Technische Universität München
TUM School of Computation, Information and Technology

# Fail-Operational Decentralized System Architectures

**Philipp Weiß**

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

**Doktors der Ingenieurwissenschaften (Dr.-Ing.)**

genehmigten Dissertation.

**Vorsitz:**
Prof. Dr.-Ing. Gerhard Rigoll

**Prüfer*innen der Dissertation:**
1. Prof. Dr. Sebastian Steinhorst
2. apl. Prof. Dr.-Ing. Walter Stechele

Die Dissertation wurde am am 21.06.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 16.10.2023 angenommen.

# Abstract

To make autonomous driving feasible, it's vital to have a system that can continue to function even in the event of a failure. With no driver present to take over in such a scenario, relying solely on fail-safe methods is insufficient. Fail-operational systems must be in place to ensure critical functions remain safe and operational even if hardware fails. However, including additional hardware components to ensure redundancy is an expensive proposition. Combined with the increased resource demands of autonomous driving functions, the hardware costs for automotive manufacturers would rise substantially.

Simultaneously, the automotive sector is facing growing customer demands and expectations. Customers will require the most up-to-date functionality through over-the-air software updates, and as a result, automotive software systems will become highly customizable. Automotive manufacturers will need to integrate an increasing number of applications into their Electrical/Electronic (E/E) architecture, and each system will be made up of a distinct and personalized configuration.

To handle the rising complexity of software, upcoming software will be designed with a modular approach, as opposed to the current monolithic Electronic Control Unit (ECU) software. Modular software design enables the dynamic movement of software components between ECUs cores, such as activating and deactivating components on different ECUs cores. This perspective provides new possibilities for implementing fail-operational systems. A system-wide software platform can manage redundancy and fail-operational behavior at run-time.

In our system model, software applications consist of interconnected tasks executed on ECU cores and messages sent over Ethernet links to enable communication between tasks. We consider failure scenarios where ECUs are assumed to be faulty, including the loss of any tasks mapped to the corresponding ECU. To keep safety-critical applications operational in such a scenario, redundancy is required.

Instead of using active redundancy, where a task replica is actively executed, or conventional passive redundancy, where a backup task instance is activated on failure, we propose to use graceful degradation. With traditional passive redundancy, the system has to be oversized to make sufficient resources available once a passive task is started. With graceful degradation, system resources are dynamically repurposed at run-time to ensure adequate resources are available for critical applications. Resources allocated by non-critical applications are a backup guarantee for critical applications. As a result, non-critical functionality might have to be shut down in a failure scenario to keep critical functionality operational. Therefore, by applying graceful degradation, fail-operational requirements can be fulfilled while lowering the hardware demand at the cost of risking the loss of non-critical functionality in a failure scenario.

State-of-the-art design-time methods are not applicable to achieve this functionality as the system needs to react and recover dynamically on failures and events. Furthermore, software functionality might change on demand, so the system has to respond flexibly to changes. To cope with the high amount of such possible configurations, we need to make the system self-aware, which can be enabled by applying agent-based strategies. In an agent-based system, the system's control is decentralized, which has the advantage that there is no single point of failure and thus makes the control itself fail-operational when adequately designed. In dynamic mapping, an agent is responsible for finding the mapping of a single application or task at run-time. However, the main challenge when designing such a system is to achieve predictable behavior and to find bounds within which a safe and dynamic operation is possible.

For this purpose, this thesis proposes an agent-based fail-operational approach utilizing graceful degradation. It addresses challenges in timing analysis, checkpointing, and reliability analysis to achieve a more predictable behavior. Our contributions are outlined in the following.

First, we introduce and analyze the effectiveness of an agent-based approach that finds application mappings at run-time, ensures the fail-operational behavior of safety-critical applications by using graceful degradation, and reconfigures itself after ECU failures. Our results indicate that the number of tolerated ECU losses until a safety-critical application fails can be significantly improved without adding redundant hardware resources. We then present a performance analysis that can analyze timing constraints of fail-operational distributed applications using graceful degradation. Our method can verify that even during a critical ECU failure, a backup solution that adheres to end-to-end timing constraints is always available. Furthermore, we present a dynamic decentralized mapping procedure that performs constraint solving at run-time using our analytical approach combined with a backtracking algorithm. In our experimental setup, our graceful degradation approach can fit about double the number of critical applications on the same architecture compared to an active redundancy approach.

While our approach ensures a behavior within timing constraints before and after a failure, an application might produce no output for a specific time during a failover. Here, it also has to be ensured that a failover within the Fault Tolerant Time Interval (FTTI) can be guaranteed. For this purpose, we analyze the impact of failover on the timing behavior of distributed fail-operational applications and derive an upper bound for the worst-case failover time. Instead of performing time-consuming experiments, our formal analysis can be used to evaluate whether a mapping would meet the failover timing constraints or not such that an evaluation at run-time is possible. We support our formal analysis by conducting experiments on a hardware platform using a distributed fail-operational neural network. Our randomly generated worst-case results are as close as 6.0% below our analytically derived exact bound.

Another major challenge is that most applications have a state that might get lost during a failure such that a recovery might be impossible. Thus, in a system with passive redundancy, periodic checkpointing with rollback recovery can send the state to another ECU, where the application can be restarted after a failure. Here, the challenge is to find a suitable checkpointing period such that network and processing overhead caused by

sending the checkpoint is minimized but to ensure that application-specific constraints are still met. We present an approach to derive the maximum checkpointing period analytically by giving an upper bound on the number of missed computational steps due to failure effects. As the implications of the state data age are application-specific, we use a Simultaneous Mapping and Localization (SLAM) algorithm, an application commonly used in autonomous systems such as robots or self-driving cars, as a real-world example to determine the effects on the quality of the application. The worst-case results of our case study using a SLAM application are consistent with our analytically derived exact bound. Overall, using our approach, a maximum achievable checkpointing period can be determined to reduce network overhead to achieve a cost-efficient and safe behavior of autonomous systems.

Last, we investigate the impact of graceful degradation on the reliability of critical and non-critical applications. As resources are shifted dynamically, non-critical applications not directly affected by an ECU failure might get shut down to free resources for restarting critical tasks. Thus, with a graceful degradation approach, the reliability of critical applications is increased at the cost of a decrease in the reliability of non-critical applications. To design such a system efficiently, it is essential to quantify and understand the effect of a graceful degradation approach on critical and non-critical applications to increase predictability. Therefore, we present an in-depth trade-off analysis of a graceful degradation approach where we analyze the resource consumption and the impact of graceful degradation on the reliability of critical and non-critical applications.

