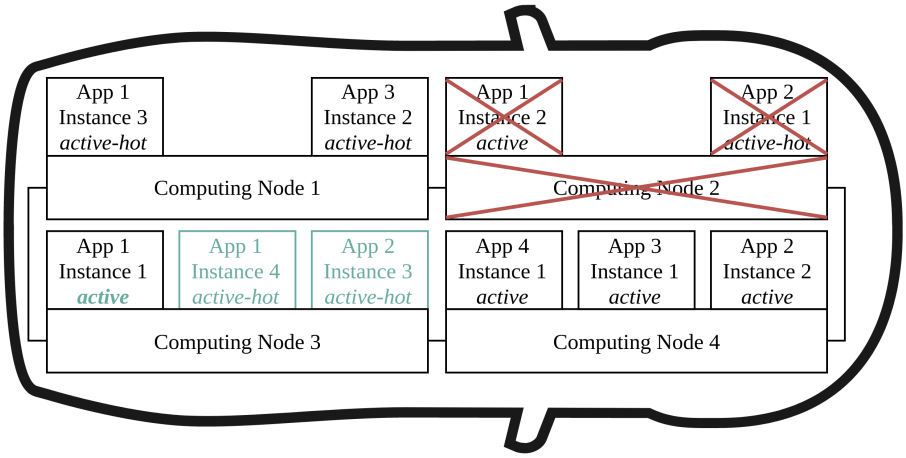

Figure 5.5: Valid solution of the application-placement problem which results in five displacements of application instances.

5.3.4 Core Function of `APD`

`APD` is a Python program to solve the application-placement problem. As mentioned before, `APD` is divided into the following three subcomponents: the input preprocessor, the application-placement solver, and the solution processor.

In the following, we focus on the implementation of the application-placement solver, whereby the `compute_application_placement` function is considered as the core of this subcomponent. This function uses OR-Tools to express and solve the application-placement problem, as described in Section 5.3. The source code of it is illustrated in the following listing and discussed in the remainder of this section.[1]

[1]Note that for the sake of simplicity, the listed code does not exactly correspond to the actual code.

```
def compute_application_placement(current_placement, applications,
                                  hardware_segregations, cpu_capacities,
                                  cpu_demands, memory_capacities,
                                  memory_demands, software_restrictions):

   solver = ortools.linear_solver.pywraplp.Solver('Solver',
                     pywraplp.Solver.BOP_INTEGER_PROGRAMMING)

   number_of_apps = len(applications)
   number_of_app_instances = len(current_placement)
   number_of_nodes = len(current_placement[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_nodes)]
                    for i in range(number_of_app_instances)]

   new_placement_trans = [[new_placement[i][j]
                           for i in range(number_of_app_instances)]
                          for j in range(number_of_nodes)]

   hardware_segregations_helper = [[solver.BoolVar("a%i n%i" % (a, n))
                                    for n in range(number_of_nodes)]
                                   for a in range(number_of_apps)]

   '''
   Constraint 1: An application instance has to be executed by exactly one
                 computing node.
   '''
   for i in range(number_of_app_instances):
      solver.Add(solver.Sum(new_placement[i]) == 1)

   '''
   Constraint 2: The sum of memory demands of all the application
                 instances running on a computing node cannot
                 exceed the memory capacity of that node.
   '''
   for n in range(number_of_nodes):
      solver.Add(
         solver.Sum([new_placement_trans[n][i] * memory_demands[i]
                     for i in range(number_of_app_instances)]) <=
         memory_capacities[n])

   '''
   Constraint 3: The sum of CPU demands of all the application instances
                 running on a computing node cannot exceed the CPU
                 capacity of that node:
   '''
   for n in range(number_of_nodes):
      solver.Add(
         solver.Sum([new_placement_trans[n][i] * cpu_demands[i]
                     for i in range(number_of_app_instances)]) <=
         cpu_capacities[n])
```

```
    '''
    Constraint 4: An application instance only runs on a computing node,
                  which offers the software required by the instance.
    '''
    for i in range(number_of_app_instances):
        for n in range(number_of_nodes):
            if software_restrictions[i][n] == 0:
                solver.Add(new_placement[i][n] == 0)

    '''
    Constraint 5: The level of hardware segregation has to be satisfied for
                  each application.
    '''
    for a in range(number_of_apps):
        for n in range(number_of_nodes):

            for i in applications[a]:
                solver.Add(new_placement[i][n] <=
                    hardware_segregations_helper[a][n])

            solver.Add(
                solver.Sum([
                    new_placement[i][n]
                    for i in applications[a]
                ]) >= hardware_segregations_helper[a][n]
            )

        solver.Add(
            solver.Sum(hardware_segregations_helper[a]) >=
                        hardware_segregations[a]
        )

    '''
    Optimization Goal: Minimize the number of application instance
                       displacements.
    '''
    solver.Maximize(solver.Sum([
        new_placement[i][n] * current_placement[i][n]
        for i in range(number_of_app_instances)
        for n in range(number_of_nodes)]))

    result_status = solver.Solve()

    if result_status == 2:
        print("No solution exists.")
        return None

    solution = [[new_placement[i][n].solution_value()
                 for n in range(number_of_nodes)]
                for i in range(number_of_app_instances)]

    return solution
```

Table 5.4: The `current_placement` parameter. Note that in this table we abbreviate "*Application Instance*" by "*AI*".

`current_placement`	*CN 1*	*CN 2*	*CN 4*
AI 1	1	0	0
AI 2	1	0	0
AI 3	0	1	0
AI 4	0	1	0
AI 5	0	1	0
AI 6	0	0	1
AI 7	0	0	0
AI 8	0	0	0

The `compute_application_placement` function requires as input the eight parameters:

- `current_placement`,
- `applications`,
- `hardware_segregations`,
- `memory_capacities`,
- `cpu_capacities`,
- `memory_demands`,
- `cpu_demands`, and
- `software_restrictions`.

For the example introduced in Subsection 5.3.3, the `current_placement` parameter is defined as shown in Table 5.4.

The definition of the correspondence between application instances and applications is defined by the `applications` parameter. This parameter is required in order to define the `hardware_segregation` parameter, which defines the desired level of hardware segregation for each application. Table 5.5 and Table 5.6 specify for the considered example which instance belongs to which application and the hardware segregation of the applications, respectively.

The memory and CPU resources provided by the computing nodes are defined by the `memory_capacities` and the `cpu_capacities` parameter, respectively. The values of those parameters are specified in Table 5.7.

Table 5.5: The `applications` parameter.

AI 1	*App 1*
AI 2	*App 3*
AI 3	*App 2*
AI 4	*App 1*
AI 5	*App 3*
AI 6	*App 4*
AI 7	*App 1*
AI 8	*App 2*

Table 5.6: The `hardware_segregation` parameter.

	App 1	*App 2*	*App 3*	*App4*
`hardware_segregation`	2	2	2	1

Table 5.7: The `memory_capacities` and `cpu_capacities` parameter.

	CN 1	*CN 2*	*CN 4*
`memory_capacities`	1000	1200	400
`cpu_capacities`	1000	1000	900

Table 5.8: The `memory_demands` and `cpu_demands` parameter.

	AI 1	*AI 2*	*AI 3*	*AI 4*	*AI 5*	*AI 6*	*AI 7*	*AI 8*
`memory_demands`	500	150	150	500	150	100	500	150
`cpu_demands`	300	300	200	300	30	200	300	200

Corresponding to those parameters, we specify the memory and CPU demands of the application instances by defining the `memory_demands` and the `cpu_demands` parameter. Table 5.8 specifies those parameters.

The `software_restrictions` parameter indicates whether a computing node fulfills all software requirements requested by an application instance. This function is specified in Table 5.9.

After defining all required input variables, the `compute_application_placement` 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_placement`

Table 5.9: The `software_restrictions` variable.

`software_restrictions`	*CN 1*	*CN 2*	*CN 4*
AI 1	1	1	1
AI 2	1	1	1
AI 3	0	1	1
AI 4	1	1	1
AI 5	1	1	1
AI 6	0	1	1
AI 7	1	1	1
AI 8	0	1	1

parameter and declares the variables that after solving the application problem correspond to the determined solution. Furthermore, we also define `current_placement_trans`, which is the transpose of the `current_placement` parameter.

Besides these variables, the solver also has to determine the variables defined by the `hardware_segregations_helper` parameter, whereby those variables indicate whether any instance of an application is executed by a specific computing node. Due to those helper variables, constraint (C_5), which is defined in Section 5.1, can be expressed linearly.

Note that the sum of the number of application instances and the number of applications multiplied by the number of computing nodes corresponds to the total number of variables t that the solver has to determine:

$$t = \texttt{number\_of\_nodes} \cdot (\texttt{number\_of\_app\_instances} + \texttt{number\_of\_apps}).$$

Since all those variables are boolean variables, the size of the solution space is 2^t. The solution space for our example, which defines eight application instances, four applications, and three computing nodes is therefore $2^{3\cdot(8+4)} = 2^{36} \approx 68 \cdot 10^9$.

To restrict the solution space, we add in lines 30 to 84 the constraints introduced in Section 5.1. Note that we only use linear operations to define those constraints.

Next, we specify in lines 90 to 93 the optimization goal, whereby the goal implemented by `APD` is to find a placement solution that differs the least from the current placement.

Finally, in lines 95 to 105, we instruct the solver to solve the specified problem and return the solution in case one exists. For our example, the `solution` placement computed by the solver is illustrated in Table 5.10.

Note that this placement corresponds to the application placement as illustrated in Figure 5.4 and causes the displacement of one application instance.

Table 5.10: The `solution` variable.

`solution`	*CN 1*	*CN 2*	*CN 4*
AI 1	1	0	0
AI 2	1	0	0
AI 3	0	1	0
AI 4	0	1	0
AI 5	0	0	1
AI 6	0	0	1
AI 7	0	1	0
AI 8	0	0	1

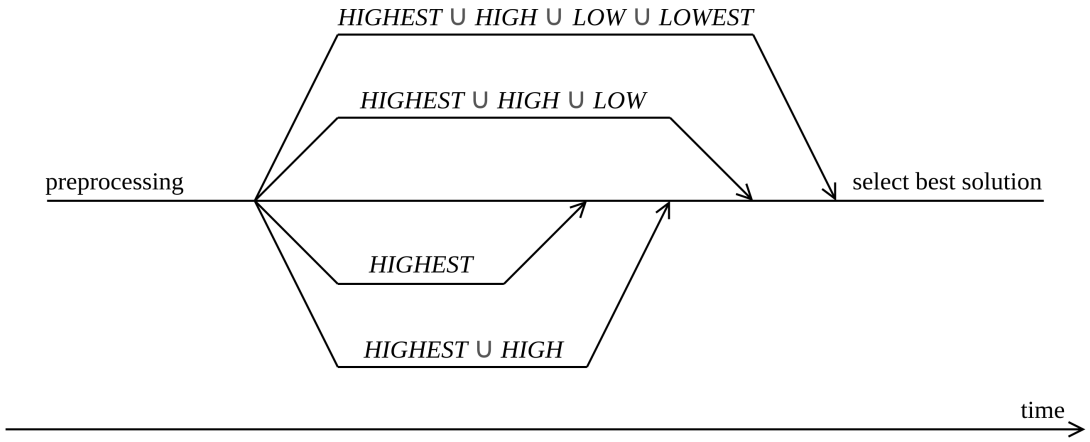

Figure 5.6: Multiple threads for determining the application-placement problem for subsets of applications.

5.3.5 Parallel-Solving Heuristic

To increase the chances of obtaining a valid solution, we divide a given set A of applications into several subsets. `APD` computes these subsets based on priorities in terms of the following four priority classes: *HIGHEST*, *HIGH*, *LOW*, and *LOWEST*.

Using these priority classes, `APD` computes the following subsets of applications:

$$A_0 = A = HIGHEST \cup HIGH \cup LOW \cup LOWEST,$$
$$A_1 = HIGHEST \cup HIGH \cup LOW,$$
$$A_2 = HIGHEST \cup HIGH, \text{ and}$$
$$A_3 = HIGHEST.$$

For each of those four subsets, `APD` starts a thread that computes an application placement that considers all applications that are part of the respective subset. Since $A_3 \subseteq A_2 \subseteq$

Table 5.11: Priorities of applications *App 1* to *App 4*.

	App 1	*App 2*	*App 3*	*App 4*
Priorities	*LOW*	*HIGHEST*	*HIGH*	*HIGH*

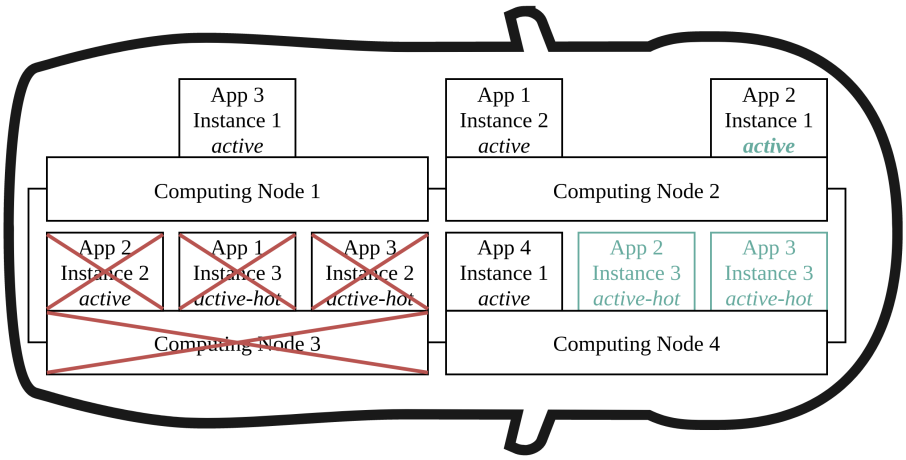

Figure 5.7: Application placement that considers only applications of priority *HIGHEST* and *HIGH*, i.e., *App 2*, *App 3*, and *App 4*.

$A_1 \subseteq A_0 = A$ holds, we can assume, as illustrated in Figure 5.6, that the time required to find a valid application placement is highest for A and lowest for A_3. 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, `APD` selects the best available solution, whereby an application placement that considers all applications of subset A_0 is the most desired solution and an application placement that only maps application instances of priority class *HIGHEST* is referred to as the *worst-case solution*.

Recalling the use case introduced in Subsection 5.3.3, assume that *Computing Node 2* remains active but instead *Computing Node 3* fails. Consequently, not all application instances executed by the system before the occurrence of the failure can be assigned to one of the remaining three computing nodes since some constraints cannot be satisfied.

Assuming that the applications that are part of this use case are prioritized as defined in Table 5.11, an application placement that considers all applications that are either of priority *HIGHEST* or *HIGH* can be determined. Figure 5.7 illustrates the application placement computed by `APD` for the described use case.

