FDIRO: A General Approach for a Fail-Operational System Design

Tobias Kain

Volkswagen Group Innovation, Volkswagen AG, Germany. E-mail: tobias.kain@volkswagen.de

Hans Tompits

Institute of Logic and Computation, Technische Universität Wien, Austria. E-mail: tompits@kr.tuwien.ac.at

Julian-Steffen Müller

Volkswagen Group Innovation, Volkswagen AG, Germany. E-mail: julian-steffen.mueller@volkswagen.de

Philipp Mundhenk

Autonomous Intelligent Driving GmbH, Germany. E-mail: philipp.mundhenk@aid-driving.eu

Maximilian Wesche

Volkswagen Group Innovation, Volkswagen AG, Germany. E-mail: maximilian.wesche@volkswagen.de

Hendrik Decke

Volkswagen Group Innovation, Volkswagen AG, Germany. E-mail: hendrik.decke@volkswagen.de

Full vehicle autonomy excludes a takeover by passengers in case a safety-critical application fails. Therefore, the system responsible for operating the autonomous vehicle has to detect and handle failures by itself to ensure the safety of the passengers and other road users. Especially in the initial phase of the availability of autonomous vehicles, building up consumer confidence is essential. Therefore, handling failures by stopping the vehicle is not desirable as such a behavior will unsettle customers. In this paper, we introduce FDIRO (standing for "Fault Detection, Isolation, Recovery, and Optimization") a fail-operational approach for handling failures in a stepwise fashion that is based on the FDIR ("Fault Detection, Isolation, and Recovery") approach known from the aerospace domain, whereby we reimplemented the steps defined by FDIR to fit the use case of autonomous driving. We designed the failure detection and isolation procedure to be of low complexity so that these steps can be performed within milliseconds. The subsequent recovery step then transforms the system configuration such that the demanded safety requirements are satisfied. However, since the resulting configuration might not be optimal regarding the current context that the autonomous vehicle is experiencing, 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 the current context.

Keywords: autonomous vehicles, fail-operational, system reconfiguration, application placement, optimization.

1. Introduction

Nowadays, vehicles are equipped with various advanced driver assistance systems that support the driver while operating the vehicle. Actions that modern vehicles are capable of doing include, e.g., 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 required to monitor their behavior and take over control, if required (Brookhuis et al. (2019)).

As far as fully autonomous vehicles are concerned, also referred to as *SAE Level 5 vehicles* or, in case the vehicle drives in a defined environment, as *SAE Level 4 vehicles* (SAE International (2018)), any takeover actions by passengers are excluded. To operate such autonomous vehicles, numerous software applications, including, e.g., 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.

Although handling faults by performing an emergency stop is feasible, such a behavior is

Proceedings of the 30th European Safety and Reliability Conference and the 15th Probabilistic Safety Assessment and Management Conference
Edited by Piero Baraldi, Francesco Di Maio and Enrico Zio
Published by Research Publishing, Singapore.
ISBN: 978-981-14-8593-0; doi:10.3850/978-981-14-8593-0

not always desirable since this will cause customer satisfaction to decline (Koopman and Wagner (2017)). Furthermore, in case the vehicle is, for example, moving at a very high speed or driving in a tunnel, it might not be safe to stop abruptly.

On the other hand, increasing the fault-acceptance rate might cause vehicles to operate unintendedly, leading to a loss of customer confidence. Therefore, solutions to handle this are required.

In this paper, we introduce FDIRO (standing for "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 a safety-critical reconfiguration. This approach is based on the FDIR ("Fault Detection, Isolation, and Recovery") approach known from the aerospace domain (Zolghadri (2012)), which is also applied by other disciplines such as chemical (Venkatasubramanian et al. (2003)) and nuclear engineering (Zhang et al. (1994)).

An essential characteristic of FDIR is the redundant design of safety-critical applications. FDIRO adopts this characteristic by defining different operation modes for redundant application instances, whereby those modes differ by their level of degradation.

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, which satisfies the demanded safety requirements. However, since the resulting configuration might not be optimal in terms of, e.g., system utilization, 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.