Overall, our fail-operational approach enables a dynamic and safe behavior of automotive applications. Our analytical approaches help developers to create more cost-efficient and predictable systems. By following our methodologies, developers can efficiently analyze the impact of graceful degradation and meet a trade-off according to their system requirements. Our findings confirm that using graceful degradation can tremendously reduce cost compared to conventional redundancy approaches with no negative impact on the reliability of critical applications if a reliability reduction of non-critical applications is acceptable.

# Deutsche Kurzfassung (German Abstract)

Um autonomes Fahren zu ermöglichen, ist die Gewährleistung eines *fail-operational* Verhaltens der Software und E/E-Architektur entscheidend. Da beim autonomen Fahren ein Fahrer als Rückfallebene in einem Ausfallszenario fehlt, reicht es nicht aus *fail-safe* Ansätze zu verwenden. *Fail-operational* Systeme müssen den sicheren Betrieb von kritischen Anwendungen auch bei Hardwareausfällen gewährleisten. Eine einfache Erweiterung von Hardwarekomponenten zur Gewährleistung der Redundanz ist eine kostspielige Lösung. In Verbindung mit dem erhöhten Ressourcenbedarf der autonomen Fahrfunktionen würden die Hardwarekosten für die Automobilhersteller daher erheblich steigen.

Gleichzeitig sieht sich die Automobilindustrie mit einer zunehmenden Anzahl von Kundenbedürfnissen und -anforderungen konfrontiert. In Zukunft wird Software im Auto in hohem Maße anpassbar sein, und Kunden werden die neuesten Funktionen durch Over-the-Air-Software-Updates verlangen. Daher müssen Automobilhersteller immer mehr Anwendungen in ihre E/E-Architektur integrieren, wobei jedes System aus einer einzigartigen und maßgeschneiderten Konfiguration bestehen wird.

Um die zunehmende Komplexität zu bewältigen, wird zukünftige Software im Gegensatz zur aktuellen monolithischen Steuergerätesoftware modular aufgebaut sein. Ein modulares Softwaredesign ermöglicht die dynamische Verschiebung von Softwarekomponenten zur Laufzeit, einschließlich der Aktivierung und Deaktivierung von Komponenten auf verschiedenen Steuergeräten. Diese Perspektive ermöglicht neue Strategien zur Implementierung von *fail-operational* Systemen. Hier können Redundanz und *fail-operational* Verhalten von einer systemweiten Softwareplattform zur Laufzeit verwaltet werden.

In unserem Systemmodell bestehen Softwareanwendungen aus miteinander verbundenen Tasks, die auf ECU-Kernen ausgeführt werden, und Nachrichten, die über Ethernet-Verbindungen gesendet werden, um die Kommunikation zwischen den Tasks zu ermöglichen. Wir betrachten Ausfallszenarien, bei denen ganze ECUs als fehlerhaft angenommen werden, was den Verlust aller Tasks einschließt, die auf den entsprechenden ECU ausgeführt wurden. Um sicherheitskritische Anwendungen in einem solchen Szenario betriebsbereit zu halten, ist Redundanz erforderlich.

Anstelle von aktiver Redundanz, bei der eine zweite Instanz einer Anwendung aktiv ausgeführt wird, oder passiver Redundanz, bei der im Falle eines Ausfalls eine Backup-Instanz aktiviert wird, schlagen wir die Verwendung von *graceful degradation* vor. Bei der herkömmlichen passiven Redundanz muss das System so überdimensioniert sein, dass genügend Ressourcen verfügbar sind, sobald eine passive Backup-Instanz gestartet wird. Bei *graceful degradation* hingegen werden die Systemressourcen während der Laufzeit dynamisch umverteilt, um sicherzustellen, dass genügend Ressourcen für kritische Anwendungen zur Verfügung stehen. Ressourcen, die von nicht-kritischen Anwendungen genutzt werden, werden als Backup-Garantie für kritische Anwendungen eingesetzt. Infolgedessen müssen nicht-kritische Anwendungen in einem Ausfallszenario möglicherweise abgeschaltet werden, um kritische Funktionalität am Laufen zu halten. Durch die Anwendung von *graceful degradation* können daher die Anforderungen an die ein *fail-operational* Verhalten erfüllt werden, während der Hardwarebedarf gesenkt wird. Jedoch entsteht dabei das Risiko in einem Ausfallszenario nicht-kritische Funktionalität zu verlieren.

State-of-the-art Methoden, die zur Entwurfszeit laufen, sind nicht geeignet, um diese Funktionalität zu erreichen, da das System dynamisch auf Ausfälle und Ereignisse reagieren muss. Außerdem muss das System jederzeit an Kundenwünsche anpassbar sein, so dass das System flexibel auf Änderungen reagieren muss. Um die große Anzahl möglicher Konfigurationen zu bewältigen können agentenbasierte Strategien eingesetzt werden die zur Laufzeit eine Lösung finden. In einem agentenbasierten System ist die Steuerung des Systems dezentralisiert, was den zusätzlichen Vorteil hat, dass es keinen Single-Point-of-Failure gibt und somit die Steuerung selbst *fail-operational* wird. Im Bereich des dynamischen Mappings hat ein Agent die Aufgabe, das Mapping einer einzelnen Anwendung oder eines Tasks zur Laufzeit zu finden. Die größte Herausforderung beim Entwurf eines solchen Systems besteht jedoch darin, ein berechenbares Verhalten zu erreichen und Grenzen zu finden, innerhalb derer ein sicherer und dynamischer Betrieb möglich ist.

Zu diesem Zweck schlagen wir in dieser Arbeit einen agentenbasierten *fail-operational* Ansatz vor, der *graceful degradation* nutzt. Dabei befassen wir uns mit Herausforderungen bei der Timing-Analyse, dem Checkpointing und der Zuverlässigkeitsanalyse, um ein berechenbares Verhalten unseres Ansatzes zu erreichen. Im Folgenden stellen werden unsere Beiträge dargestellt.