Besides allowing to find an application placement for the most important applications

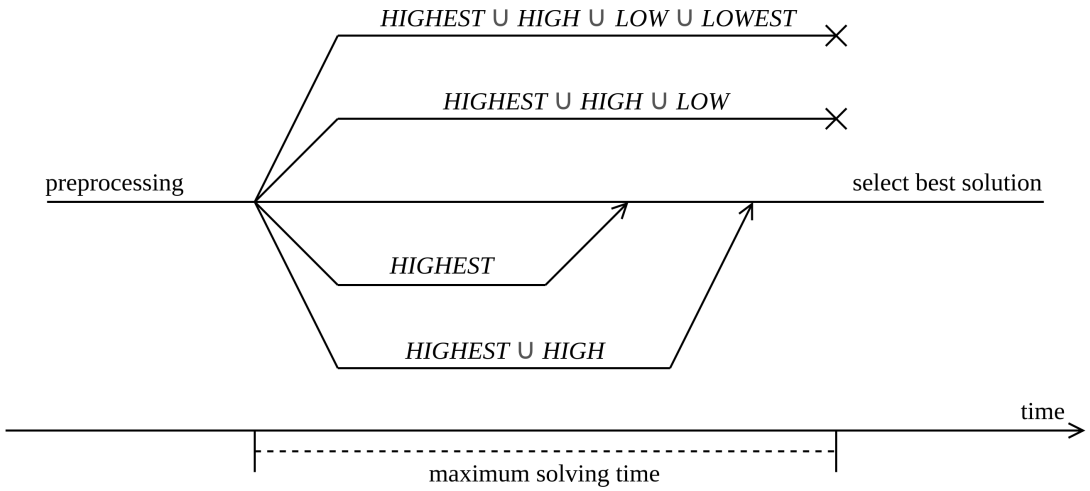

Figure 5.8: Maximum solving time causing two application-placement determiner threads to abort.

in case that 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 5.8, defining a maximum solving time causes that the application determiner threads that cannot find a valid solution within the specified time are aborted.

Defining a maximum solving time is desirable since some safety-critical situations require an application placement within a guarded time rather than a placement that considers all applications.

In case where for none of the four subsets a solution can be found, an emergency system brings the vehicle to a safe stop.

5.4 Application-Placement Determiner Testing Tool

The application-placement determiner testing tool, called `APD-TT`, is a web tool that allows a developer of an application-placement determiner to test its placement behavior in different situations. The idea thereby is that a user defines test cases which specify the current system state as well as the event that triggers the computation of a new application-placement plan. A developer can then analyze the computed placement plan.

As illustrated in Figure 5.9, `APD-TT` mocks the behavior of the switchover component, the current-state determiner, and the reconfiguration-plan advertiser. The actions performed by these mock-components can be controlled and displayed using the AngularJS [22] frontend of `APD-TT`.

Besides the web frontend, `APD-TT` also provides a corresponding backend, whereby

the backend is developed in Java and uses the Spring framework [51]. It is responsible for providing the data required by the frontend via REST interfaces. Furthermore, the backend communicates with the application-placement determiners via gRPC [23]. We used gRPC since it allows to develop application-placement determiners in several programming languages, including, for example, Java, Python, Go, and C++.

In what follows, we illustrate the workflow of APD-TT using APD and the example introduced in Subsection 5.3.3.

5.4.1 Workflow of APD-TT

The user interface of APD-TT is separated into the following three tabs:

- the *Test Cases* tab,
- the *App-Placement Determiners* tab, and
- the *Test Runs* tab.

First, the user has to switch to the *App-Placement Determiners* tab and add a new application-placement determiner. Therefore, the name, the host, as well as the port of the application-placement determiner, has to be specified. Figure 5.10 shows the configuration of APD.

Next, in the *Test Cases* tab, a test case has to be defined as illustrated in Figure 5.11, whereby we differentiate between the following two types of test cases:

- redundancy recovery and
- optimization test cases.

For both types of test cases, the current system state, i.e., the current mapping between application instances and computing nodes, has to be specified. To define the current system state, we use JSON since this format can be easily converted into gRPC messages.

The current state of the use case introduced in Subsection 5.3.3 is illustrated in the following listing:

```
{
  "applications": [
    {
      "id": "App 1",
      "minLevelOfHardwareSegregation": 2,
      "minMemory": 500,
      "minCPU": 300,
      "features": [ "x" ],
      "priority": "HIGHEST"
    },
    {
      "id": "App 2",
      "minLevelOfHardwareSegregation": 2,
```

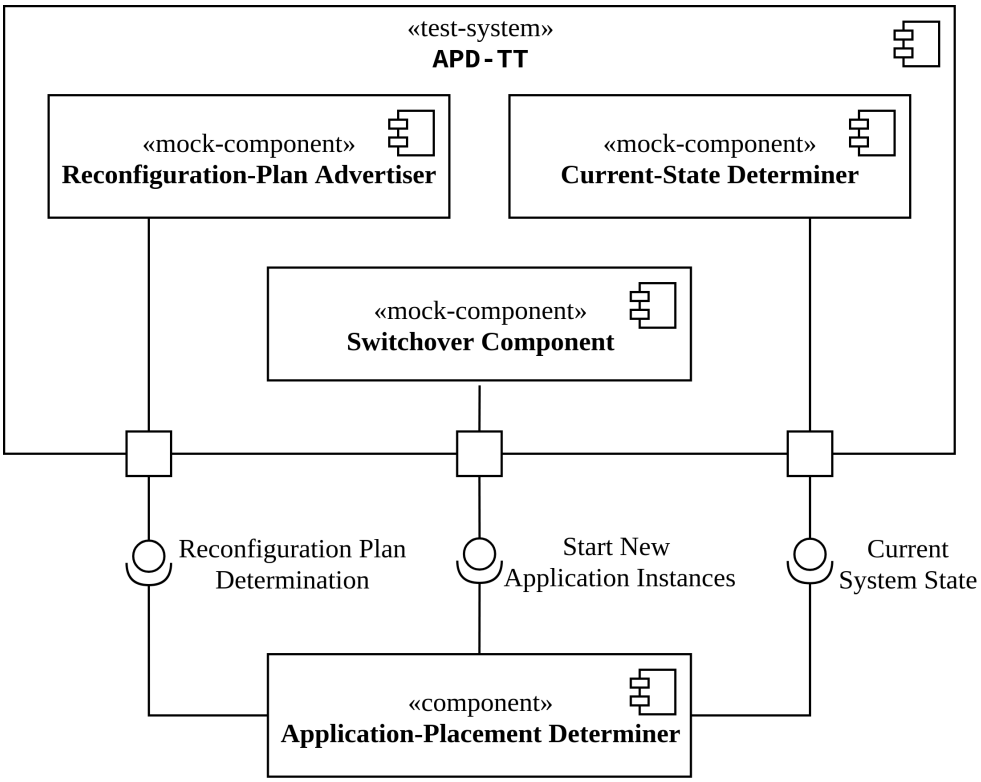

Figure 5.9: Component diagram, showing the structural design of APD-TT.

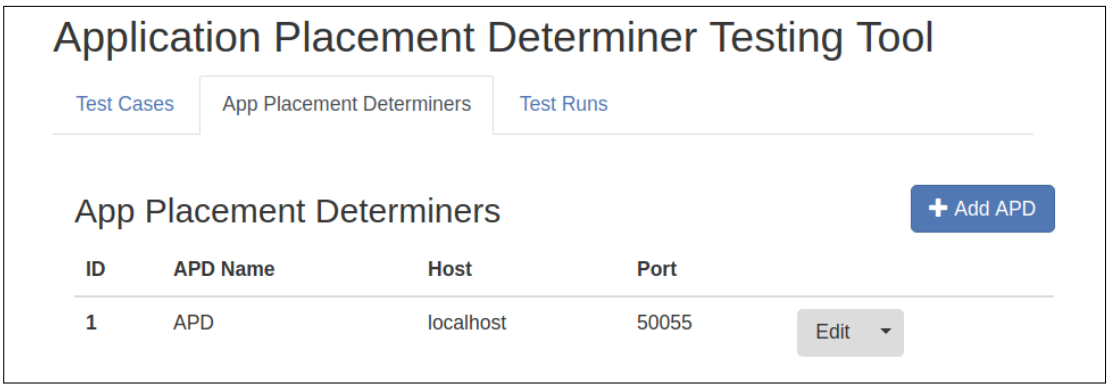

Figure 5.10: The *application-placement determiners* tab of APD-TT.

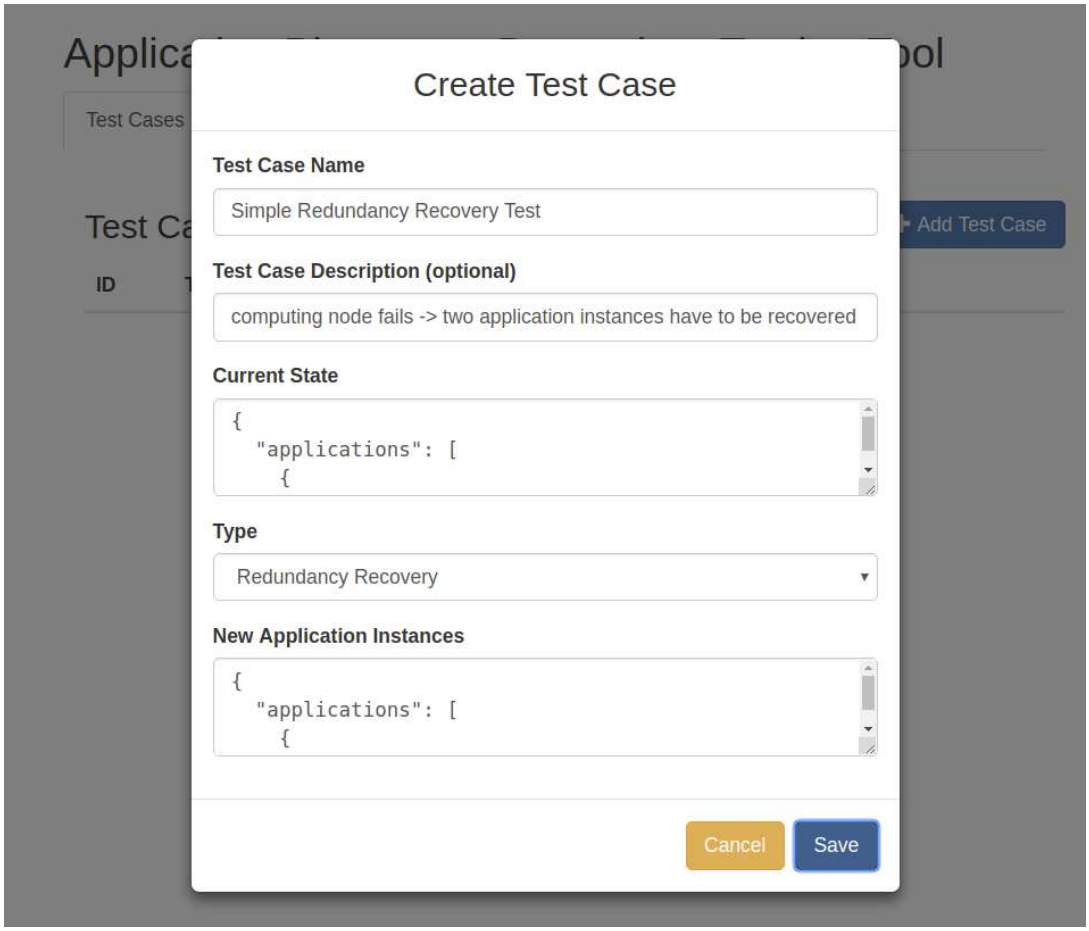

Figure 5.11: Creating a new redundancy recovery test case.

```
  "minMemory": 150,
  "minCPU": 200,
  "features": [ "y" ],
  "priority": "HIGHEST"
},
{
  "id": "App 3",
  "minLevelOfHardwareSegregation": 2,
  "minMemory": 150,
  "minCPU": 300,
  "features": [ "x", "z" ],
  "priority": "HIGHEST"
},
{
  "id": "App 4",
  "minLevelOfHardwareSegregation": 1,
  "minMemory": 100,
  "minCPU": 200,
```

```
      "features": [ "y", "z" ],
      "priority": "HIGHEST"
    }
  ],
  "computingNodes": [
    {
      "id": "CN 1",
      "totalMemory": 700,
      "totalCpu": 600,
      "features": [ "x", "z" ],
      "applicationInstances": [
        {
          "id": "1",
          "applicationId": "App 1",
          "operationMode": "ACTIVE"
        },
        {
          "id": "1",
          "applicationId": "App 3",
          "operationMode": "ACTIVE"
        }
      ]
    },
    {
      "id": "CN 3",
      "totalMemory": 1200,
      "totalCpu": 1000,
      "features": [ "x", "y", "z" ],
      "applicationInstances": [
        {
          "id": "2",
          "applicationId": "App 2",
          "operationMode": "ACTIVE"
        },
        {
          "id": "2",
          "applicationId": "App 1",
          "operationMode": "ACTIVE_HOT"
        },
        {
          "id": "1",
          "applicationId": "App 3",
          "operationMode": "ACTIVE_HOT"
        }
      ]
    },
    {
      "id": "CN 4",
      "totalMemory": 400,
      "totalCpu": 900,
      "features": [ "x", "y", "z" ],
      "applicationInstances": [
        {
```

```
        "id": "1",
        "applicationId": "App 4",
        "operationMode": "ACTIVE"
      }
    ]
  }
 ]
}
```

In the case of an optimization test case, the user has to provide no other information than the current state. Test cases of this type are indented to test the optimization ability of an application-placement determiner concerning the current state. On the other hand, in case of a redundancy recovery test, the user has to define, besides the current state, also the application instances that shall be started in order to recover the redundancy requirements. As before, we use JSON [13] to define those application instances.

For the considered use case, it is required to start two new application instances. The following listing illustrates the definition of those application instances:

```
{
  "applications": [
    {
      "id": "App 1",
      "minLevelOfHardwareSegregation": 2,
      "minMemory": 500,
      "minCPU": 300,
      "features": [ "x" ],
      "priority": "HIGHEST"
    },
    {
      "id": "App 2",
      "minLevelOfHardwareSegregation": 2,
      "minMemory": 150,
      "minCPU": 200,
      "features": [ "y" ],
      "priority": "HIGHEST"
    }
  ],
  "applicationInstances": [
    {
      "id": "3",
      "applicationId": "App 1",
      "operationMode": "ACTIVE_HOT"
    },
    {
      "id": "2",
      "applicationId": "App 2",
      "operationMode": "ACTIVE_HOT"
    }
  ]
}
```

After creating the test case, it is listed in in the *Test Cases* tab as shown in Figure 5.12. By clicking on the *Run* button, the user can select one of the previously added application-placement determiners that shall execute the test case.

The resulting test runs are listed in the *Test Run* tab, as illustrated in Figure 5.13. By clicking on the *Details* button, the user can inspect the returned result for the test case.