The execution of the FDIRO procedure after the occurrence of a failure causes the system state to be raised from *safety-critical* to *safe* due to the execution of the first three steps of FDIRO. The subsequent optimization step further upgrades the system state to *safe & optimized*. Figure 1 illustrates the different system states while performing the FDIRO procedure.

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, in some cases, 90 ms, according to Orlov (2017). FDIRO meets this requirement due to keeping the complexity of the detection and isolation step low, enabling that those steps can be performed within milliseconds.

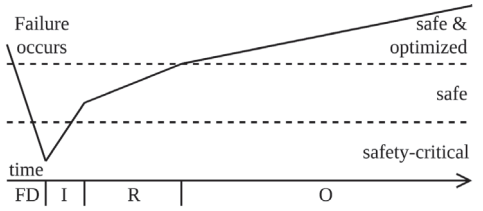


Fig. 1.: System states after the occurrence of a failure.

Compared to the detection and isolation step of FDIRO, the time criticality of the subsequent recovery and optimization step is lower as the prior executed isolation step ensures a safe system state. Therefore, the actions performed during the recovery and optimization procedure can be more complex and thus more time-consuming.

The paper is organized as follows: In Section 2, we provide background information about functional safety and discuss the difference between fail-safe and fail-operational systems. Afterwards, in Section 3, we introduce our stepwise reconfiguration approach FDIRO. In Section 4, we show, using a proof-of-concept implementation, that the detection and isolation step can be performed within milliseconds. In Section 5, we compare our work to related approaches, and the paper concludes with Section 6 in which we discuss the future development of FDIRO.

2. Functional Safety

Even though the vehicles currently available on the consumer market are far from being considered fully automated, according to the definition of SAE (SAE International (2018)), 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 (Kohn et al. (2015)).

To reduce this risk, functional safety requirements of road vehicles have been introduced by the ISO 26262 standard (International Organization for Standardization (2018)). This standard defines functional safety as the absence of unreasonable risks due to hazards caused by malfunctioning behavior of E/E systems. To achieve this goal, ISO 26262 introduces, e.g., 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 deactivate the complete system

in case a failure is detected. Such systems are referred to as *fail-safe systems*. In particular, traction control (Gerstenmeier (1986)), power steering (Kozaki et al. (1999)), as well as adaptive cruise control systems (Pasenau et al. (2007)) are designed to be fail-safe. For example, in case an adaptive cruise control system fails, the driver is warned that this function is disabled. Therefore, the task of keeping the speed and distance to a preceding vehicle is handed over to the driver.

Although a fail-safe behavior is acceptable in today's vehicles, such behavior is insufficient for SAE Level 4 and SAE Level 5 vehicles since any takeover actions by the passengers are excluded. Instead, fully autonomous vehicles require *fail-operational systems*. Compared to fail-safe systems, fail-operational systems ensure a correct and safe operation even if the system is affected by failures (APTIV et al. (2019)).

3. FDIRO: A Stepwise Reconfiguration Approach

3.1. *Overview*

Our approach for handling occurring failures is based on FDIR ("Fault Detection, Isolation, and Recovery") (Zolghadri (2012)), whereby we redesigned the FDIR concept to fit the use case of autonomous driving. The idea is to handle failures in a stepwise manner by executing the following four procedures:

(1) detection of the failure,
(2) isolation of the failure by a switchover between redundant instances,
(3) recovery of the safety requirements, and
(4) optimizing the placement of the application instances.

The first three steps correspond to the *detection*, *isolation*, and *recovery actions* as specified by FDIR. The execution of those steps results in an error-free configuration, which satisfies the needed safety requirements. However, since the resulting configuration might not be optimal, we extended the FDIR strategy by an additional optimization step. We refer to this extended FDIR approach as FDIRO ("Fault Detection, Isolation, Recovery, and Optimization").

The newly introduced optimization step aims to enhance the system according to goals that depend on the current situation. For example, for the case of an autonomous electric vehicle, a conceivable goal would be an 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.