Zunächst stellen wir einen agentenbasierten Ansatz vor, der zur Laufzeit Anwendungen Ressourcen zuordnet. Dieser Ansatz gewährleistet das *fail-operational* Verhalten sicherheitskritischer Anwendungen durch die Nutzung von *graceful degradation* rekonfiguriert das System nach ECU-Ausfällen neu indem notwendige Redundanz wiederhergestellt wird. Unsere Ergebnisse zeigen, dass die Anzahl der tolerierten ECU-Ausfälle bis zum Ausfall einer sicherheitskritischen Anwendung erheblich erhöht werden kann, ohne dass redundante Hardware-Ressourcen hinzugefügt werden müssen. Anschließend erweitern wir den Ansatz und präsentieren eine Performanceanalyse, die in der Lage ist, zeitliche Einschränkung von verteilten *fail-operational* Anwendungen in unserem System zu analysieren. Unsere Methode stellt sicher, dass selbst während eines kritischen ECU-Ausfalls immer eine Backup-Lösung zur Verfügung steht, die die Ende-zu-Ende-Zeitvorgaben einer Anwendung einhält. Darüber hinaus stellen wir ein dynamisches, dezentrales Mapping-Verfahren vor, das mit Hilfe unseres analytischen Ansatzes zur Laufzeit Lösungen innerhalb aller aufgestellten Bedingungen findet. In unserem Versuchsaufbau kann unser *graceful degradation* Ansatz im Vergleich zu einem aktiven Re-

dundanzansatz etwa doppelt so viele kritische Anwendungen auf der gleichen Architektur unterbringen.

Während dieser Ansatz ein Verhalten innerhalb der zeitlichen Grenzen vor und nach einem Ausfall gewährleistet, könnte eine Anwendung während eines *failovers* für eine bestimmt Zeit lang keinen output erzeugen. Hier muss sichergestellt werden, dass ein *failover* innerhalb der FTTI garantiert werden kann. Zu diesem Zweck analysieren wir die Auswirkungen eines *failovers* auf das zeitliche Verhalten von verteilten, ausfallsicheren Anwendungen und leiten eine obere Schranke für die *worst-case-failover-Zeit* her. Anstatt zeitaufwändige Experimente durchzuführen, kann unsere formale Analyse verwendet werden, um zu bewerten, ob ein Mapping die *failover*-Zeitbeschränkungen erfüllt oder nicht, so dass sogar eine Bewertung zur Laufzeit möglich ist. In unseren Experimenten stellen wir das *fail-operational* Verhalten eines verteilten, neuronalen Netzes sicher und untersuchen dieses um die *worst-case-failover-Zeit* zu ermitteln. Unsere zufällig generierten Ergebnisse liegen bis zu $6,0\%$ unter unserer analytisch abgeleiteten Grenze.

Eine weitere große Herausforderung besteht darin, dass die meisten Anwendungen einen Status haben, der bei einem Ausfall verloren gehen kann, so dass eine Wiederherstellung dieses Zustands unmöglich wird. In einem System mit passiver Redundanz kann daher ein Ansatz mit periodischem *checkpointing* und *rollback recovery* verwendet werden, um den Zustand an einen anderen ECU zu senden, wo die Anwendung nach einem Ausfall neu gestartet werden kann. Hier besteht die Herausforderung darin, eine geeignete *checkpointing*-Periode zu finden, so dass der durch das Senden des *checkpoints* verursachte Netzwerk- und Verarbeitungsaufwand minimiert wird. Gleichzeitig muss aber sichergestellt werden, dass anwendungsspezifische Anforderungen weiterhin erfüllt werden. Wir stellen einen Ansatz vor, um die größtmögliche *checkpointing*-Periode analytisch herzuleiten, indem wir eine Obergrenze für die Anzahl der verpassten Berechnungsschritte aufgrund von Fehlern analysieren. Da die Auswirkungen des Alters der Zustandsdaten anwendungsspezifisch sind, verwenden wir einen SLAM-Algorithmus, eine Anwendung, die häufig in autonomen Systemen wie Robotern oder selbstfahrenden Autos verwendet wird, als realistisches Beispiel, um die Auswirkungen auf die Anwendung zu bestimmen. Die Ergebnisse unserer Fallstudie stimmen mit unserer analytisch abgeleiteten Schranke überein. Insgesamt kann mit unserem Ansatz eine größtmögliche *checkpointing*-Periode bestimmt werden, um den Netzwerk- und Verarbeitungs-Overhead zu minimieren und so ein kosteneffizientes und sicheres Verhalten von autonomen Systemen zu erreichen.

Abschließend untersuchen wir die Auswirkungen unseres *graceful degradation* Ansatzes auf die Zuverlässigkeit von kritischen und nicht-kritischen Anwendungen. Da Ressourcen dynamisch zur Laufzeit verschoben werden, können nicht-kritische Anwendungen, die selber nicht direkt von einem Ausfall betroffen sind, trotzdem abgeschaltet werden, um Ressourcen für den Neustart kritischer Anwendungen freizugeben. Mit unserem *graceful degradation* Ansatz wird also die Zuverlässigkeit kritischer Anwendungen erhöht auf Kosten einer niedrigeren Zuverlässigkeit nicht-kritischer Anwendungen . Um ein solches System effizient zu gestalten, ist es wichtig, die Auswirkungen unseres *graceful degradation* Ansatzes auf kritische und nicht-kritische Anwendungen zu quantifizieren und zu

verstehen, um die Berechenbarkeit unseres Systems zu erhöhen. Deshalb präsentieren wir eine tiefgehende trade-off-Analyse unseres *graceful degradation* Ansatzes, in der wir den Ressourcenverbrauch und die Auswirkungen von *graceful degradation* auf die Zuverlässigkeit von kritischen und nicht-kritischen Anwendungen analysieren.

Insgesamt ermöglicht unser agentenbasierter *fail-operational* Ansatz ein dynamisches und sicheres Verhalten von Automobilanwendungen. Unsere analytischen Ansätze helfen Entwicklern, kosteneffizientere und berechenbarere Systeme zu erstellen. Mithilfe unserer Methoden können Entwickler die Auswirkungen von *graceful degradation* effizient analysieren und einen trade-off entsprechend ihrer Systemanforderungen finden. Unsere Ergebnisse bestätigen, dass der Einsatz von *graceful degradation* die Kosten im Vergleich zu konventionellen Redundanzansätzen enorm senken kann, ohne die Zuverlässigkeit kritischer Anwendungen zu beeinträchtigen, wenn eine Verringerung der Zuverlässigkeit unkritischer Anwendungen akzeptabel ist.