The following listing displays the result returned by APD for the previously defined test case:

```
{
  "computingNodes": [
    {
      "id": "CN 1",
      "applicationInstances": [
        {
          "id": "1",
          "applicationId": "App 1",
          "operationMode": "ACTIVE"
        },
        {
          "id": "1",
          "applicationId": "App 3",
          "operationMode": "ACTIVE"
        }
      ]
    },
    {
      "id": "CN 3",
      "applicationInstances": [
        {
          "id": "2",
          "applicationId": "App 2",
          "operationMode": "ACTIVE"
        },
        {
          "id": "3",
          "applicationId": "App 1",
          "operationMode": "ACTIVE_HOT"
        },
        {
          "id": "4",
          "applicationId": "App 1",
          "operationMode": "ACTIVE_HOT"
        }
      ]
    },
    {
      "id": "CN 4",
      "applicationInstances": [
        {
          "id": "2",
          "applicationId": "App 3",
          "operationMode": "ACTIVE_HOT"
```

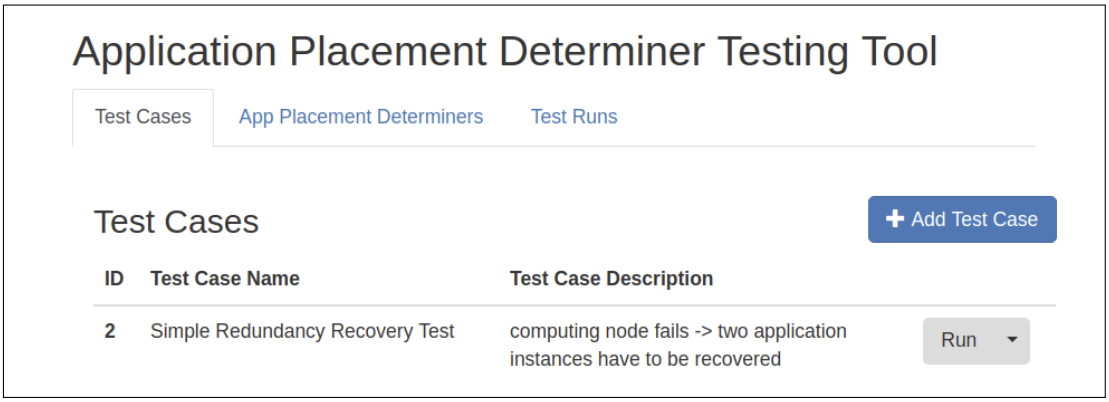

Figure 5.12: The *Test Cases* tab of `APD-TT`.

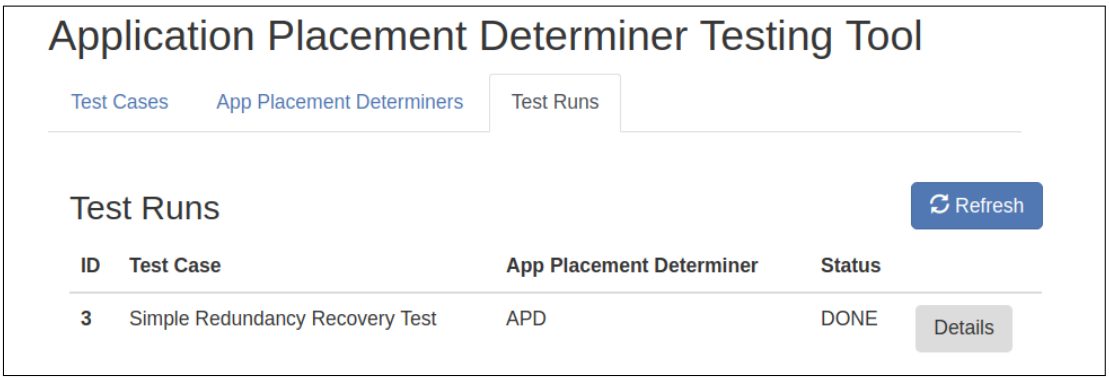

Figure 5.13: The *Test Runs* tab of `APD-TT`.

```
      },
      {
        "id": "1",
        "applicationId": "App 4",
        "operationMode": "ACTIVE"
      },
      {
        "id": "3",
        "applicationId": "App 2",
        "operationMode": "ACTIVE_HOT"
      }
    ]
  }
 ]
}
```

Note that this placement corresponds to the application placement as illustrated in Figure 5.4.

CHAPTER 6

Application-Placement Optimization

The application-placement optimization procedure is the final step of the FDIRO process. After the redundancy-recovery component recovered the redundancy requirements, it is the responsibility of the placement optimizer to optimize the application placement so that the desired goals are fulfilled.

Therefore, the placement optimizer has to solve the application-placement problem introduced in Section 5.1. Note that compared to the redundancy-recovery component, which has to solve the application-placement problem for a set of new application instances, the placement optimizer has to consider all application instances currently executed by the system.

Besides this difference, also the optimization goals implemented by the redundancy-recovery component and the placement optimizer differ. The optimization goal implemented by the redundancy-recovery component is designed to support fast solving times while, due to relaxed solving-time constraints, the placement optimizer can perform more sophisticated optimizations.

As the optimization performed by the placement optimizer shall take the current driving situation into account, we introduce, in the following, ideas to update the optimization goal based on the prevailing situation. Furthermore, we introduce ideas concerning precomputing and sharing optimized application placements.

6.1 Context-Based Application-Placement Optimization

The goal function used for optimizing the application placement strongly depends on the current driving situation as well as on the health state of the system, whereby by

system we refer to the totality of all software applications, computing nodes, and other hardware (e.g., sensors, network equipment, ...) that an autonomous vehicle is equipped with. The following examples illustrate this dependency:

- Assume that a vehicle drives a passenger to an important meeting, and the vehicle is low on battery. In such a situation, the goal of arriving on time at the desired destination is more important than entertaining the passenger. Therefore, to extend the range of the vehicle, all applications that serve the purpose of entertainment can be stopped. As a result, some computing nodes can be shut down, which saves energy.
- Suppose a vehicle is stuck in a traffic jam. Since the vehicle is barely moving, redundant instances of driving functions can be stopped. Therefore, resources are released, which can be, for example, used to execute an application that helps to improve the traffic flow [47].
- Assume that due to a crash, half of the computing nodes installed in the vehicle stopped working. Since the remaining computing nodes provide not enough resources to execute all the applications that have been executed before the crash, a new application-placement plan has to be computed. To guarantee the safety of the passengers, and the safety of other road users, the goal of the placement optimization is to map all applications that are necessary for bringing the vehicle to a safe stop.

We next introduce a method that allows determining the goal function best suited for the current situation.

6.1.1 General Concept for a Context-Based Application-Placement Optimization

To allow an application-placement optimization based on the current context, we add a layer on top of the *configuration graph*, which is a graph where the nodes correspond to configurations and directed edges indicate configuration transitions corresponding to events (e.g., failures). The layer on top of the configuration graph subdivides the configuration graph into multiple levels, whereby a level number, x, for which $0 \leq x \leq N$ holds, identifies each level, whereby N is the number of levels. The levels are defined in such a way that the safety and availability of the system are increasing as the level number is increasing. Therefore, 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 requirements cannot be satisfied anymore, a fail-safe system has to take over the control of the vehicle and bring it to a safe stop.

For each level, we define properties (e.g., minimal redundancy requirements) that an application placement of that level has to fulfill. Furthermore, the levels have to be based on each other, i.e., an application placement of level x has also to fulfill the properties required by all levels y, for $0 \leq y < x$.

Another criterion that has to be considered while defining the levels is that edges are not allowed to intersect more than one level border (recall that edges correspond to events). This means that it has to be excluded that an occurring event (e.g., the failure of a computing node) causes a degradation of the level number by two or more. Hence, an event can only cause a drop to the level below the current level, i.e., multilevel jumps are not allowed.

Furthermore, for each level, we define a goal function. As already mentioned before, level N is the most desired level. Therefore, the goal of all the other levels is to get as fast as possible to level N. This can be achieved by a goal function that prioritizes application placements that fulfill as many properties requested by the next level as possible. Once level N is reached, the application placement can be optimized based on the current driving situation.

Figure 6.1 illustrates the idea of the context-based application-placement optimization.

6.1.2 Context-Based Application-Placement Optimization based on Priorities

In the previous section, we laid down the criteria that the defined levels have to fulfill to match our approach of a context-based application-placement optimization. In this section, we give an example and prove that the specified levels meet the required criteria.

Similar as in Subsection 5.3.5, we use the following priority levels *HIGHEST*, *HIGH*, *LOW*, and *LOWEST*. We assume that each application is of exactly one priority category.

For this example, we use five levels, which are specified as follows:

- *Level 0*: At least one application of priority *HIGHEST* cannot be executed anymore.
- *Level 1*: All applications of priority *HIGHEST* can be executed. The system cannot run at least one application of priority *HIGH*.
- *Level 2*: For each application of priority *HIGHEST*, there is at least one redundant instance, whereby each redundant instance of an application is executed by a different computing node (the level of hardware segregation is two). For each application of priority *HIGH*, at least one instance is executed by the system. The system cannot run at least one application of priority *LOW*.
- *Level 3*: For each application of priority *HIGHEST*, there are at least two redundant instances, and the level of hardware segregation is three. For each application of priority *HIGH*, there is at least one redundant instance, and the level of hardware segregation is two. For each application of priority *LOW*, at least one instance is executed by the system. The system cannot run at least one application of priority *LOWEST*.
- *Level 4*: For each application of priority *HIGHEST*, there are at least three redundant instances, and the level of hardware segregation is four. For each application of priority *HIGH*, there are at least two redundant instances, and the

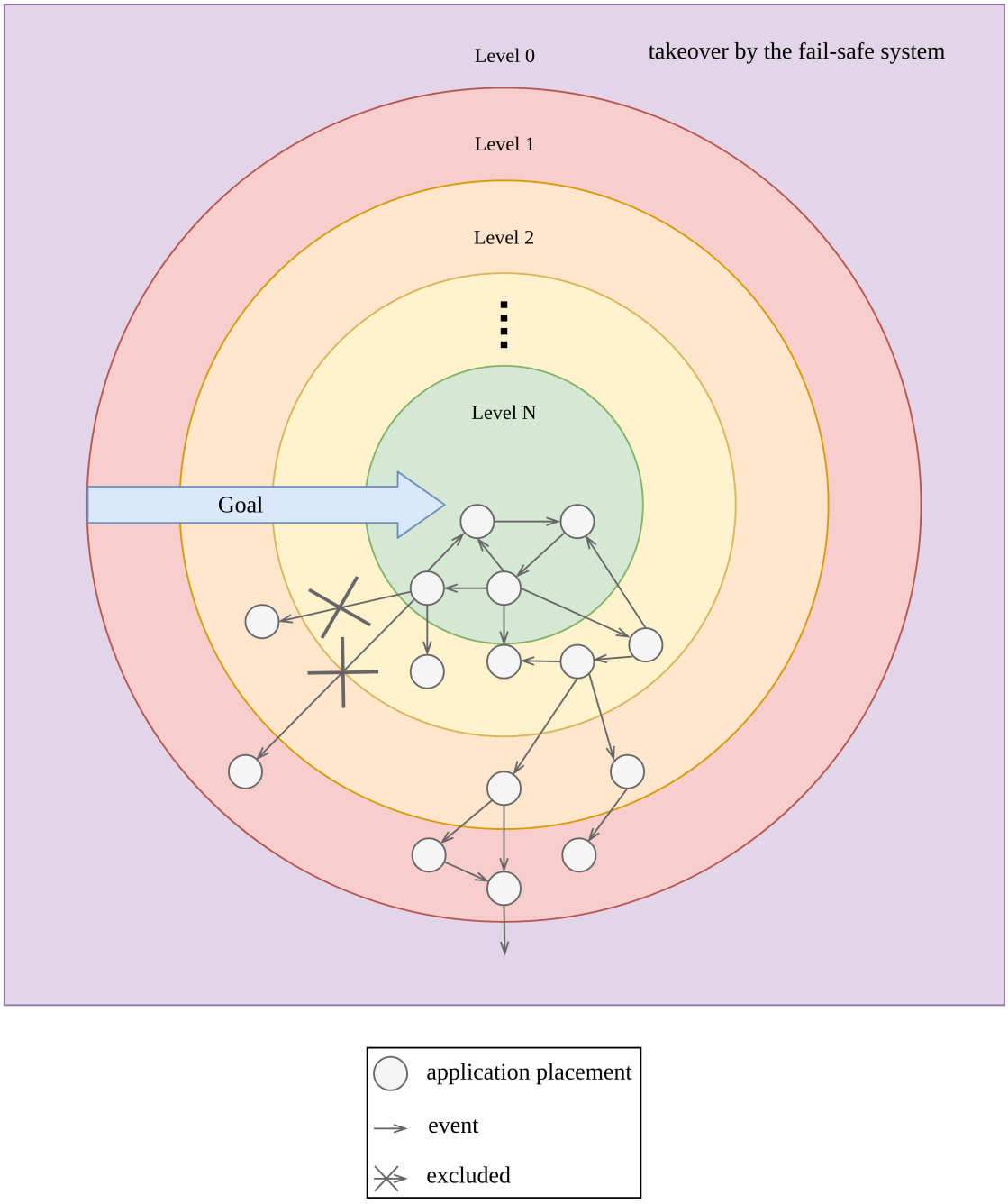

Figure 6.1: Visualization of an approach for a context-based placement optimization.

level of hardware segregation is three. For each application of priority *LOW*, there is at least one redundant instance, and the level of hardware segregation is two. For each application of priority *LOWEST*, at least one instance is executed by the system.

Figure 6.2 illustrates the arrangement of the five levels as well as their definition.

In the previous section, we defined two properties that have to hold for all levels:

- an application placement of level x has also to fulfill the properties required by all levels y, for $1 \leq y < x$, and
- multilevel jumps are excluded.

The first property is trivial to prove since the minimum redundancy and segregation requirements are increasing as the level number rises.

To prove that multilevel jumps are excluded, we have to show for all possible events which can occur that the level gets degraded at most to the level below the current level. In total, three types of events can occur:

- a failure of an application instance,
- a failure of the runtime environment, and
- a failure of a computing node.

Assume that the system is currently in level N, for $N > 0$. Therefore, the minimum number of instances and the minimal level of hardware segregation per application per priority category are given as follows:

- *HIGHEST*: N,
- *HIGH*: $\max(N-1, 0)$,
- *LOW*: $\max(N-2, 0)$, and
- *LOWEST*: $\max(N-3, 0)$.

Note that the function $\max(x, y)$ takes two numbers x and y as input and returns x if $x > y$ holds and y otherwise.