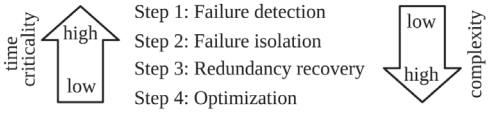


Fig. 2.: Comparison of time criticality and computational complexity of the FDIRO steps.

Note that the optimization procedure can also be initiated by other events apart from failures. For instance, a change in the driving environment can cause triggering the optimization of the application placement. However, for the sake of simplicity, we do not consider other events aside from failures in this paper.

The motivation for implementing a stepwise reconfiguration is that, for some failures, the reconfiguration time is critical. In particular, certain cases require that the time until a failure is isolated and a redundant instance takes over the actions of the faulty instance must lie within a range of milliseconds.

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

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

Fast failure isolation by switching over to a redundant instance is ensured due to the implementation of different operation modes. The idea is that each safety-critical application is executed redundantly, whereby one application instance is executing in *active operation mode* and is interacting with other applications. The other redundant application instances are executing in *active-hot operation mode*. Therefore, these instances are not communicating with other active instances or controlling actuators. Consequently, they can perform a degraded set of operations. However, active-hot instances are capable of taking over the responsibilities of the active instance within milliseconds if required.

In contrast to the detection and isolation steps, the complexity of the redundancy recovery step as well as of the optimization step is significantly higher. However, since the time criticality of the latter steps is lower than that of the former steps, such a behavior is acceptable. Figure 2 illustrates this complementary effect.

Another possible approach to enable short reconfiguration times is to precompute successor configurations for all possible failures that can occur. However, as we argue next, such an approach is not feasible since today's storage capacities are insufficient.

Assume that N is the set of computing nodes, and I is the set of application instances, i.e., all active and active-hot instances that are executed by the system. Moreover, let us call a *configuration* an assignment 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 and 0 otherwise.

We demonstrate that precomputing and storing all configurations is not feasible by showing that this strategy is not even feasible in case we only assume failures of computing nodes and application instances. Considering only these two types of failures, the number of possible failures is $|N| + |I|$. Since the reconfiguration actions after a failure occurred depend on all previous failures, 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 an $|N| \times |I|$ 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 instances, i.e., $|I| = 100$, about $5 \cdot 10^{169}$ bytes of storage are required.

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

Nevertheless, precomputing configurations for a fixed set of failures is feasible. Indeed, one may precompute the successor configurations of failures which are of high severity and are likely to occur. In case such a failure occurs, the vehicle can reconfigure the system according to a precomputed configuration. Hence, those failures can be handled in minimum time. In case no successor configuration has been precomputed, the configuration has to be computed during runtime.

Figure 3 illustrates the concept of precomputing configurations graphically. In case a failure occurs, the FDIRO procedure transfers the current system configuration (indicated by the black node) to an adjacent configuration which can be either a precomputed one (shown by a grey node) or a not yet precomputed one (a white node). Afterwards, the precomputing procedure computes not yet precomputed configurations in the neighborhood of the new current configuration.

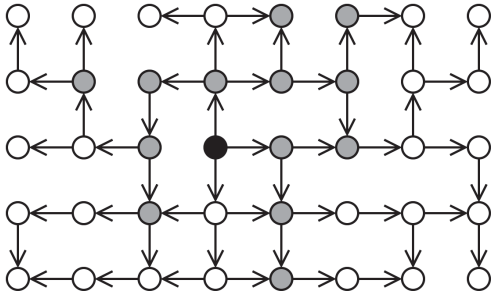


Fig. 3.: Partially precomputed configuration graph (edges: failures; black node: current configuration; gray nodes: precomputed configurations; white nodes: not precomputed configurations).

3.2. *Workflow of* FDIRO

We define several logical components that are responsible for performing the FDIRO procedure. Figure 4 illustrates the workflow of FDIRO in an activity diagram. Note that all components which are responsible for performing the FDIRO procedure have to be fail-operational.

The components that detect failures and consequently trigger a reconfiguration are called *monitors*. We distinguish between *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 includes an application-instance monitor that monitors 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 any of those different types of monitors detect a failure, they inform a *switchover component* about the affected application instances. The switchover component instructs the faulty instance to switch to the isolated operation mode, which prevents the faulty instance from communicating with other applications or controlling actuators.