# Contents

# Acronyms

| | |
|---|---|
| 2oo2DFS | Two-out-of-Two Diagnosis Fail-Safe. |
| API | Application Programming Interface. |
| ASIL | Automotive Safety Integrity Level. |
| ATD | Asynchronous Time Division. |
| BDD | Binary Decision Diagram. |
| CAN | Controller Area Network. |
| CDF | Cumulative Distribution Function. |
| CPU | Central Processing Unit. |
| DCC | Duplex Control Computer. |
| E/E | Electrical/Electronic. |
| ECC | Error Correcting Code. |
| ECU | Electronic Control Unit. |
| EKF | Extended Kalman Filter. |
| FTTI | Fault Tolerant Time Interval. |
| ISO | International Organization for Standardization. |
| LIN | Local Interconnect Network. |
| MCU | Microcontroller Unit. |
| MPSoC | Multiprocessor-System-on-a-Chip. |
| MTTF | Mean Time to Failure. |
| MTU | Maximum Transmission Unit. |
| NoC | Network-on-Chip. |
| OTA | Over-the-Air. |
| PFC | Primary Flight Computer. |

| | |
|---|---|
| RAM | Random-Access Memory. |
| RR | Round-Robin. |
| SLAM | Simultaneous Mapping and Localization. |
| SoC | System-on-a-Chip. |
| SOME/IP | Scalable service-Oriented MiddlewarE over IP. |
| TCP | Transmission Control Protocol. |
| TDM | Time-Division Multiplexing. |
| TDMA | Time-Division Multiple Access. |
| TMR | Triple Modular Redundancy. |
| WCET | Worst-Case Execution Time. |

# Nomenclature

| | |
|---|---|
| $A_N$ | Set of non-critical applications. |
| $A_S$ | Set of safety-critical applications. |
| $A_{N,f}$ | Set of of operational non-critical applications after the $f$-th failover. |
| $A_{S,f}$ | Set of of operational critical applications after the $f$-th failover. |
| $A$ | Set of applications. |
| $BW$ | Bandwidth budget of a link. |
| $B$ | The search space of a passive task instance. |
| $CL_{inter}$ | Worst-case communication interference latency. |
| $CL_{trans}$ | Worst-case transmission latency. |
| $CL$ | Worst-case communication latency. |
| $C$ | CPU budget of an ECU. |
| $E$ | Set of Electronic Control Units. |
| $F$ | Set of failed ECUs. |
| $G_a$ | Application instance graph. |
| $L$ | Set of Ethernet links. |
| $MTTF_{AVG,active,nc}$ | Average MTTF of non-critical applications of the active redundancy approach. |
| $MTTF_{AVG,deg,nc}$ | Average MTTF of non-critical applications of our graceful degradation approach. |
| $MTTF_{AVG}$ | The averaged Mean Time to Failure. |
| $MTTF_{reduction,nc}$ | Percental MTTF reduction of non-critical applications of our degradation approach compared to the active redundancy approach. |
| $MTTF$ | Mean Time to Failure. |
| $M_a$ | Set of active message instances. |
| $M_b$ | Set of backup message instances. |
| $N_a$ | Total amount of applications. |
| $N_c$ | Total amount of critical applications. |
| $N_f$ | Number of failed ECUs. |
| $N_t$ | Number of failed application executions. |
| $N_{SI,ar}$ | Total number of service intervals on an ECU that are both allocated and reserved. |
| $N_{SI,a}$ | Total number of service intervals on an ECU that are exclusively allocated. |

| | |
|---|---|
| $N_{SI,r}$ | Total number of service intervals on an ECU that are exclusively reserved. |
| $N_{SL,a}$ | Total number of slots on a link that are exclusively allocated. |
| $N_{SL,r}$ | Total number of slots on a link that are exclusively reserved. |
| $N_{e,c}$ | Number of exploration of a critical applications. |
| $N_{l,wc}$ | Worst-case number of lost application executions. |
| $N_{m,wc}$ | Worst-case number of missed application executions. |
| $N_{nc}$ | Total amount of non-critical applications. |
| $N_{t,wc}$ | Worst-case number of failed application executions. |
| $PL$ | Path latency. |
| $P_S$ | Share of critical applications in the total number of applications. |
| $P_a$ | Application period. |
| $P_c$ | Checkpointing period. |
| $P_{c,d}$ | Default checkpointing period. |
| $P_{c,max}$ | Optimal checkpointing period. |
| $P$ | Probability of a proper working system. |
| $QoS_N$ | Percentage of operational non-critical applications after a failure. |
| $QoS_S$ | Percentage of operational safety-critical applications after a failure. |
| $R_{savings}$ | Percental resource savings of our degradation approach over the active redundancy approach. |
| $R$ | Reliability. |
| $SI_a$ | Set of service intervals allocated for active task instances. |
| $SI_r$ | Set of service intervals reserved for passive task instances. |
| $SI_{max}$ | Maximum number of available service intervals. |
| $SI$ | Service interval. |
| $SL_a$ | Set of slots allocated for active message instances. |
| $SL_r$ | Set of slots reserved for backup messages. |
| $SL_{max}$ | Maximum number of available slots. |
| $SL$ | Scheduling slot. |
| $S_O$ | Total amount of occupied slots in a system. |
| $S_e$ | Slot capacity of an ECU. |
| $S_t$ | Total slot capacity of the system. |
| $S_{O,active}$ | Total number of consumed slots of the active redundancy approach. |
| $S_{O,deg}$ | Total number of consumed slots of our degradation approach. |

| | |
|---|---|
| $S_{O,no}$ | Total number of consumed slots when using no redundancy. |
| $S_{OH,active}$ | Slot overhead introduced by the active redundancy approach. |
| $S_{OH,deg}$ | Slot overhead introduced by our degradation approach. |
| $TL_{exec}$ | Worst-case task execution latency without interference. |
| $TL_{inter}$ | Worst-case task interference latency. |
| $TL$ | Worst-case task latency. |
| $T_a$ | Set of active task instances. |
| $T_c$ | Set of critical task instances. |
| $T_p$ | Set of passive task instances. |
| $T^r_{t_a}$ | Set of tasks of critical applications where a passive task instance has reserved a slot that is allocated by the task instance $t_a$. |
| $T$ | Set of all task instances. |
| $W$ | Worst-case execution time. |
| $\Delta O_{tol}$ | Maximum tolerated output delay. |
| $\Delta O$ | Time difference between two application outputs. |
| $\alpha$ | Binding function of active task instances. |
| $\beta$ | Binding function of passive task instances. |
| $\delta$ | Application deadline. |
| $\lambda$ | Failure rate. |
| $\mathcal{E}$ | Application graph edges. |
| $\mathcal{F}_{wc}$ | Worst-case application failover time. |
| $\mathcal{F}$ | Application failover time. |
| $\mathcal{L}_i$ | End-to-end application latency of a single execution run $i$. |
| $\mathcal{L}_{bc}$ | Best-case end-to-end application latency. |
| $\mathcal{L}_{wc}$ | Worst-case end-to-end application latency. |
| $\mathcal{V}$ | Application graph vertices. |
| $\rho$ | Routing function of active message instances. |
| $\sigma$ | Routing function of backup message instances. |
| $\tau_d$ | Worst-case failure detection time. |
| $\tau_r$ | Worst-case task recovery time. |
| $\tau_{SI}$ | Service interval time. |
| $\tau_{SL}$ | Slot interval time. |
| $\tau_{off}$ | Worst-case offering time. |
| $\tau_{step}$ | Time step size in the simulation. |
| $\tau_{sub}$ | Worst-case subscription time. |
| $\tau_{trans}$ | Transmission time over one link. |
| $\tau$ | A specified amount of time. |
| $\varphi$ | Structure function. |