The minimum number of instances and the minimal level of hardware segregation per application per priority category of level $N-1$ are given as follows:

- *HIGHEST*: $N-1$,

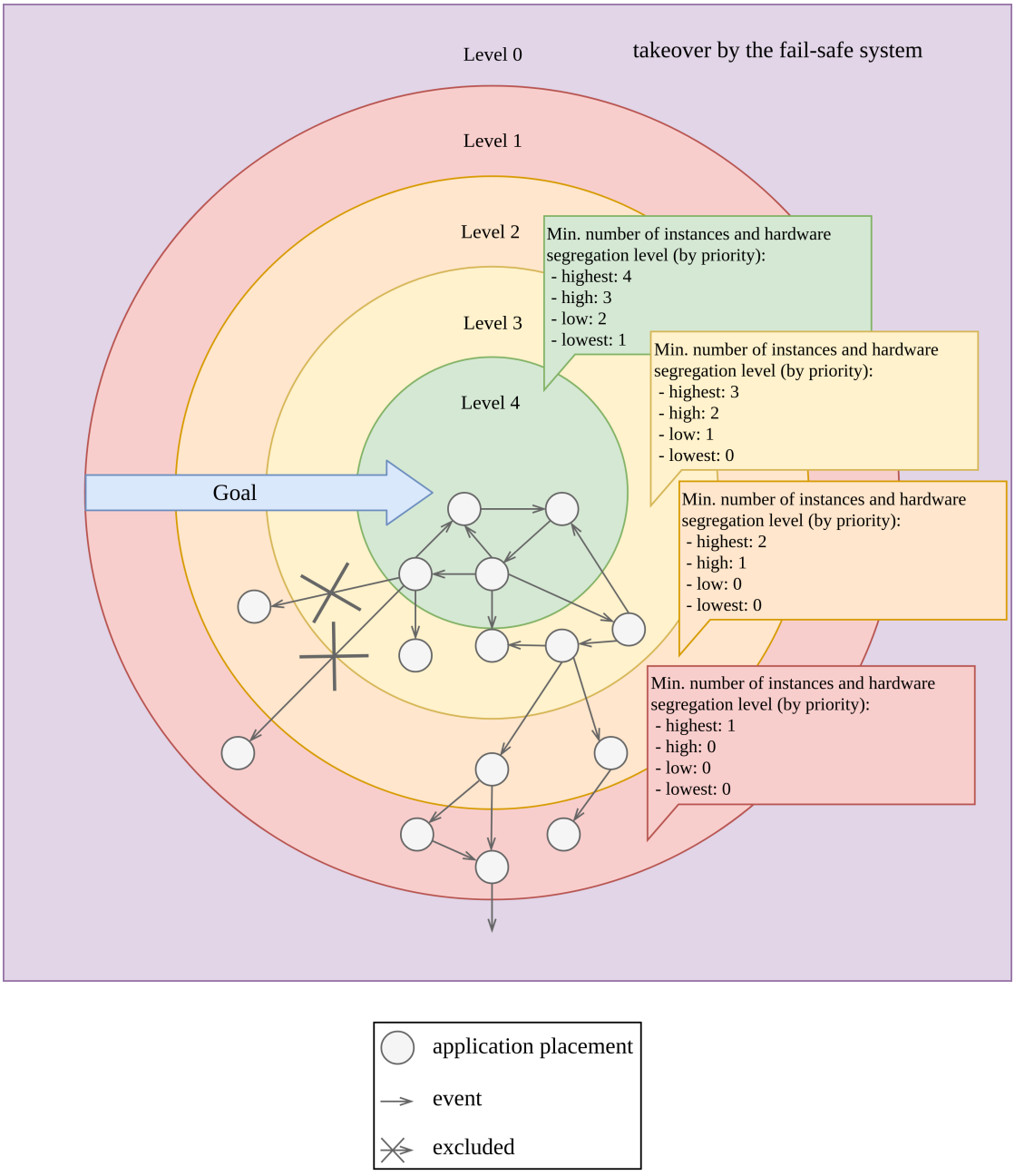

Figure 6.2: Visualization of an approach for a context-based placement optimization which is based on priorities.

- *HIGH*: $\max(N-2,0)$,
- *LOW*: $\max(N-3,0)$, and
- *LOWEST*: $\max(N-4,0)$.

In the case that an application instance fails, the number of instances of the corresponding application is reduced by one. If the application exactly matches the minimal required number of instances of level N, then the system will drop to level $N-1$. Otherwise (in case the application exceeds the minimal required number of instances), the system remains in level N.

A failure of the runtime environment causes that all the application instances executed by the runtime environment will also fail. Since, for each application, the minimum level of hardware segregation is equal to the minimum number of application instances, we can assume that for applications that match exactly the minimal requirements of level N, at most one instance is affected by a runtime environment failure. Therefore, this type of failure can be reduced to the failure of an application instance. As we already argued before, our approach avoids multilevel jumps in case an application instance fails. Therefore, multilevel jumps are also excluded in case of a runtime environment failure.

A failure of a computing node causes that all application instances executed by this computing node will also fail. As argued before, for each application, the minimal level of hardware segregation is equal to the minimum number of application instances. Therefore, we can assume that for applications that match exactly the minimal requirements of level N, at most one instance is affected by the failure of the computing node. Therefore, this type of failure can be reduced to the failure of an application instance. Thus, multilevel jumps are also excluded in case of a computing node failure.

We have shown that, for all three types of events that the current level the system is in, gets reduced by at most one. Therefore, multilevel jumps are excluded.

Since the goal of levels 1, 2, and 3 is to get to level 4, the optimization goals of those levels are defined as specified in Table 6.1.

Example Use Case

Consider a car with six computing nodes, as illustrated in Figure 6.3. In total, four applications (*App 1* to *App 4*) are executed by the system, whereby the applications belong to the following priority classes:

- *HIGHEST*: *App 1*,
- *HIGH*: *App 2*,
- *LOW*: *App 3*, and

Table 6.1: Goals of the context-based application-placement optimization.

Level	Optimization Goal
1	Try to run one redundant instance of each application of priority *HIGHEST*, whereby the level of hardware segregation is two. Run as many applications of priority *HIGH* as possible.
2	Try to run one additional redundant instance of each application of priority *HIGHEST*, whereby the level of hardware segregation is three. Try to run one redundant instance of each application of priority *HIGH*, whereby the level of hardware segregation is two. Run as many applications of priority *LOW* as possible.
3	Try to run one additional redundant instance of each application of priority *HIGHEST*, whereby the level of hardware segregation is four. Try to run one additional redundant instance of each application of priority *HIGH*, whereby the level of hardware segregation is three. Try to run one redundant instance of each application of priority *LOW*, whereby the level of hardware segregation is two. Run as many applications of priority *LOWEST* as possible.
4	Driving situation based optimization.

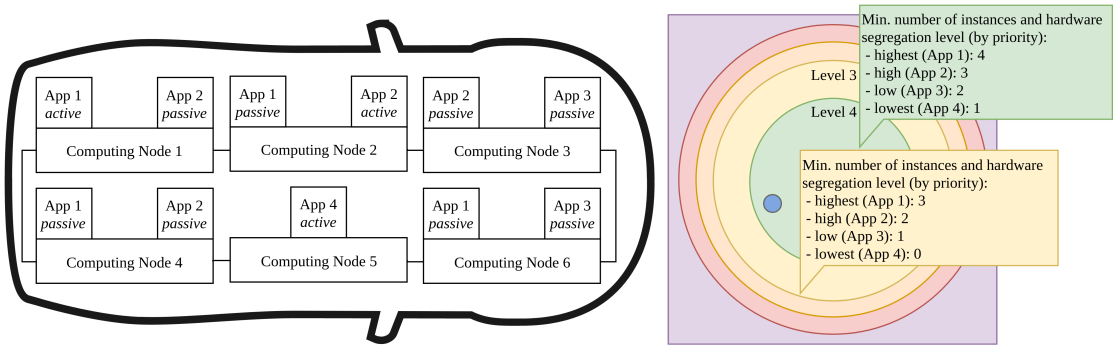

Figure 6.3: Initial configuration that satisfies the requirements of level 4.

- *LOWEST*: *App 4*.

In its initial configuration, the system is in optimal condition since it fulfills all the requirements requested by level 4.

Next, we assume, as illustrated in Figure 6.4, that the active instance of *App 1* fails. Through a switchover to one of the passive instances of *App 1*, a total loss of *App 1* can be avoided. Note that level 4 requires that for each application of priority *HIGHEST*, it holds that the minimal number of instances, as well as the minimal level of hardware

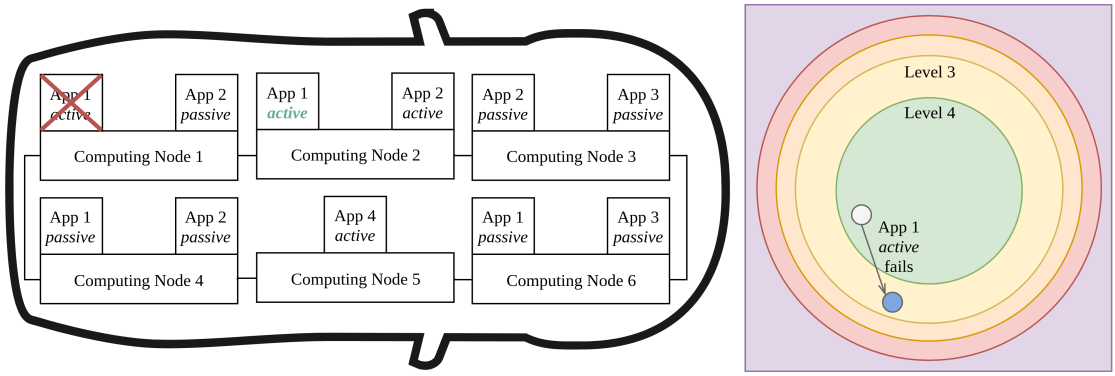

Figure 6.4: Configuration after the active instance of *App 1* fails.

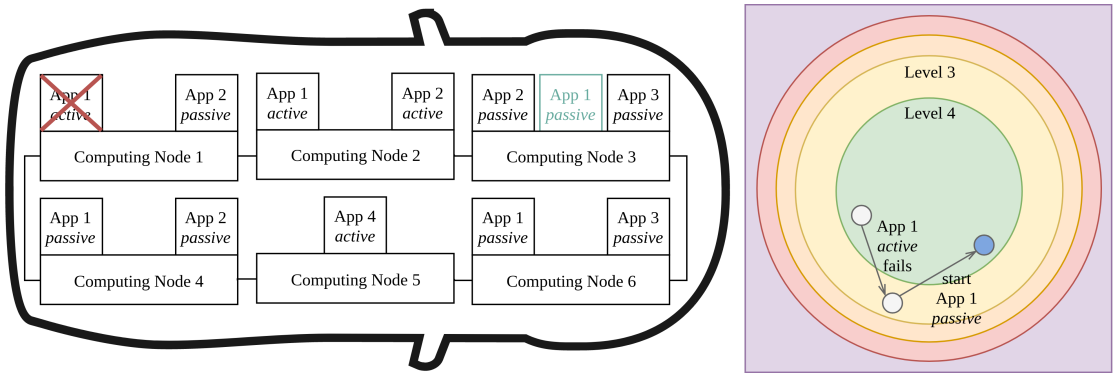

Figure 6.5: Configuration that recovers the system to meet the requirements of level 4 again.

segregation, is four. Due to the failure of the active instance, *App 1* misses one application instance to satisfy the requirements of level 4. Therefore, the successor configuration is not an element of level 4 but rather belongs to level 3.

As mentioned before, the goal of all the non-optimal levels, i.e., all levels apart from level 4, is to perform recovery operations so that the system gets to the highest possible level. As illustrated in Figure 6.5, level 4 can be recovered by starting a new passive instance of *App 1*.

Once the system is again in level 4, optimizations based on the current driving situations can be performed. Assuming that the goal is to extend the range of the vehicle, a switch to an application placement, which utilizes only a subset of all available computing nodes is preferred. Therefore, as illustrated in Figure 6.6, the active instance of *App 4* is moved to another computing node so that *Computing Node 5* can be shut down. Apart from that optimization, further energy-saving measures can be performed as long as the system is in level 4.

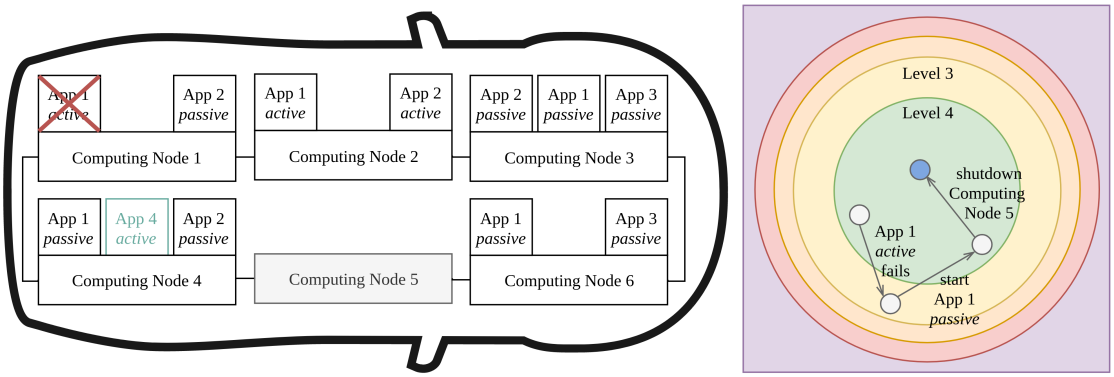

Figure 6.6: Configuration that extends the range of the vehicle.

6.2 Dynamic Context-Based Application-Placement Optimization

In the approach described before, we defined the goals of the optimization problem statically. As a result, the system has to reach level N before an optimization based on the current driving situation is allowed. However, some driving situations, like, for example, cruising at low speed in a traffic jam, do not require a highly redundant software architecture.

To address this issue, we introduce in what follows two approaches which allow a dynamic context-based application-placement optimization.

6.2.1 Dynamic-Level Approach

A very general approach to make our model of a context-based application-placement optimization more dynamic is to adjust the number N of levels based on the current context.

Recall that in Subsection 6.1.1, we defined that an optimization based on the driving situation is only allowed at level N. On the other hand, the optimization goal of all the other levels is to fulfill the requirements requested by the next highest level. Therefore, adjusting the number of levels causes that the level in which a driving situation-based optimization is allowed is varying.

Figure 6.7 illustrates the idea of a dynamic adjustment of the number of levels. The figure shows that based on the current context, like, for example, the speed of the vehicle, levels are added or removed.