Next, the switchover component determines whether, for the faulty application instance, a redundant instance exists. In case such an instance exists, the switchover component instructs the selected instance to take over the responsibilities of the faulty instance.

In case that no redundant instance exists, the switchover component has to evaluate whether the failure of the instance causes the safety level

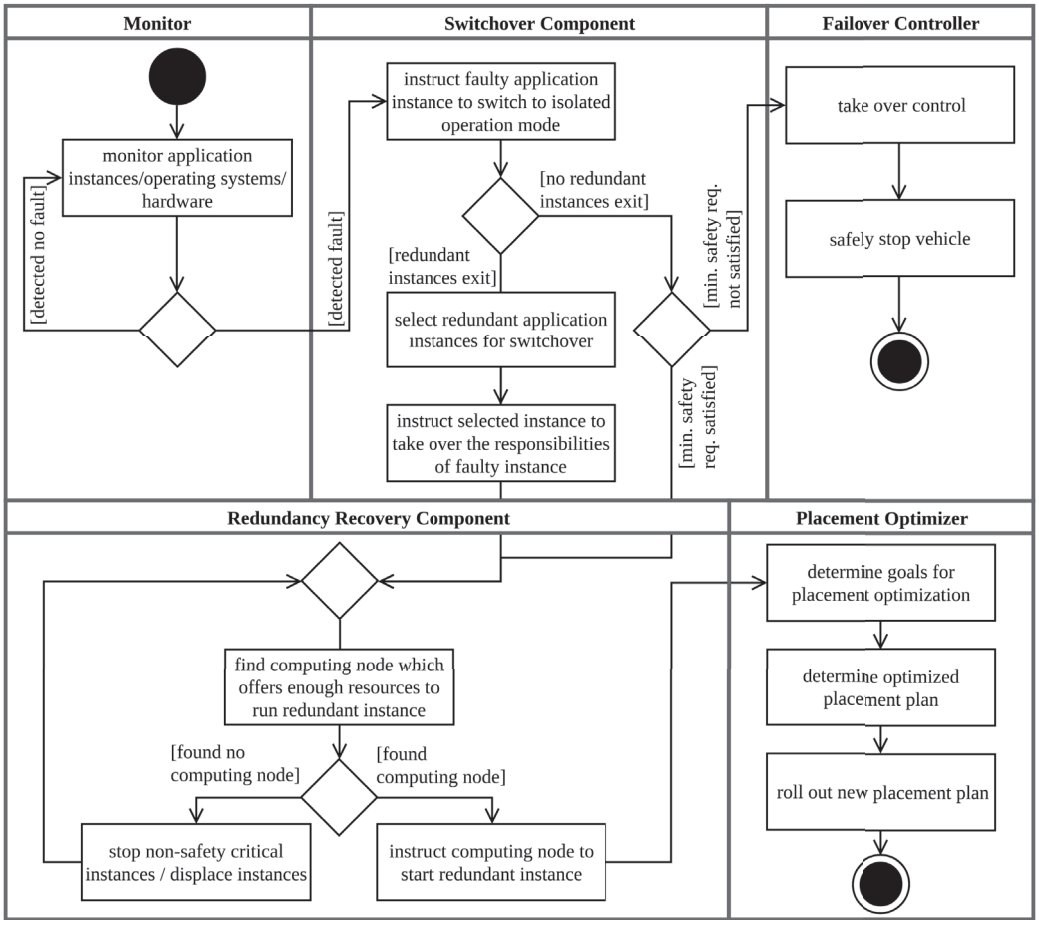


Fig. 4.: The activity diagram showing the workflow of FDIRO.

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*, which is responsible for restoring the lost redundancy.

To achieve this, the redundancy recovery component obtains a load profile overview of all available computing nodes. Using this information, the redundancy recovery component determines whether one of the computing nodes offers enough available resources to run an instance that implements the same functionality as the faulty instance. If such a computing node exits, the redundancy recovery component instructs that node to start a redundant instance.