| | |
|---|---|
| $a$ | Application. |
| $bw_{alloc}$ | Amount of reserved bandwidth. |
| $bw$ | Bandwidth requirement of a message. |
| $c_{alloc}$ | Amount of allocated CPU resources. |
| $c$ | CPU consumption of a task. |
| $d_{max}$ | Maximum tolerable data age. |
| $d_{wc}$ | Worst-case data age. |
| $d$ | Data age. |
| $e$ | Electronic Control Unit. |
| $f$ | Failed ECU. |
| $g$ | A boolean function. |
| $l$ | Ethernet link. |
| $m$ | A message instance. |
| $p_f$ | Probability function. |
| $p$ | A preceding task instance in the application graph. |
| $s$ | Switch. |
| $t$ | A task instance. |
| $u$ | Function which translates a boolean variable z which is dependent on the state of a task instance to a boolean variable y which is dependent on the state of an ECU. |
| $x$ | A boolean variable. |
| $y$ | Function which indicates the operational status of an ECU. |
| $z$ | Function which indicates the operational status of a task instance. |
| $\Delta\bar{e}$ | Average difference between the error before and after a failure. |
| $\Delta e_{wc}$ | Difference between the worst-case error after a failure compared to the average error before the failure. |

# 1 Introduction

## Contents

## 1.1 Motivation

With the advent of autonomous driving, safety requirements for automotive systems are strongly increasing. In the past, a fail-safe behavior was often sufficient where automotive functions were brought into a safe state or shut down completely as a driver was always available as a backup solution to steer the car. Without any driver, automotive systems must be able to mitigate any hardware or software failures and ensure a fail-operational behavior. At the same time, automotive manufacturers are required to offer over-the-air software updates so that customers get the latest functionality and can customize their cars. Automotive companies have to integrate more and more applications into their E/E architecture while each system will have a unique and customized configuration. These drivers lead to major changes in automotive hardware and software architectures.

In the past, automotive vendors added a new ECU for each new functionality in the vehicle with software that was tightly coupled to the hardware. Today, cars often consist of more than 100 ECUs to control functions in domains such as infotainment, chassis, powertrain, or comfort [1]. Now the automotive industry is moving towards zonal or more centralized architectures to cope with software development's complexity and reduce wiring and hardware components.

While, initially, the development of the centralization of hardware might appear contradictory to a decentralized software approach, the changes to the software architecture enable new decentralized fail-operational strategies. In the past, software applications were hardwired to a unique ECU in the system. Now, the centralization and homogenization of ECUs with a common software platform allows software execution on different ECUs. Our system model assumes that the architecture consists of a few powerful ECUs. New software platforms and virtualization techniques decouple hardware and software to allow a separate development. Furthermore, they enable a more dynamic behavior and the installment of software updates and new applications over-the-air with the help of service-oriented architectures and middleware solutions. Although this requires changes on both software and hardware levels, it also presents new opportunities for optimization.

The trend to centralized E/E architectures and more modular software design leads to more cross-domain systems with applications from different domains and different real-time and safety requirements running on the same hardware [1]. Here, composability is required to isolate software from each other to meet timing and safety-related constraints. Instead of finding a new deployment for all applications after each software update or redesigning the system manually, an automated isolated examination of each application is required to enable customized solutions.

Furthermore, a software platform has to enable a safe and dynamic fail-operational behavior by design. A system-wide, reproducible fail-operational approach could reduce the effort spent on validation and integration. One of the main challenges here is that a fail-operational behavior requires redundancy. With the already increasing computational demand by software, replicating the expensive hardware multiple times would increase hardware costs significantly. In other industries with fail-operational requirements such as aviation, static redundancy with triple-triple redundancy is used [2]. However, the automotive industry is much more cost-sensitive and requires frequent software updates with unique configurations. Therefore, state-of-the-art static redundancy approaches can not fulfill the requirements for a cost-sensitive and adaptable solution. Here, new approaches are required to achieve a dynamic, safe, and cost-efficient behavior.

A modular software design allows the dynamic shifting of software components at run-time, including the activation and deactivation of components on different ECUs. Combined with the possibility of running safety-critical and non-critical applications together on the same hardware platform, this perspective opens new strategies to implement fail-operational systems. Here, a system-wide software platform can manage redundancy and fail-operational behavior at run-time. Hardware costs can be reduced by applying graceful degradation and prioritizing safety-critical applications in a failure scenario. For example, if a critical application for autonomous driving is affected directly by an ECU failure, it could be restarted on another ECU while shutting down non-critical functionality from the infotainment domain to free resources. This approach has the advantage that resources already used in the car are repurposed instead of withholding resources specifically for this scenario that would be otherwise unused.

State-of-the-art design-time methods are not applicable to achieve this functionality as the system needs to react and recover dynamically on failures and events. Furthermore, software functionality might change on demand, so the system has to react flexibly to changes. To cope with the high amount of such possible configurations, we need to make the system self-aware, which can be enabled by applying agent-based strategies. In an agent-based system, the system's control is decentralized, which has the advantage that there is no single point of failure and thus makes the control itself fail-operational when adequately designed. In the field of dynamic mapping, an agent has the responsibility to find the mapping of a single application or task at run-time. However, the main challenge when designing such a system is to achieve predictable behavior and to find bounds within which a safe and dynamic operation is possible.

This thesis aims to investigate the feasibility of implementing graceful degradation in automotive systems with decentralized and dynamic control, intending to achieve significant resource savings and higher flexibility to react on failures compared to static redundancy approaches while maintaining the same reliability for critical applications. We present an approach to dynamically achieve a graceful degrading system behavior using an agent-based approach and introduce methodologies to achieve a more predictable system design and to analyze non-functional properties with a focus on timing behavior, state recovery and reliability analysis.