Since many different contexts are considerable, conflicts regarding the adjustment of the number of levels can emerge. For example, assume a car driving at a very low speed on a road where the sidewalks are filled with pedestrians. Assume further that, based on the present low speed, the required number of levels is two. However, since the car is

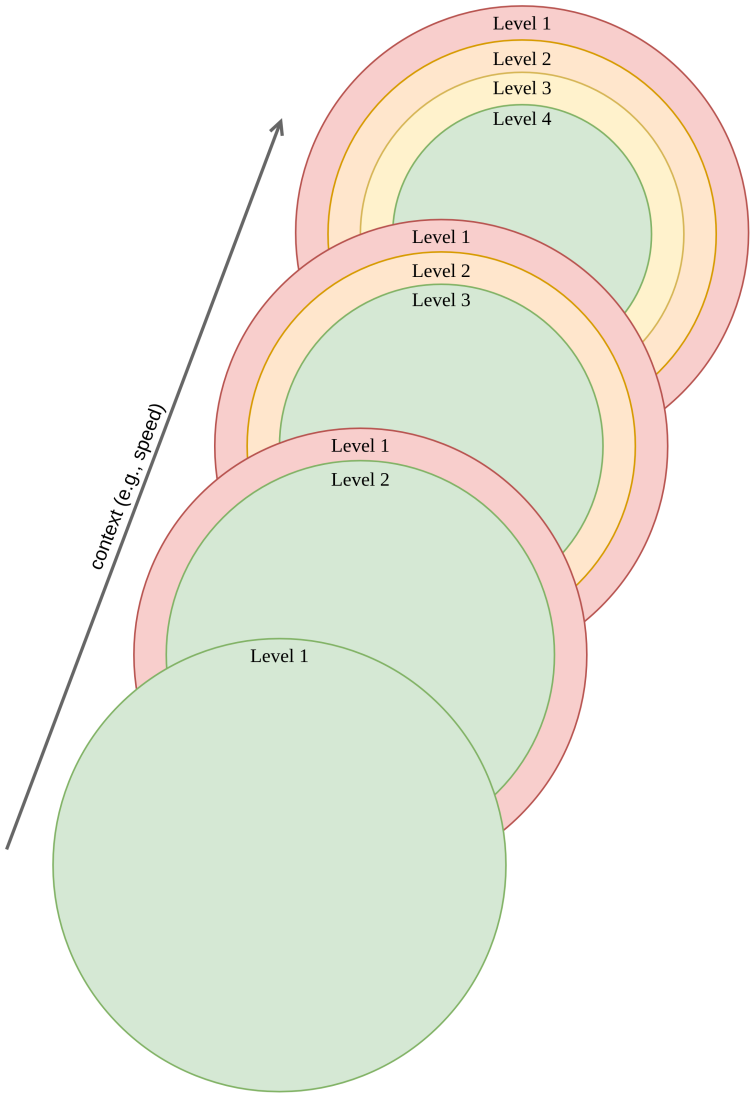

Figure 6.7: Dynamic adjustment of the number of levels based on the context.

surrounded by many pedestrians, the minimal requirement is $N = 3$. This means that the latter context demands a higher number of levels than the former. In such a case, always the higher requirements are applied. This ensures that the minimal demands of each context are satisfied.

Example Use Case

To illustrate the dynamic level approach, we provide in what follows an example in which the number of levels is adjusted based on the speed of the vehicle. To keep this example simple, we do not consider any other context besides speed.

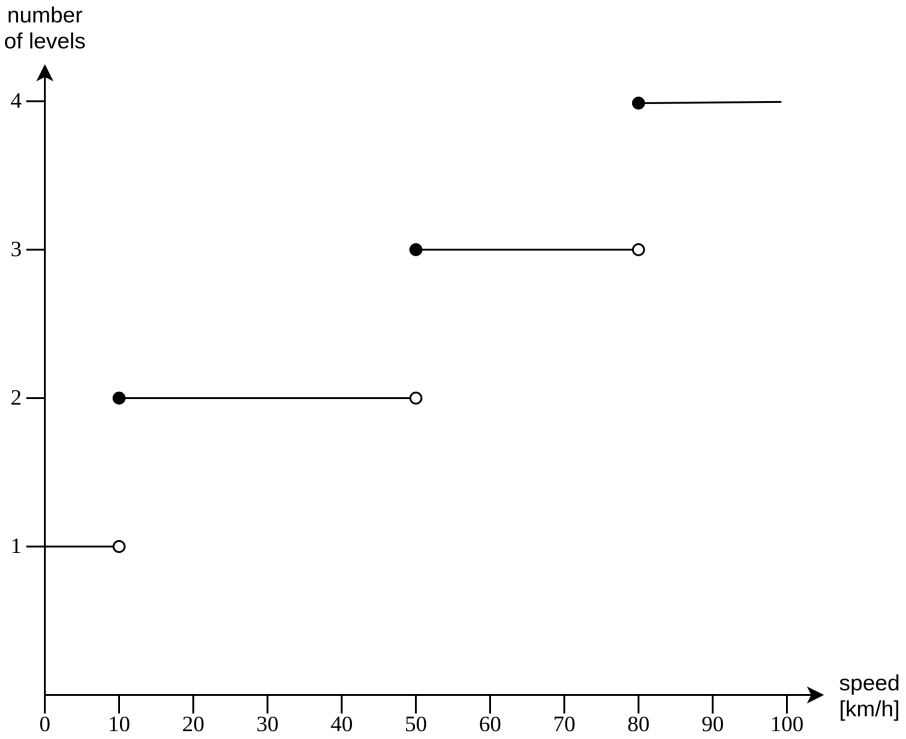

Figure 6.8: Function that specifies the number of levels required at different speeds.

The function displayed in Figure 6.8 specifies the correspondence between speed and the number of required levels.

Assuming a car is parked, i.e., the speed in 0 km/h, according to Figure 6.8, the number N of levels equals 1. Furthermore, we define that in total five applications (*App 1* to *App 5*) are executed by the system, whereby the applications belong to the following priority classes:

- *HIGHEST*: *App 1*, *App 2*,
- *HIGH*: *App 3*,
- *LOW*: *App 4*, and
- *LOWEST*: *App 5*.

The configuration depicted in Figure 6.9 shows a valid application placement that satisfies all the properties required by level 1.

Assuming that the car accelerates to 30 km/h, the number N of levels increases to 2. As a result, the optimization goal of level 1 is to bring the system to level 2. This can be

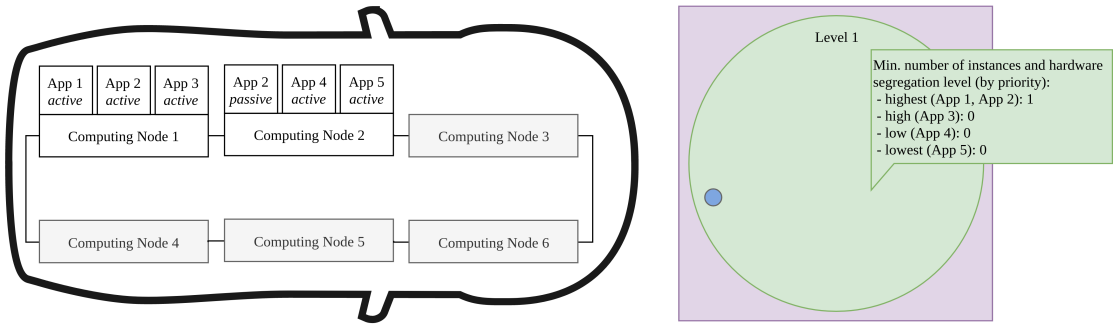

Figure 6.9: Initial configuration that satisfies all requirements of level 1 (speed is 0 km/h, and thus the number N of required levels is 1).

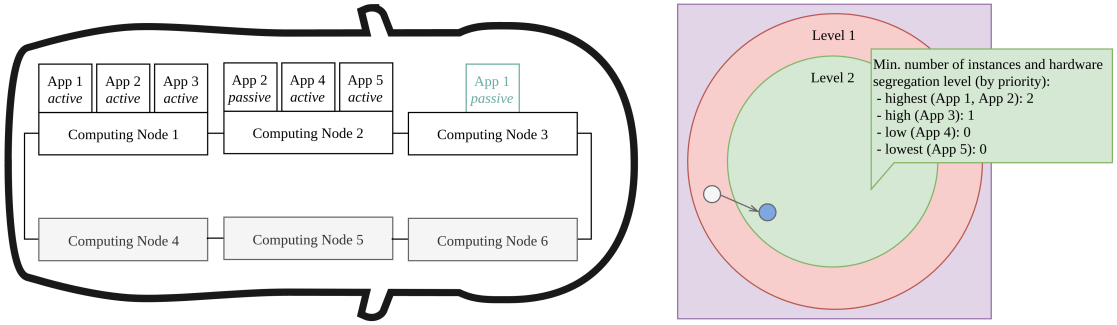

Figure 6.10: Configuration after starting a passive instance of *App 1* (speed is 30 km/h, and thus the number N of required levels is 2).

achieved by starting a passive instance of *App 1*. The resulting application placement is displayed in Figure 6.10.

Next, the vehicle increases its speed to 70 km/h. Consequently, the number N of levels gets updated again to 3. The newly introduced level requires that at least three instances of all applications of priority *HIGHEST* are executed by the system. Furthermore, level 3 requires the execution of at least two instances of all applications of priority *HIGH*. Therefore, to upgrade the system to level 3, additional passive instances of *App 1* and *App 2* are required. The resulting configuration is displayed in Figure 6.11.

6.2.2 Dynamic-Prioritization Approach

Another approach for a dynamic context-based application-placement optimization that is based on the idea proposed in Subsection 6.1.2 is to dynamically vary the priority class an application belongs to. This means that for each application and each context, the corresponding priority has to be defined. In case that two contexts define conflicting priorities, always the higher priority class gets applied. Therefore, it is guaranteed that the minimum requirements of all contexts are satisfied.

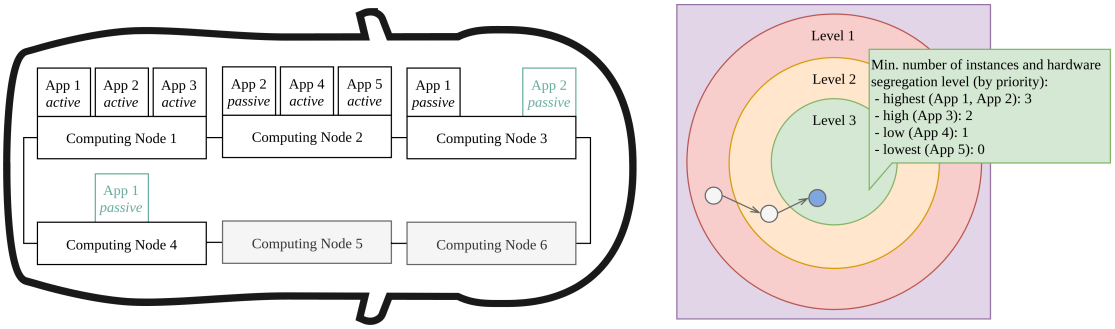

Figure 6.11: Configuration after starting a passive instance of *App 1* and *App 2* (speed is 70 km/h, and thus the number N of required levels is 3).

Table 6.2: Specification of the priority class per application at different speed classes.

speed v [km/h]	*App 1*	*App 2*	*App 3*	*App 4*	*App 5*
$v < 10$	*LOWEST*	*HIGHEST*	*HIGH*	*LOWEST*	*LOW*
$10 \leq v < 50$	*LOW*	*HIGH*	*HIGHEST*	*LOW*	*HIGH*
$50 \leq v < 80$	*HIGH*	*LOW*	*HIGH*	*LOW*	*HIGH*
$v \geq 80$	*HIGHEST*	*LOWEST*	*LOW*	*LOW*	*HIGHEST*

Example Use Case

To clarify the dynamic-prioritization approach, consider a system that runs five applications. Recall that we defined earlier that the prioritization of the applications depends on the current context. For reasons of simplification, we only consider speed as context in this example.

Table 6.2 defines the priority classes for all applications based on the different speed intervals.

Assuming that a vehicle is moving at a speed of less than 10 km/h, according to Table 6.2, *App 1* and *App 4* both belong to the priority class *LOWEST*, *App 5* is of priority *LOW*, *App 3* of priority *HIGH*, and *App 2* belongs to the priority class *HIGHEST*.

The configuration depicted in Figure 6.12 shows a valid application placement that fulfills the requirements of level 4.

Assuming that the car accelerates to a speed of 40 km/h, the priority of the applications change. According to Table 6.2, the priorities change as follows:

- the priority of *App 1* and *App 4* get upgraded from *LOWEST* to *LOW*,
- the priorities of *App 2* gets downgraded to priority *HIGH*,
- the priority of *App 3* gets upgraded to *HIGHEST*, and

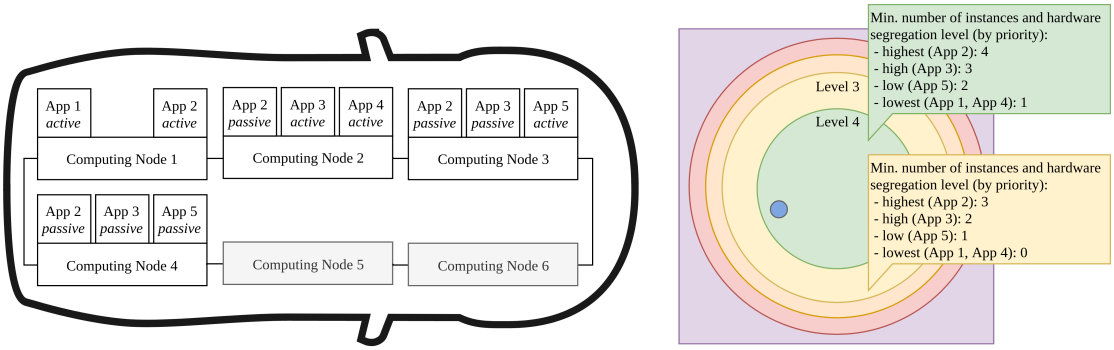

Figure 6.12: Initial configuration that fulfills all requirements of level 4 (speed less than 10 km/h, and thus the number of required levels is 1).

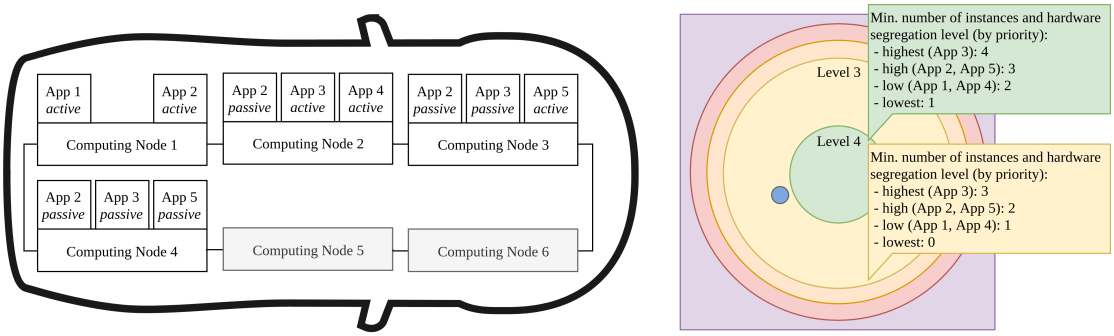

Figure 6.13: Context change (speed is 40 km/h) that cause the current configuration to drop from level 4 to level 3.

- the priority of *App 5* gets upgraded to *HIGH*.