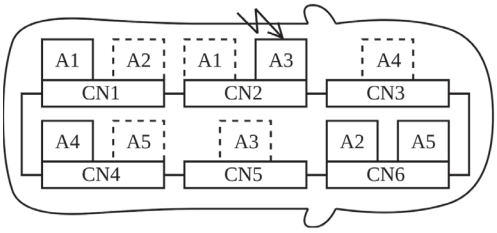
In case that no computing node has sufficient resources left, the redundancy recovery component is allowed to displace application instances and stop non-safety critical application instances.

The actions performed by the redundancy recovery component can be considered 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 an additional step, the placement optimizer tries to find an application placement that is better suited for the current driving situation. To this end, the placement optimizer first determines the goals the application placement shall meet. Next, a placement plan that satisfies the previously defined goals is computed. Finally, the placement optimizer rolls out the new placement plan to all computing nodes, which then perform the actions defined by this plan.

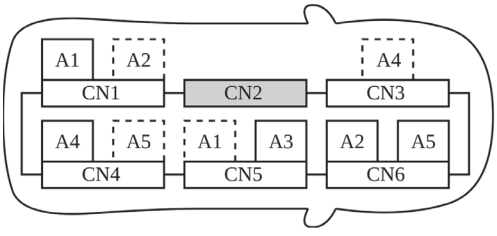
To further clarify the tasks performed by FDIRO, we next give a concrete example.

3.3. FDIRO *Example*

Consider the system illustrated in Figure 5a which executes five applications, where for each application, one active instance and one active-hot



(a) Initial configuration, where the active application instance of A3 failed.



(b) Configuration after the FDIRO procedure was performed.

Fig. 5.: Workflow example of FDIRO.

instance (denoted by dashed lines) exists. We assume that the active instance of application A3, which is executed by computing node CN2, fails.

Once the failure occurred, the application-instance monitor has to become aware of the malfunctioning of the active instance of A3. 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 instance. Once the previously active instance 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 instance of application A3 is available. In our scenario, this is the case since computing node CN5 executes a redundant instance of A3. So, the switchover component selects this instance to upgrade its operation mode to active.

Due to the isolation and switchover procedure, A3 is not executed redundantly anymore. Therefore, in the third step of the FDIRO process, the redundancy recovery component tries to recover the lost redundancy. Assuming that computing node CN3 and CN5 are the only two nodes offering enough resources for an additional A3 instance, the redundancy recovery component will select CN3 for executing the redundant instance of A3. Now, computing node CN3 is preferred over CN5 since the latter already executes the active instance of A3. Therefore, selecting computing node CN3 increases the hardware segregation, which results in an improvement of reliability.

In the fourth step of the FDIRO process, the control is handed over to the placement optimizer. 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 A3 instance and migrate the active-hot instance of A1 to computing node CN5. As a result, computing node CN2 can be turned off, since this node executes no applications anymore. The optimized application placement is displayed in Figure 5b.

4. Monitor and Switchover Component Test Implementation

To prove the hypothesis that the fault detection and isolation step of FDIRO can be performed within milliseconds, we performed an experiment using a proof-of-concept implementation.

For the test set-up, we used six identical computing nodes (with OS Ubuntu 18.04.1 LTS, CPU Intel Pentium Silver J5005, and 8 GB memory), which are connected via Gigabit Ethernet. Three computing nodes execute a safety-critical application, whereby one computing node runs the active instance, and the other two execute an active-hot instance. For the sake of simplicity, we do not use an autonomous driving application. Instead, we implemented an application that adds two integers. Moreover, we equipped the active instance with a function that allows us to corrupt its results.

The remaining three computing nodes execute an application-instance monitor, a switchover component, and a *measuring component*. The application-instance monitor sends periodically a test case containing two random integers to the active and the two active-hot application instances. Once an application instance receives a test case, it adds the two integers and returns the result to the application-instance monitor. A failure of the active instance is detected if the result returned by this instance is different from the results returned by the active-hot instances, and the results returned by the two active-hot instances are equal. In this case, the application-instance monitor reports to the switchover component that the active instance is malfunctioning. The switchover component then instructs the faulty active instance to switch to the isolated operation mode. Furthermore, one of the active-hot instances is instructed to upgrade its operation mode to active.