In the following, we give an overview of recent and future development in E/E architectures and software platforms (Section 1.2). Afterward, we introduce fail-operational systems and the concept of graceful degradation (Section 1.3). Then, we discuss the research challenges that arise when applying graceful degradation to automotive systems (Section 1.4) and present our contributions to address these challenges (Section 1.5). Finally, we list our publications (Section 1.6) and the remaining outline of the thesis (Section 1.7).

## 1.2 Development in E/E Architectures and Software Platforms

The automotive industry sees itself confronted with increasing customer needs and requirements, forcing a change in E/E and software platforms. In the near future, software systems will be highly customizable and customers will demand the latest functionality by over-the-air software updates. To cope with complexity, automotive companies integrate applications on a few powerful ECUs. Fast and stable connections over mobile networks are required to achieve increased connectivity. In the distant future, mobile connections might even be used to run automotive services in the cloud. The interest in over-the-air software updates is also driven by automotive vendors which are experimenting with new business models, where functions are sold via subscription or where software updates can be purchased after buying a car.

With the advent of autonomous driving, automotive companies also see themselves confronted with highly increased computational demands by applications. More and more data produced by different types of sensors require high-speed in-car connections. Additionally, autonomous systems have to be fail-operational to fulfill increased safety requirements. This results in even higher hardware costs as redundancy is required to achieve a safe fail-operational behavior.

As a consequence and to cope with increasing costs and complexity, applications are being integrated into more powerful multicore control units (Subsection 1.2.1). This leads to a consolidation of existing ECUs and a more centralized E/E architecture [1]. New software platforms decouple hardware and software, allowing a more dynamic system behavior to enable software updates (Subsection 1.2.2).

These changes lead to many new challenges for software's safe and dynamic integration in a mixed-critical system (Subsection 1.2.3). As automotive companies need to reduce costs, fail-operational architectures must be as resource-efficient as possible. Here, the changes on E/E and software platforms also open new strategies to achieve a fail-operational behavior on the system level (Subsection 1.2.4).

### 1.2.1 E/E Architectures

In the past, automotive vendors often added a new ECU for each new functionality in the vehicle. The BMW 7 series in 2007 contained around 270 functions, which were deployed over 67 embedded platforms [3]. This leads to a heterogeneous mixture of many different ECUs and bus systems [4]. Now, cars often consist of more than 100 ECUs to control functions in the domains such as infotainment, chassis, powertrain or comfort [1]. Figure 1.1 depicts the current and future development of automotive E/E architectures. Today, domain-specific functions are integrated into domain controllers (Figure 1.2). The domain controllers are connected via high-bandwidth Ethernet over a central gateway, which allows increased routing complexity and throughput. Furthermore, the controllers are connected via network systems such as Controller Area Network (CAN), Local Interconnect Network (LIN), FlexRay, and Ethernet to specialized ECUs, controllers and actuators. The central gateway is the only connection for external communication and enables secure over-the-air software updates.

This trend of centralization and integration is believed to continue. The automotive industry is aiming towards zonal or more centralized architectures for the future. Some vendors such as Tesla prefer a centralized architecture, where most functions are executed on a single ECU such as the Full Self-Driving Computer of Tesla [6]. Bosch is developing a vehicle-centralized, zone-oriented E/E architecture with a few centralized, powerful vehicle computers integrating cross-domain functionality similar as shown in Figure 1.3 [5, 7]. These vehicle computers are connected to actuators and sensors via zone ECUs. This reduces the required wiring and weight in vehicles but also system complexity.

Centralized architectures as propositioned by Bosch have been presented in research [8]. The RACE research project developed an architecture with a central computer

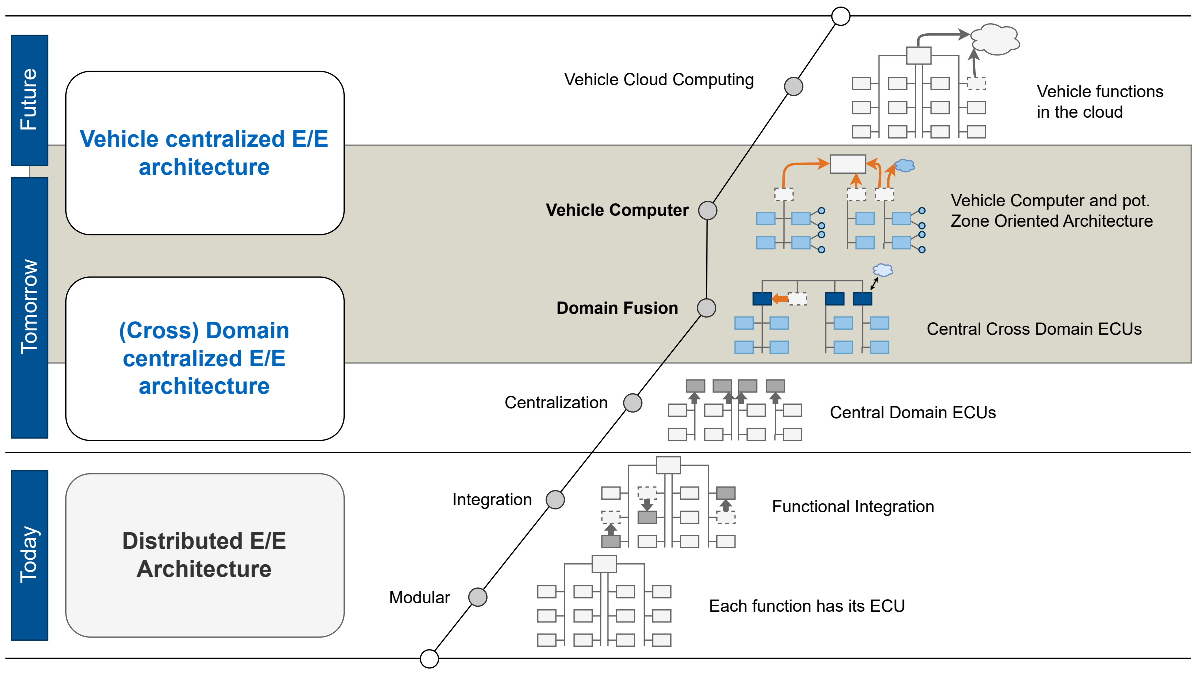

Future
Tomorrow
Today
Vehicle centralized E/E architecture
(Cross) Domain centralized E/E architecture
Distributed E/E Architecture
Vehicle Cloud Computing
Vehicle Computer
Domain Fusion
Centralization
Integration
Modular
Vehicle functions in the cloud
Vehicle Computer and pot. Zone Oriented Architecture
Central Cross Domain ECUs
Central Domain ECUs
Functional Integration
Each function has its ECU

**Figure 1.1:** Roadmap for the development of E/E architectures. [5]

platform [9]. The platform comprises five Duplex Control Computers (DCCs) connected via an Ethernet network ring. Each DCC has two ECUs which can be used for executing applications redundantly or in parallel.