Since *App 1*, *App 3*, *App 4*, and *App 5* do not fulfill requirements of level 4 anymore, the level of the current configuration drops to level 3, as is illustrated in Figure 6.13. Note that the dynamic prioritization approach causes that the level boundaries are flexible, i.e., a context change can cause a drop or a lift of the level of the system.

To get to level 4 again, additional instances of *App 1*, *App 3*, *App 4*, and *App 5* are required. Since the priority of *App 2* dropped from *HIGHEST* to *HIGH*, one redundant instance of *App 2* can be stopped. Figure 6.14 shows an application placement that satisfies all requirements demanded by level 4.

Note that in order to not violate the multilevel jump property defined in Subsection 6.1.1, it must be ensured that the priority of an application does not get upgraded by more than one priority level in case of a context change. This means that, for example, a lift of *App 1* from priority *LOWEST* to priority *HIGH* in case that the speed exceeds the 10 km/h threshold has to be excluded.

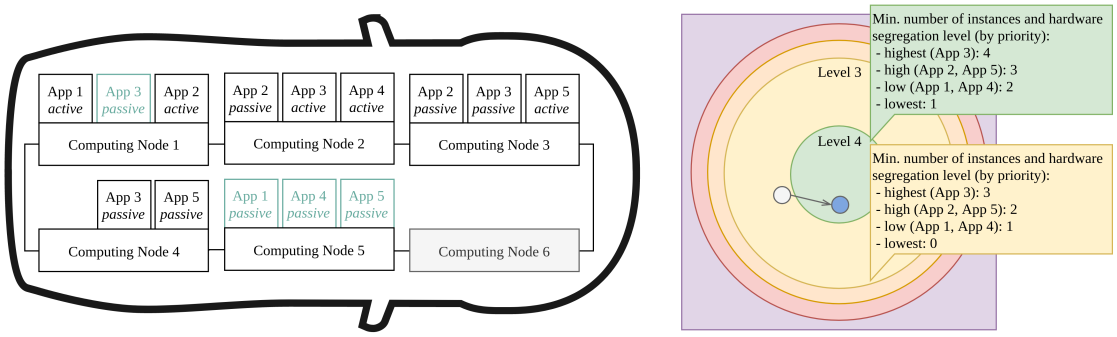

Figure 6.14: Configuration that recovers the system to meet the requirements of level 4 again.

6.2.3 Dynamic-Level Approach vs. Dynamic-Prioritization Approach

At first glance, the two approaches for a dynamic context-based application-placement optimization seem quite similar. However, on closer examination, it turns out that each approach has different characteristics though.

The dynamic-level approach, for example, is a general approach that is not bounded to a specific manner of defining levels. On the other hand, the dynamic prioritization approach is less general. As indicated by the name of this approach, priorities are a key aspect of this idea. In case that the levels are not defined using priorities, this approach might not be applicable.

However, the dynamic-prioritization approach allows a more detailed specification of how the redundancy requirements of an application vary in case of a context change. For example, with this approach, as illustrated in Subsection 6.2.2, it is expressible that the redundancy requirements of an application rise as the speed increases, while, on the other hand, the redundancy requirements of another application decrease as the speed increases. Due to this flexible priority classification, computing resources are utilized more efficiently.

Although the dynamic-prioritization approach requires that the developer of an application specifies for each context the corresponding priority level, this can be a cumbersome and challenging task, especially if many different contexts are considered. The dynamic-level approach, on the other hand, requires just to define the minimum number of levels that are demanded by the different contexts, which is considered to be an easier task than the challenge described before. Therefore, such an approach might be easier to implement.

6.3 Preventive-Reconfiguration Computation

It can be assumed that computing an optimized reconfiguration is a time-consuming task. To reduce the time of finding an appropriate reconfiguration after an event occurred, we propose to precompute reconfigurations, i.e., a reconfiguration for a certain event

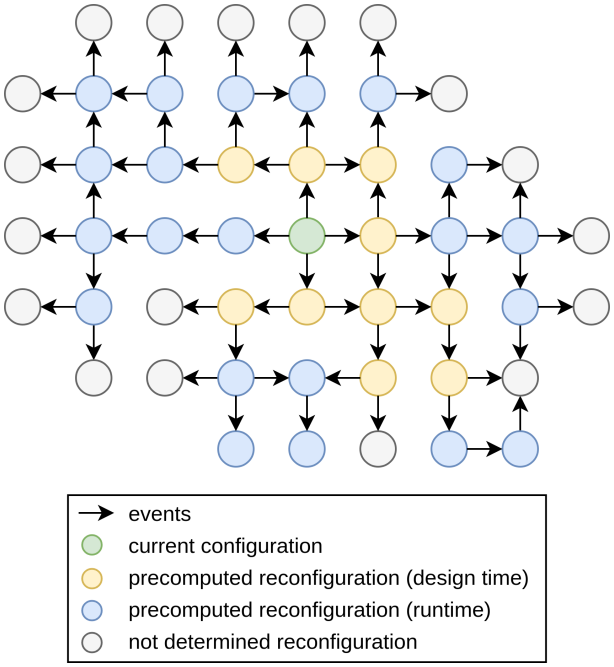

Figure 6.15: Visualization of a partially precomputed reconfiguration graph.

is determined before this event actually occurs. Besides faster reconfiguration times, precomputing reconfigurations can also lead to an application placement of better quality since more computation time can be consumed as the event has not occurred yet.

In case that the hardware architecture, as well as the applications executed by the system, are known, a precomputation of reconfigurations for common events can be done even before the vehicle goes into operation for the first time, i.e., at design time.

Figure 6.15 illustrates the idea of precomputing reconfigurations. The yellow nodes in the reconfiguration graph correspond to reconfigurations that have to be precomputed at design time. Since, as already argued in Section 3.2, it is not possible to precompute the entire reconfiguration graph, the idea is to precompute reconfigurations for events that have not occurred yet at runtime. Precomputed reconfigurations are illustrated by blue nodes in Figure 6.15. In case that no precomputed reconfiguration for an occurring event exists, a new reconfiguration has to be computed at runtime.

Since the optimization of the application placement depends on the actions performed in the isolation step of the FDIRO process, the operations done in those steps shall be precomputed as well. Thus, a reconfiguration includes the switchover commands that shall be rolled out by the switchover controller and an optimized placement plan.

In case that a precomputed reconfiguration plan exists, the redundancy-recovery component can be bypassed since the optimized configuration plan can consider the recovery of the redundancies, as illustrated in Figure 6.16.

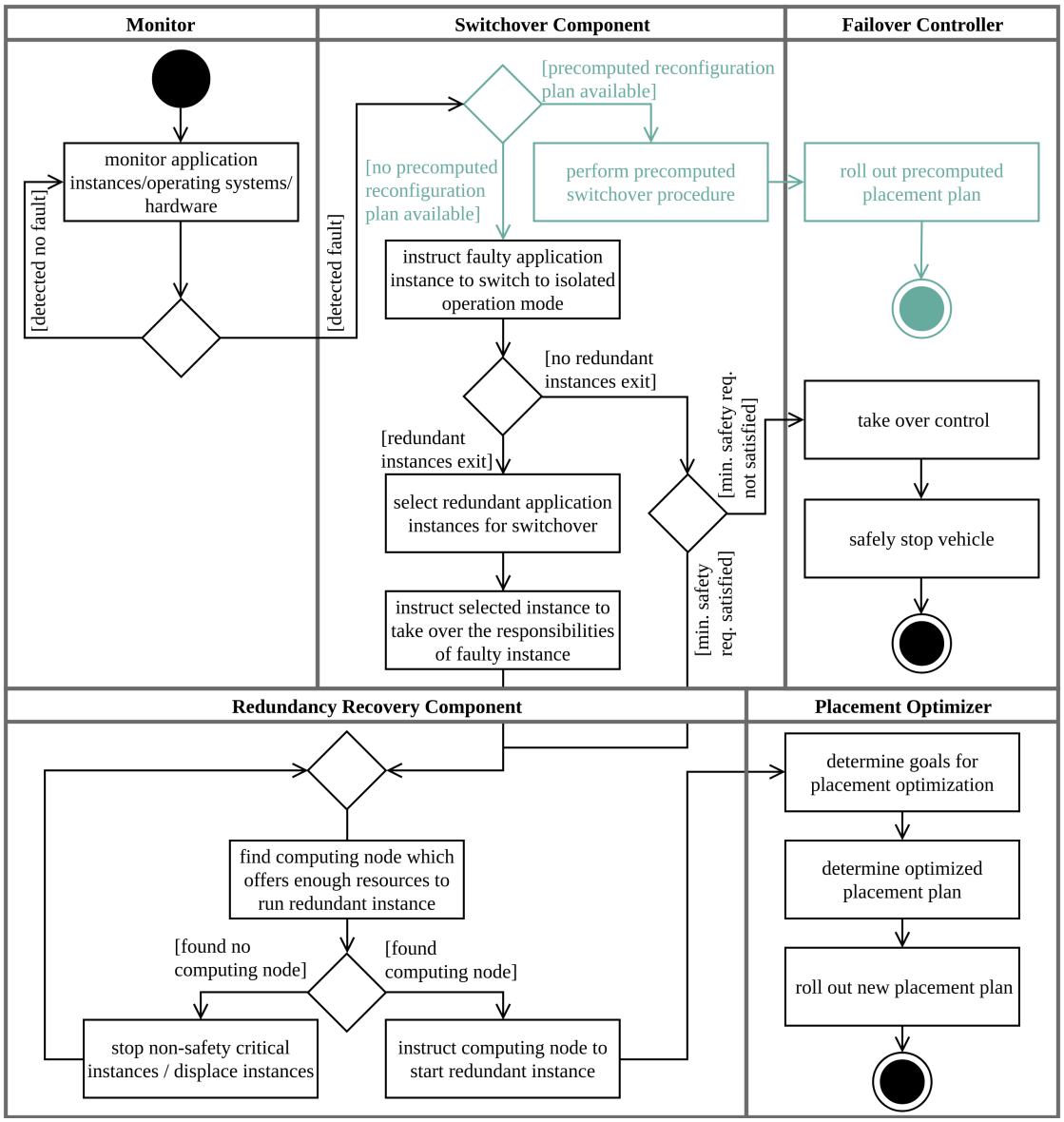

Figure 6.16: Activity diagram of the stepwise reconfiguration process extended by the preventive reconfiguration computation approach.

6.3.1 Naive Configuration Precomputation

A naive approach for precomputing reconfigurations is to precompute n hierarchy levels of the reconfiguration graph, for some $n > 0$. As illustrated in Figure 6.17, the hierarchy levels are sequentially precomputed.

The advantage of this approach is that it is easy to implement. However, this approach does not take the severity of the events as well as the probability of their occurrence into

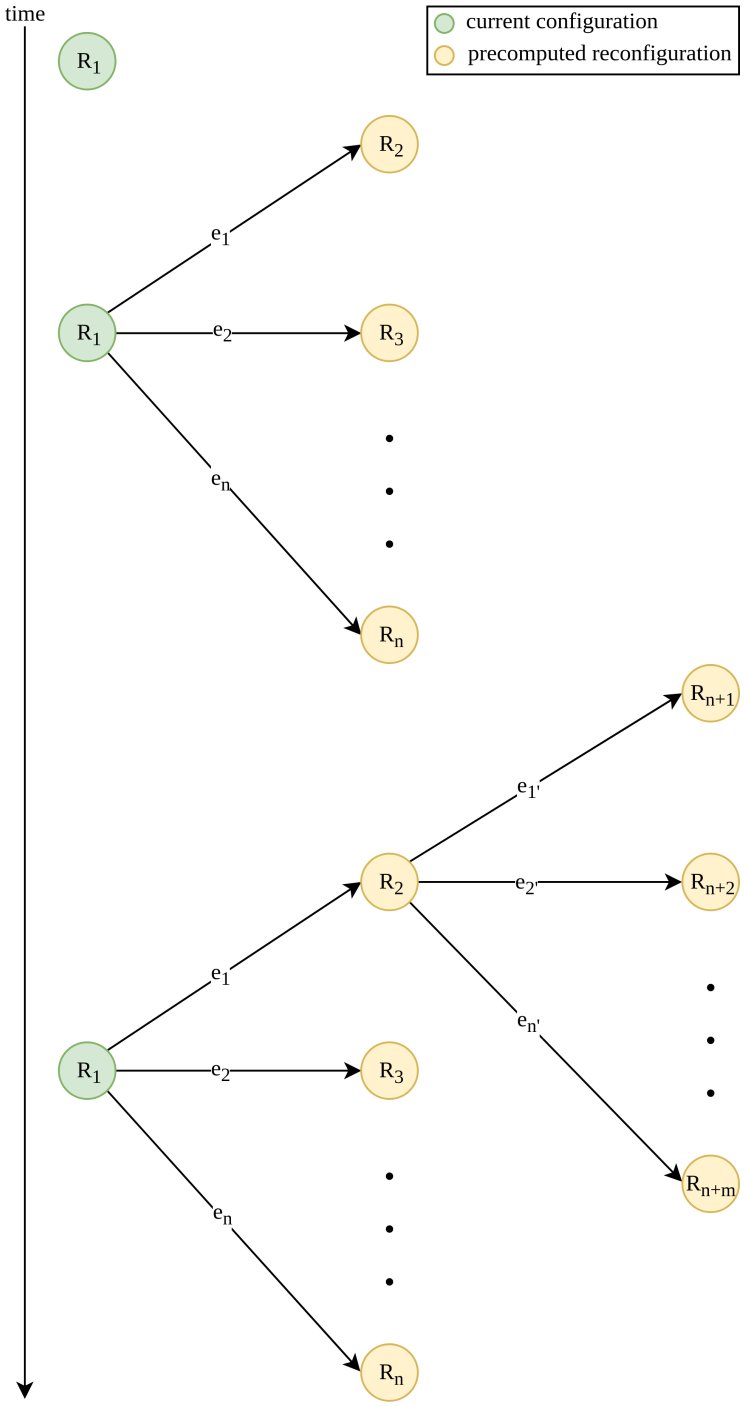

Figure 6.17: Visualization of a naive approach for precomputing reconfigurations.

account. Therefore, we introduce in what follows another approach that considers these two properties.

6.3.2 Informed Reconfiguration Precomputation

The idea of the informed reconfiguration precomputation approach is that the precomputation of reconfigurations is based on the probability of the occurrence of sequences of events as well as the severity of those sequences. In order to realize this approach, we have to determine, for each configuration, the probability, $P(e)$, of the occurrence of each event e that can affect this configuration. Furthermore, for each event e, the severity, $S(e)$, has to be determined.

Since we do not know the exact likelihood of the occurrence of an event, nor the exact severity, we assume that each event is rated according to the ASIL ("Automotive Safety Integrity Level") risk classification scheme as defined in the ISO 26262 standard [30, Part 9]. ASIL defines four severity classes (S0 to S3), which are specified as follows:

- S0: no injuries;
- S1: light and moderate injuries;
- S2: severe and life-threatening injuries (survival possible); and
- S3: life-threatening injuries (survival uncertain).