The measuring component simulates the interaction of the application with other applications executed by the system. Therefore, the active instance sends every millisecond a message to the measuring component. This message contains the unique application-instance identifier as well as a flag that signals whether the instance is working correctly. Using the receiving times of these messages, the *takeover time*, i.e., the time between the occurrence of the failure and the takeover by one of the active-hot instances, is determined.

Table 1.: The test results of the experiment.

Fault-Detection Sampling Rate	Average Takeover Time
5 ms	8.5 ms
10 ms	14.4 ms
15 ms	16.2 ms
20 ms	16.3 ms
25 ms	19.3 ms
30 ms	22.4 ms
35 ms	21.2 ms
40 ms	30.0 ms
45 ms	27.2 ms
50 ms	36.4 ms

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

Table 1 shows the results of the performed tests. The average takeover time for a specific fault-detection sampling rate is determined based on 100 test iterations.

We can observe that, on average, the takeover time is about half of the fault-detection sampling rate plus an offset of approximately 7 ms. Assuming that the 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 takeover time will converge to half of the fault-detection sampling rate as the number of test runs increases. Hence, 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 approximately 7 ms.

5. Related Work

Fail-operational systems, similar to the one presented in this paper, are also required in other domains. In particular, the avionics area has an inevitable need for such systems (Flühr (2014)). To improve the safety of airplanes and spacecrafts, systems based on duo-duplex (Traverse et al. (2004)), triple-triple (Yeh (1996)), or quadruplex architectures (Blair-Smith (2009)) are employed. Those architectural concepts require a high level of software and hardware redundancy. For example, triple-triple systems, as, e.g., used by Boeing, consists of three homogeneous lanes, whereby each lane includes three distinct computing nodes which execute distinct software implementations.

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 systems (Isermann et al. (2002); Rooks et al. (2005)) 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 (Nenninger (2007)), an application of the above concepts in autonomous vehicles is unlikely.

Nevertheless, as various safety-critical systems are required to operate autonomous vehicles, a fail-operational behavior is inevitable (Koopman and Wagner (2016); APTIV et al. (2019)). 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 an 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, e.g., the concept proposed by Becker et al. (2014), which addresses 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 (2018). In their work, they present a rule-based methodology for configuring an autonomous vehicle during runtime such that certain preconditions are fulfilled.

6. Conclusion

In this paper, we introduced FDIRO ("Fault Detection, Isolation, Recovery, and Optimization"), a fail-operational approach for handling failures in a stepwise fashion. FDIRO allows a fast isolation of an occurring failure and a recovery of the lost functionality. Furthermore, FDIRO provides means to recover and optimize the system state.

In our future research activities, we plan to implement a simulator to show the efficacy of the FDIRO approach. Using this simulation environment, we can also test optimization strategies based on, e.g., integer linear programming (Kain et al. (2020)), evolutionary game theory (Ren et al. (2014)), or reinforcement learning (Bello et al. (2016)), calculating application placements. Moreover, we plan to integrate the FDIRO approach in a layered context-based reconfiguration approach. This approach defines three interconnected layers, which we differentiate by their level of awareness (Kain et al. (2020)).

Further open issues not yet addressed by our approach include research about the monitoring of software components, the effects of multiple failures that occur simultaneously, and the validation

of reconfiguration and optimization actions.

References

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

Becker, K., B. Schätz, M. Armbruster, and C. Buckl (2014). A formal model for constraint-based deployment calculation and analysis for fault-tolerant systems. In *Proceedings of the 12th International Conference on Software Engineering and Formal Methods* (*SEFM 2014*).

Bello, I., H. Pham, Q. V. Le, M. Norouzi, and S. Bengio (2016). Neural combinatorial optimization with reinforcement learning. *CoRR abs/1611.09940*.