Figure 1.1 presents the development of automotive E/E architectures and shows the trend moving from distributed systems consisting of ECUs with single functionalities to more centralized architectures. In the future, more and more in-vehicle functionality is expected to be offloaded to the cloud. We presented a hybrid cloud architecture using onboard and cloud resources in [10]. The advantages of the offloading process could be reduced onboard energy consumption and more computational power. Yet there are still many open challenges regarding strict real-time requirements and network availability that have to be met by future mobile networks.

### 1.2.2 Software Platform

The continuous integration on fewer controllers also requires new software platforms that support new functionalities such as Over-the-Air (OTA) updates. In the past, functionality was tightly coupled with the hardware. Once the vehicle was built, changing software was difficult, and introducing new features impossible.

Today abstraction layers and virtualization techniques allow a decoupling of software and hardware at least on domain and central controllers [11]. The AUTOSAR Adaptive Platform, a standard based on POSIX operating systems, allows a more flexible E/E architecture [12, 13]. Adaptive ECUs allow an extension of applications and adds new software functionality. Furthermore, this allows us to develop applications independently

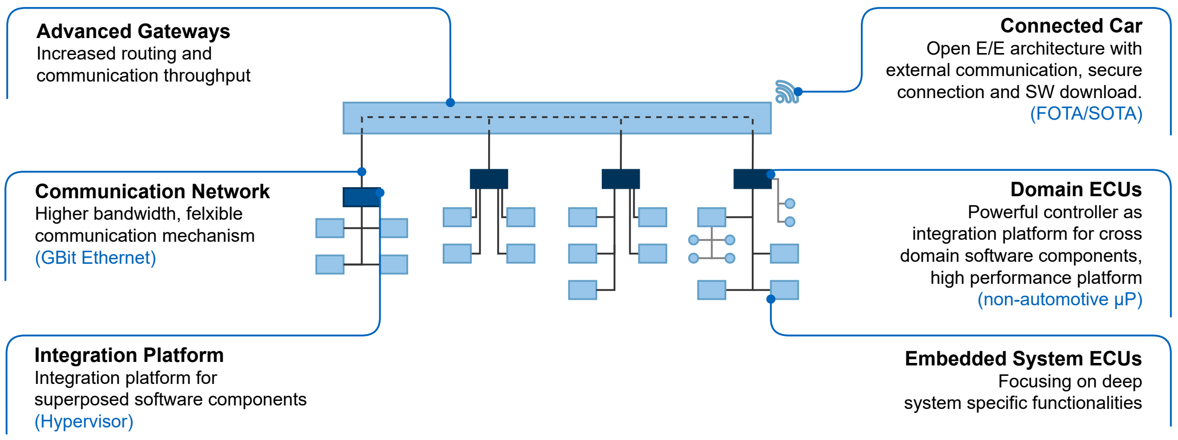

Advanced Gateways
Increased routing and communication throughput
Connected Car
Open E/E architecture with external communication, secure connection and SW download.
(FOTA/SOTA)
Communication Network
Higher bandwidth, felxible communication mechanism
(GBit Ethernet)
Domain ECUs
Powerful controller as integration platform for cross domain software components, high performance platform
(non-automotive µP)
Integration Platform
Integration platform for superposed software components
(Hypervisor)
Embedded System ECUs
Focusing on deep system specific functionalities

**Figure 1.2:** Domain-centralized E/E architecture consisting of a few domain ECUs which are interconnected via GBit Ethernet to a central gateway. [5]

from each other and without determining on which hardware the software will run [14]. Next to hardware abstraction a software platform typically handles communication, persistent storage of data, and encryption amongst others, and offers various services to user applications (Figure 1.4).

To allow software updates, automotive companies are also moving toward service-oriented architectures. Scalable service-Oriented MiddlewarE over IP (SOME/IP) is a middleware solution described in multiple AUTOSAR standards [15]. This middleware includes a decentralized service discovery to dynamically find services in the system and a publish/subscribe scheme to publish and subscribe to events at run-time. Furthemore, it takes care of serialization and allows remote procedure calls and messaging.

## 1.2.3 Consequences of E/E and Software Changes

The trend to centralized E/E architectures and more modular software design leads to more heterogeneous systems [1]. Applications from different domains, with different real-time requirements and safety criticalities, will run on the same ECUs [16]. Here, composability is required to enable regular software updates and reduce the integration complexity. Instead of redesigning the entire software system and the application mappings on the hardware platform after each update, the validation of the system has to be done on board. Therefore, finding a mapping and ensuring requirements are isolated from other applications would be beneficial to allow a simple and fast integration. As it is crucial to fulfill timing and safety requirements, applications have to be isolated from each other to prevent interference and to enable a separate analysis [17]. A software platform has to enable portability and composability by design to allow safe and dynamic integration of safety-critical and non-critical applications [1]. Temporal isolation allows an independent timing analysis such that timing constraints can be verified in-