The severity of the failure of a passive instance of a multimedia application is, for example, categorized as S0. On the other hand, a failure of an active instance of the trajectory planner, whereby no redundancies exist, is represented by the severity class S3.

Furthermore, ASIL defines five levels of exposure (E0 to E4):

- E0: incredibly unlikely;
- E1: very low probability;
- E2: low probability;
- E3: medium probability; and
- E4: high probability.

As illustrated in Figure 6.18, for each event, the severity and exposure level has to be specified. Since the goal of the informed reconfiguration precomputation is first to precompute reconfigurations which originate from events for which the probability of occurrence is high and which are of critical severity, we have calculated for each reconfiguration R that has not been precomputed yet a utilization value $U(R, R_{current})$,

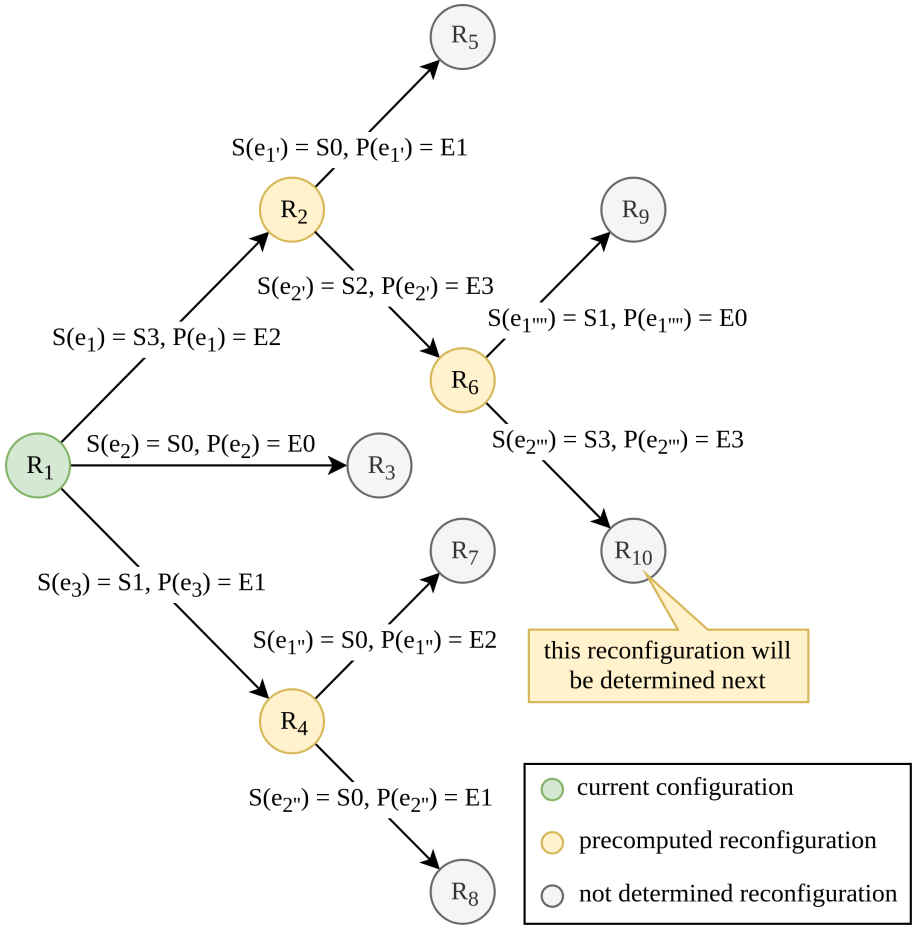

Figure 6.18: Visualization of the informed reconfiguration precomputation approach.

where this value is determined based on the severity and the probability of the occurrence of the sequence of events that originates from the current configuration, $R_{current}$. The utilization values are then used to determine the reconfiguration that shall be precomputed next.

A naive approach to determine $U(R, R_{current})$ is to define a function, F, which specifies for each severity-exposure combination a value, whereby along the path from $R_{current}$ to R, the values defined by F are multiplied. Using this approach, whereby the function F is defined as illustrated in Table 6.3, the utilization values of the reconfiguration graph, as shown in Figure 6.18, are calculated as follows:

$$
\begin{aligned}
U(R_2, R_1) &= F(S3, E2) = 0.5, \\
U(R_3, R_1) &= F(S0, E0) = 0.02, \\
U(R_4, R_1) &= F(S2, E1) = 0.18, \\
U(R_5, R_1) &= U(R_2, R_1) \cdot F(S0, E1) = 0.5 \cdot 0.06 = 0.03,
\end{aligned}
$$

Table 6.3: Function F that rates the different severity-exposure combinations.

F	S0	S1	S2	S3
E0	0.02	0.04	0.06	0.1
E1	0.06	0.12	0.18	0.3
E2	0.1	0.2	0.3	0.5
E3	0.14	0.28	0.42	0.7
E4	0.2	0.4	0.6	1

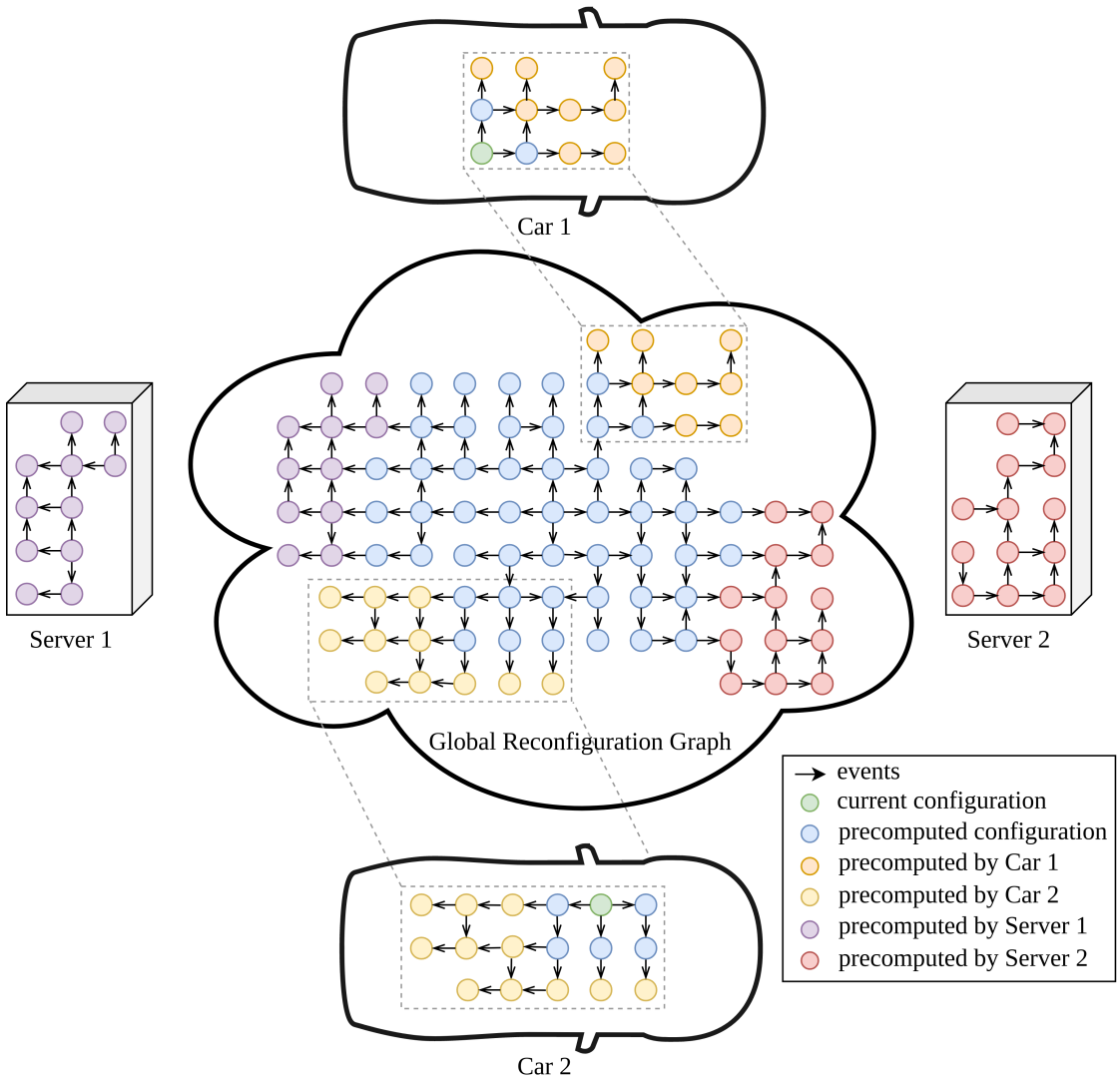

Figure 6.19: Visualization of the global reconfiguration graph.

$$
\begin{aligned}
U(R_6, R_1) &= U(R_2, R_1) \cdot F(S2, E3) = 0.5 \cdot 0.42 = 0.21, \\
U(R_7, R_1) &= U(R_4, R_1) \cdot F(S0, E2) = 0.18 \cdot 0.3 = 0.054, \\
U(R_8, R_1) &= U(R_4, R_1) \cdot F(S0, E1) = 0.18 \cdot 0.06 = 0.0108, \\
U(R_9, R_1) &= U(R_6, R_1) \cdot F(S1, E0) = 0.21 \cdot 0.04 = 0.0084, \text{ and} \\
U(R_{10}, R_1) &= U(R_6, R_1) \cdot F(S3, E3) = 0.21 \cdot 0.7 = 0.147.
\end{aligned}
$$

According to Figure 6.18, R_2, R_4, and R_6 are already precomputed. Therefore, R_{10} is the reconfiguration with the highest utilization value. Hence, R_{10} will be determined next.

6.3.3 Global Reconfiguration Graph

Since it can be assumed that the reconfiguration graphs of the individual cars are often similar, we propose that whenever a new reconfiguration is computed, it shall be made available to other cars as well. Therefore, the idea is to build up a global reconfiguration graph, which is stored in the backend and can be accessed by all cars. Before a car starts precomputing reconfigurations, it checks if the required reconfiguration has already been determined and is, therefore, part of the global reconfiguration graph. Therefore, wasting resources for computing the same configuration multiple times can be avoided.

As illustrated in Figure 6.19, not only can cars participate in extending the reconfiguration graph but also powerful backend servers, which can run more resource-intensive precomputing algorithms.

CHAPTER 7

Related Work

Fail-operational systems, similar to the one presented in this thesis, are also required in other domains. Especially, the avionics area has an inevitable need for such systems. To improve the safety of airplanes and spacecrafts, systems based on duo-duplex, triple-triple, or quadruplex architectures are employed [18].

A duo-duplex architecture, as illustrated in Figure 7.1, 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 [60], can tolerate one failure.

An architecture that can tolerate more failures is the triple-triple redundancy architecture, which is used, for example, by Boeing [66]. A triple-triple system, as shown in Figure 7.2, consists of three homogeneous lanes, whereby each lane includes three distinct computing nodes, which execute distinct software implementations. Furthermore, each lane implements a 2oo3 ("two out of three") voting logic, 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 failures of the remaining lane.

Also, the quadruplex architecture, which was, for example, used in the NASA Space Shuttle [11], can tolerate up to two lane failures. Systems based on this architecture, as illustrated in Figure 7.3, implement four homogenous lanes, whereby each lane consists of a voter, a switch, as well as a computing node, which executes the software implementation.

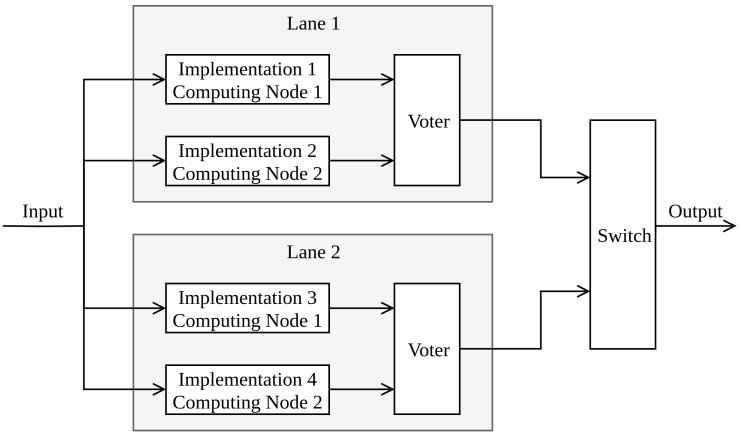

Figure 7.1: Duo-duplex architecture.

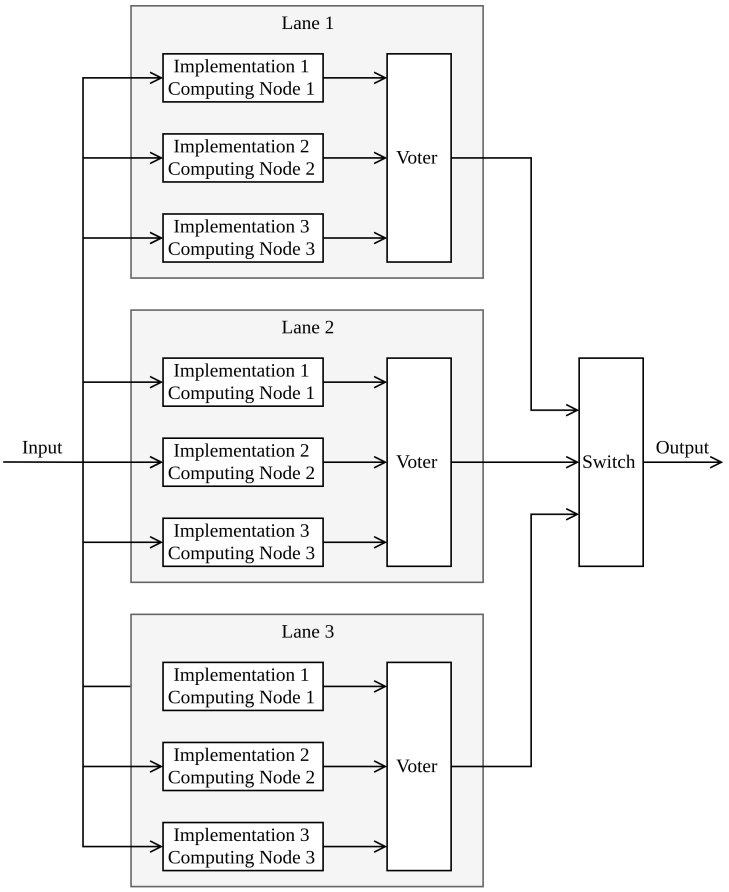

Figure 7.2: Triple-triple architecture.