Blair-Smith, H. (2009). Space Shuttle fault tolerance: Analog and digital teamwork. In *Proceedings of the 28th IEEE/AIAA Digital Avionics Systems Conference* (*DASC 2009*), pp. 6.B.1–1–6.B.1–11.

Brookhuis, K. A., D. De Waard, and W. H. Janssen (2019). Behavioural impacts of advanced driver assistance systems–An overview. *European Journal of Transport and Infrastructure Research 1*(3).

Flühr, H. (2014). *Avionik und Flugsicherung*. Springer.

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

International Organization for Standardization (2018). *ISO 26262:2018 Road vehicles – Functional Safety*.

Isermann, R., R. Schwarz, and S. Stölzl (2002). Fault-tolerant drive-by-wire systems. *IEEE Control Systems Magazine 22*(5), 64–81.

Kain, T., J.-S. Müller, P. Mundhenk, H. Tompits, M. Wesche, and H. Decke (2020). Towards a reliable and context-based system architecture for autonomous vehicles. In *Proceedings of the 2nd International Workshop on Autonomous Systems Design* (*ASD 2020*).

Kain, T., H. Tompits, J.-S. Müller, P. Mundhenk, M. Wesche, and H. Decke (2020). Optimizing the Placement of Applications in Autonomous Vehicles. Submitted draft.

Kohn, A., M. Kasmeyer, R. Schneider, A. Roger, C. Stellwag, and A. Herkersdorf (2015). Fail-operational in safety-related automotive multi-core systems. In *Proceedings of the 10th IEEE International Symposium on Industrial Embedded Systems* (*SIES 2015*).

Koopman, P. and M. Wagner (2016). Challenges in Autonomous Vehicle Testing and Validation. *SAE International Journal of Transportation Safety 4*(1), 15–24.

Koopman, P. and M. Wagner (2017). Autonomous vehicle safety: An interdisciplinary challenge. *IEEE Intelligent Transportation Systems Magazine 9*(1), 90–96.

Kozaki, Y., G. Hirose, S. Sekiya, and Y. Miyaura (1999). Electric power steering (EPS). *Motion & Control 6*, 9–15.

Nenninger, P. (2007). *Vernetzung verteilter sicherheitsrelevanter Systeme im Kraftfahrzeug*. Ph. D. thesis, Universitätsverlag Karlsruhe.

Orlov, S. (2017). AutoKonf: Forschung und Technologie für automatisiertes und vernetztes Fahren. Talk presented at the Fachtagung "Automatisiertes und vernetztes Fahren", Berlin.

Pasenau, T., T. Sauer, and J. Ebeling (2007). Active cruise control with stop&go function in the BMW 5 and 6 series. *ATZ worldwide 109*(10), 6–8.

Ren, Y., J. Suzuki, A. Vasilakos, S. Omura, and K. Oba (2014). 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*).

Rooks, O., M. Armbruster, A. Sulzmann, G. Spiegelberg, and U. Kiencke (2005). Duo duplex drive-by-wire computer system. *Reliability Engineering & System Safety 89*(1).

SAE International (2018). Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles. SAE Standard J3016.

Traverse, P., I. Lacaze, and J. Souyris (2004). Airbus fly-by-wire: A total approach to dependability. In *Proceedings of the IFIP 18th World Computer Congress*, Volume 156 of *IFIP International Federation for Information Processing*, pp. 191–212. Springer.

Venkatasubramanian, V., R. Rengaswamy, K. Yin, and S. N. Kavuri (2003). A review of process fault detection and diagnosis, Part I: Quantitative model-based methods. *Computers & Chemical Engineering 27*(3), 293–311.

Wotawa, F. and M. Zimmermann (2018). Adaptive system for autonomous driving. In *Proceedings of the 18th IEEE International Conference on Software Quality, Reliability, and Security Companion* (*QRS-C 2018*), pp. 519–525.

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

Zhang, Q., X. An, J. Gu, B. Zhao, D. Xu, and S. Xi (1994). Application of FBOLES–A prototype expert system for fault diagnosis in nuclear power plants. *Reliability Engineering & System Safety 44*(3), 225–235.

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