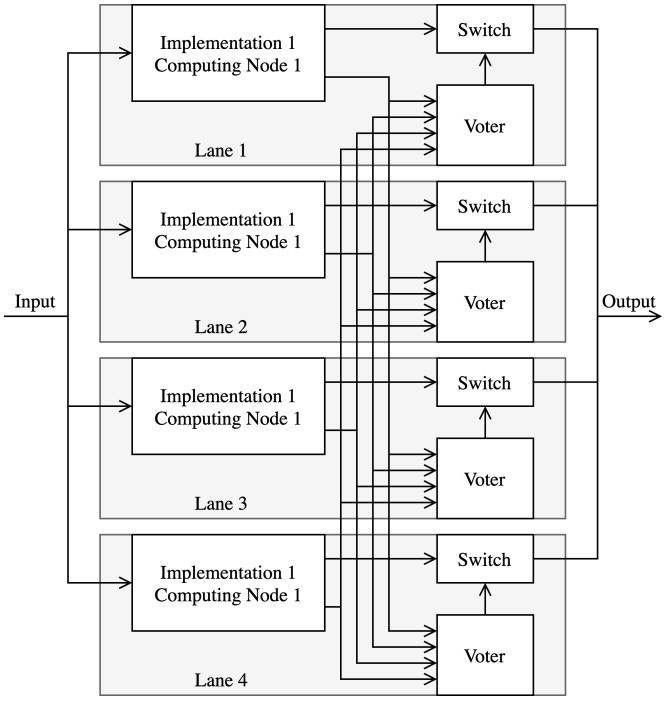

Figure 7.3: Quadruplex architecture.

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

In the past, the fail-operational system designs used in the aviation domain have been used in automotive research as well. Especially, drive-by-wire [31, 53] systems were designed based on these architectural concepts. The problem, however, is that due to the high hardware and software redundancy, implementing those approaches is costly. Since the cost limitations in the automotive industry are much higher than in the aerospace domain [46], an application of the above fail-operational concepts in autonomous vehicles is unlikely.

Nevertheless, as various safety-critical systems are required to operate autonomous vehicles, a fail-operational behavior is inevitable [38, 2]. Furthermore, the safety-criticality of many applications depends on the current context the car is experiencing. For example, the safety-critically, and therefore the required redundancy of the application responsible for detecting pedestrians, is higher in case the vehicle is maneuvering in an urban environment than cruising on a highway. Consequently, employing dynamic fail-operational concepts, like, for instance, FDIRO, is expedient.

Other dynamic fail-operational approaches include, for example, the concept proposed by Becker, Schätz, Armbruster, and Buckl [7], which anticipates a dynamic deployment

of mixed-critically applications. Similar to our approach, to maintain the operation of safety-critical applications, after the occurrence of a failure, other less safety-critical applications are degraded. The focus of their work is on developing a formal method to calculate the optimal application deployment.

Another interesting dynamic fail-operational approach was introduced by Wotawa and Zimmermann [65]. In their work, they present an approach for a rule-based concept for configuring an autonomous vehicle during runtime such that certain preconditions are fulfilled.

CHAPTER 8

Conclusion

In this thesis, we introduced FDIRO ("Fault Detection, Isolation, Recovery, and Optimization"), a fail-operational approach for handling failures in a stepwise fashion. The idea of FDIRO is that in case a failure occurs, a monitoring mechanism first detects the failure. Next, a switchover component isolates the faulty application instance and instructing a redundant application instance to take over the responsibilities of the faulty instance. Due to this switchover, the level of redundancy of the affected application decreases. Therefore, the recovery step of the FDIRO approach attempts to recover the lost redundancy by starting a new application instance. Since the placement of this instance might not be optimal, a subsequent optimization step improves the stability and efficiency of the system by optimizing the application placement, i.e., the assignment between application instances and computing nodes.

FDIRO is designed in such a manner that the complexity of the detection and isolation step is low. Therefore those steps can be performed fast. We demonstrated that under laboratory conditions, the detection and isolation step can be performed within milliseconds.

In future work, a simulator to show the usability of the FDIRO approach would be beneficial [28]. Using such a simulation environment, we can also test further approaches for solving the application-placement problem. Application-placement determiners based on evolutionary game theory [52] and reinforcement learning [10] would be, for example, a promising direction for research.

Furthermore, an integration of the FDIRO approach in a layered context-based reconfiguration approach is an interesting topic for further work. Such an approach defines three interconnected layers, which are distinct by their level of awareness. The top layer, referred to as the *context layer*, is responsible for observing the context. These context observations, in turn, imply a set of requirements, which are the input for the *reconfiguration layer*. This layer is required to determine reconfiguration actions, which

are then executed by the *architecture layer*. An overview of such an approach is discussed in a recent paper [34].

Bibliography

[1] Tobias Achterberg. SCIP: Solving Constraint Integer Programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[2] APTIV, Audi, Baidu, BMW, Continental, Daimler, FCA, HERE, Infineon, Intel, and Volkswagen. Safety First for Automated Driving, 2019. White paper.

[3] Álvaro Arcos-García, Juan A. Álvarez-García, and Luis M. Soria-Morillo. Deep Neural Network for Traffic Sign Recognition Systems: An Analysis of Spatial Transformers and Stochastic Optimisation Methods. *Neural Networks*, 99:158–165, 2018.

[4] 5G Infrastructure Association. 5G Automotive Vision, 2015. White paper.

[5] Aharon Bar Hillel, Ronen Lerner, Dan Levi, and Guy Raz. Recent Progress in Road and Lane Detection: A Survey. *Machine Vision and Applications*, 25(3):727–745, 2014.

[6] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. *CoRR*, abs/1504.05140, 2015.

[7] Klaus Becker, Bernhard Schätz, Michael Armbruster, and Christian Buckl. A Formal Model for Constraint-Based Deployment Calculation and Analysis for Fault-Tolerant Systems. In *Proceedings of the 12th International Conference on Software Software Engineering and Formal Methods* (*SEFM 2014*), 2014.

[8] Klaus Becker, Bernhard Schätz, Christian Buckl, and Michael Armbruster. Deployment Calculation and Analysis for a Fail-Operational Automotive Platform. *CoRR*, abs/1404.7763, 2014.

[9] Sagar Behere and Martin Törngren. A Functional Reference Architecture for Autonomous Driving. *Information and Software Technology*, 73(C):136–150, 2016.

[10] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural Combinatorial Optimization with Reinforcement Learning. *CoRR*, abs/1611.09940, 2016.

[11] 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, 2009.

[12] Zhaowei Cai, Quanfu Fan, Rogerio S. Feris, and Nuno Vasconcelos. A Unified Multi-Scale Deep Convolutional Neural Network for Fast Object Detection. In *Proceeding of 14th European Conference on Computer Vision* (*ECCV 2016*), pages 354–370. Springer, 2016.

[13] Douglas Crockford. The Application/JSON Media Type for JavaScript Object Notation (JSON). *RFC 4627*, 2006.

[14] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[15] 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.

[16] Alessio Fascista, Giovanni Ciccarese, Angelo Coluccia, and Giuseppe Ricci. A Localization Algorithm Based on V2I Communications and AOA Estimation. *IEEE Signal Processing Letters*, 24(1):126–130, 2017.

[17] 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.

[18] Holger Flühr. *Avionik und Flugsicherung*. Springer, 2014.

[19] Sae Fujii, Atsushi Fujita, Takaaki Umedu, Shigeru Kaneda, Hirozumi Yamaguchi, Teruo Higashino, and Mineo Takai. Cooperative Vehicle Positioning via V2V Communications and Onboard Sensors. In *Proceedings of 2011 IEEE Vehicular Technology Conference* (*VTC 2011*), pages 1–5, 2011.

[20] Jürgen Gerstenmeier. Traction Control (ASR)–An Extension of the Anti-Lock Braking System (ABS), 1986. SAE Technical Paper.

[21] David González, Joshué Pérez, Vicente Milanés, and Fawzi Nashashibi. A Review of Motion Planning Techniques for Automated Vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 17(4):1135–1145, 2016.

[22] Google. Angular. `https://angular.io`.

[23] Google. gRPC. `https://grpc.io/`.

[24] Google. OR-Tools. `https://developers.google.com/optimization`.

[25] Google. The Glop Linear Solver. `https://developers.google.com/optimization/lp/glop`.

[26] Gurobi. Gurobi Optimizer. https://www.gurobi.com/products/gurobi-optimizer/.

[27] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[28] 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 International Conference on Connected and Autonomous Driving* (*MetroCAD 2020*), 2020.

[29] Rasheed Hussain and Sherali Zeadally. Autonomous Cars: Research Results, Issues, and Future Challenges. *IEEE Communications Surveys and Tutorials*, 21(2):1275–1313, 2019.

[30] International Organization for Standardization. ISO 26262:2018 Road Vehicles: Functional Safety, 2018.

[31] Rolf Isermann, Ralf Schwarz, and Stefan Stölzl. Fault-Tolerant Drive-by-Wire Systems. *IEEE Control Systems Magazine*, 22(5):64–81, 2002.

[32] Kichun Jo, Junsoo Kim, Dongchul Kim, Chulhoon Jang, and Myoungho Sunwoo. Development of Autonomous Car–Part I: Distributed System Architecture and Development Process. *IEEE Transactions on Industrial Electronics*, 61(12):7131–7140, 2014.

[33] 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.

[34] 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*), 2020.

[35] 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*), 2020.

[36] Tobias Kain, Hans Tompits, Julian-Steffen Müller, Philipp Mundhenk, Maximilan Wesche, and Hendrik Decke. Optimizing the Placement of Applications in Autonomous Vehicles. Submitted to the *31st IEEE Intelligent Vehicles Symposium* (*IV 2020*), 2020.

[37] Andre Kohn, Michael Kasmeyer, Rolf Schneider, Andre Roger, Claus Stellwag, and Andreas Herkersdorf. Fail-Operational in Safety-Related Automotive Multi-Core Systems. In *Proceedings of the 10th IEEE International Symposium on Industrial Embedded Systems* (*SIES 2015*), 2015.

[38] Philip Koopman and Michael Wagner. Challenges in Autonomous Vehicle Testing and Validation. *SAE International Journal of Transportation Safety*, 4(1):15–24, 2016.

[39] Philip Koopman and Michael Wagner. Autonomous vehicle safety: An interdisciplinary challenge. *IEEE Intelligent Transportation Systems Magazine*, 9(1):90–96, 2017.

[40] Yuji Kozaki, Goro Hirose, Shozo Sekiya, and Yasuhiko Miyaura. Electric Power Steering *(*EPS*)*. *Motion & Control*, 6:9–15, 1999.

[41] 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.

[42] Todd Litman. Autonomous Vehicle Implementation Predictions: Implications for Transport Planning. *Victoria Transport Policy Institute Reports*, 2019.

[43] Markus Maurer, J. Christian Gerdes, Barbara Lenz, and Hermann Winner. *Autonomous Driving: Technical, Legal and Social Aspects*. Springer, 2016.

[44] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, Anna Petrovskaya, Mike Pflueger, Ganymed Stanek, David Stavens, Antone Vogt, and Sebastian Thrun. Junior: The Stanford Entry in the Urban Challenge. In *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic*, pages 91–123. Springer, 2009.

[45] 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.

[46] Philipp Nenninger. *Vernetzung verteilter sicherheitsrelevanter Systeme im Kraftfahrzeug*. PhD thesis, Universitätsverlag Karlsruhe, 2007.

[47] 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.

[48] Sergey Orlov. AutoKonf: Forschung und Technologie für automatisiertes und vernetztes Fahren. Talk presented at the Fachtagung "Automatisiertes und vernetztes Fahren", Berlin, 2017.

[49] Thiemo Pasenau, Thomas Sauer, and Jörg Ebeling. Active Cruise Control with Stop & Go Function in the BMW 5 and 6 Series. *ATZ worldwide*, 109(10):6–8, 2007.

[50] Scott Pendleton, Hans Andersen, Xinxin Du, Malika Meghjani, You Eng, Daniela Rus, and Marcelo Jr. Perception, Planning, Control, and Coordination for Autonomous Vehicles. *Machines*, 5(1):1–54, 2017.

[51] Pivotal Software. Spring. `https://spring.io/`.

[52] Yi Ren, Junichi Suzuki, Athanasios Vasilakos, Shingo Omura, and Katsuya Oba. Cielo: An Evolutionary Game Theoretic Framework for Virtual Machine Placement in Clouds. In *Proceedings of the 2014 International Conference on Future Internet of Things and Cloud* (*FiCloud 2014*), 2014.

[53] Oliver Rooks, Michael Armbruster, Armin Sulzmann, Gernot Spiegelberg, and Uwe Kiencke. Duo duplex drive-by-wire computer system. *Reliability Engineering & System Safety*, 89(1), 2005.

[54] SAE International. Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles. In *SAE Standard J3016*, pages 1–16, 2018.

[55] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, Inc., 1986.

[56] Hadas Shachnai and Tami Tamir. On Two Class-Constrained Versions of the Multiple Knapsack Problem. *Algorithmica*, 29(3):442–467, 2001.

[57] 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.

[58] 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, 2007.

[59] Torben Brück. Erstellung eines Messprogramms zur Messung des Datendurchsatzes einer Middleware. IHK Braunschweig, 2019. Project Report.

[60] Pascal Traverse, Isabelle Lacaze, and Jean Souyris. Airbus Fly-by-Wire: A Total Approach to Dependability. In *Building the Information Society*, pages 191–212. Springer, 2004.

[61] 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-97*), pages 893–895, 1979.

[62] Venkat Venkatasubramanian, Raghunathan Rengaswamy, Kewen Yin, and Surya N Kavuri. A Review of Process Fault Detection and Diagnosis, Part I: Quantitative Model-Based Methods. *Computers & Chemical Engineering*, 27(3):293–311, 2003.

[63] Daniel Watzenig and Horn Martin. *Automated Driving*. Springer, 2017.

[64] Junqing Wei, Jarrod M. Snider, Tianyu Gu, John M. Dolan, and Bakhtiar Litkouhi. A Behavioral Planning Framework for Autonomous Driving. *Proceedings of the 2014 IEEE Intelligent Vehicles Symposium* (*IV 2014*), pages 458–464, 2014.

[65] Franz Wotawa and Martin Zimmermann. Adaptive System for Autonomous Driving. In *Proceedings of the 18th IEEE International Conference on Software Quality, Reliability, and Security Companion* (*QRS-C 2018*), pages 519–525, 2018.

[66] Y.C. Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference* (*AeroConf 1996*), pages 293–307, 1996.

[67] Qin Zhang, Xuegao An, Jin Gu, Binquan Zhao, Dazhi Xu, and Shuren Xi. Application of FBOLES – A Prototype Expert System for Fault Diagnosis in Nuclear Power Plants. *Reliability Engineering & System Safety*, 44(3):225–235, 1994.

[68] Ali Zolghadri. Advanced Model – Based FDIR Techniques for Aerospace Systems: Today Challenges and Opportunities. *Progress in Aerospace Sciences*, 53:18–29, 2012.

[69] Ömer Şahin Taş, Florian Kuhnt, J. Marius Zöllner, and Christoph Stille. Functional System Architectures towards Fully Automated Driving. In *Proceedings of the 2016 IEEE Intelligent Vehicles Symposium* (*IV 2016*), pages 304–309, 2016.