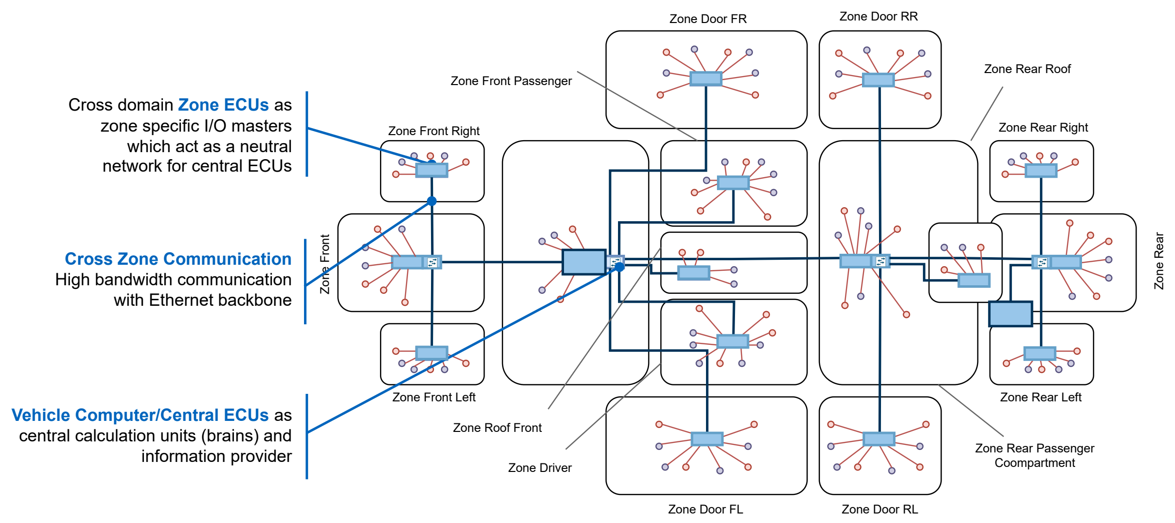

Cross domain Zone ECUs as zone specific I/O masters which act as a neutral network for central ECUs
Cross Zone Communication High bandwidth communication with Ethernet backbone
Vehicle Computer/Central ECUs as central calculation units (brains) and information provider
Zone Door FR
Zone Door RR
Zone Front Passenger
Zone Rear Roof
Zone Front Right
Zone Rear Right
Zone Front
Zone Rear
Zone Front Left
Zone Rear Left
Zone Roof Front
Zone Driver
Zone Rear Passenger Coompartment
Zone Door FL
Zone Door RL

**Figure 1.3:** Vehicle centralized E/E architecture consisting of central vehicle computers and cross domain zone ECUs. [5]

dependent from other application execution times. Overall, isolation and composability reduce design complexity, allowing faster integration and avoiding critical failures.

The trend towards autonomous driving increases safety requirements enormously. As the system has to cope with software and hardware failures independently without human interaction, it must be designed fail-operationally. Subsection 1.3 will further detail the development from fail-safe to fail-operational approaches. As redundancy is required to enable a fail-operational behavior, hardware costs will increase tremendously. Finding a resource-efficient solution to reduce costs could be a pivotal factor in remaining competitive for automotive vendors. Furthermore, validation of safety requirements requires a lot of expertise and effort. A system-wide, reproducible fail-operational approach could reduce the effort spent on validation and integration.

### 1.2.4 New Possibilites for Fail-Operational Strategies

While the trend to a more centralized E/E architecture might appear contradictory to a decentralized software approach at first glance, future software platforms will decouple hardware and applications further and enable a more dynamic behavior. An entirely centralized solution always bears the risk of a single point of failure. A modular software design will allow dynamic shifting of software components at run-time, including activation and deactivation of components on different ECUs. This perspective opens new strategies to enable fail-operational behavior. We believe that future software platforms have to offer fail-operational capabilities and be able to automatically distribute safety-critical applications on the system with appropriate redundancy and failure detection measures. Furthermore, this platform must be able to cope with frequent software updates, and feature upgrades, and dynamically react to events such as software or hardware failures. Here, isolation techniques and composability are essential to reduce

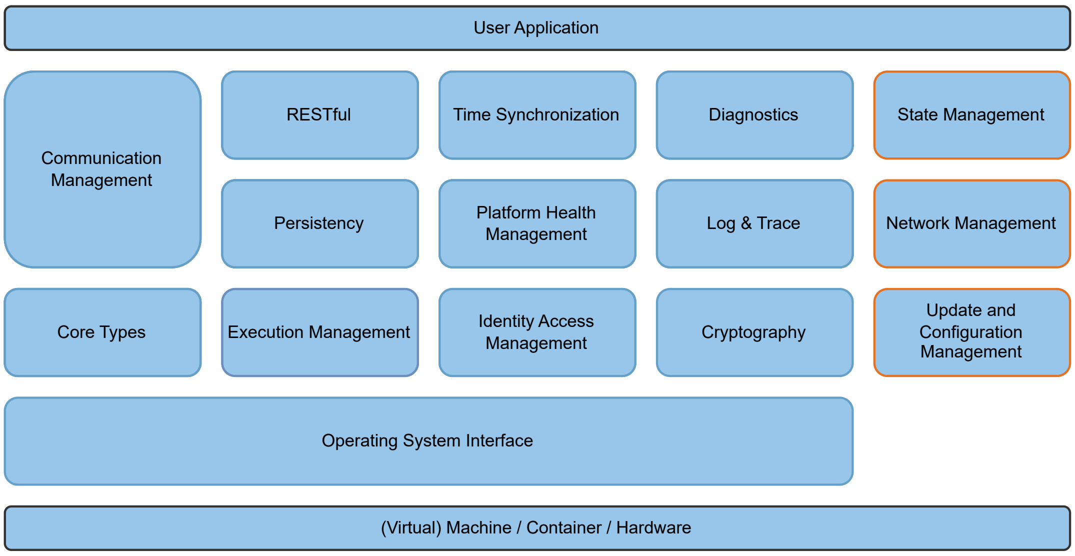


**Figure 1.4:** Depiction of the AUTOSAR Adaptive Platform and its functional clusters. [12]

design complexity, allow faster integration times, and avoid critical failures. Additionally, a system-wide, reproducible fail-operational approach could reduce the effort spent on validation and integration.

## 1.3 Fail-Operational Systems

Automotive systems always had high safety requirements. Most safety-critical applications in cars today show a fail-safe behavior. In a fail-safe solution, the system would detect a failure and switch it to a safe state where the functionality is often shut down to avoid interference with other applications [18]. This behavior is sufficient for support functions such as steering assistance, where a driver can maintain control. By contrast, a system is fail-operational if it can continue operation without changing its performance or objectives even if a single failure occurs [19]. As more and more functionality is being automated, the demand for fail-operational requirements increases as failures are more likely to cause hazardous situations [20] (Figure 1.5). To achieve a fail-operational behavior, redundancy is required. In the following, we present well-known redundancy concepts to achieve a fail-operational behavior on ECU and Microcontroller Unit (MCU) level (Subsection 1.3.1).

As these methods significantly increase hardware costs, finding more resource-efficient solutions is essential. Furthermore, validation of safety requirements requires a lot of expertise and effort. A system-wide, reproducible fail-operational approach could reduce the effort spent on validation and integration. While these concepts are often implemented directly in hardware they could also be entirely controlled and implemented by software. In the work at hand, we present an agent-based approach, which dynamically applies redundancy on a software level using graceful degradation. However, a software-based fail-operational approach always requires hardware support to detect failures,

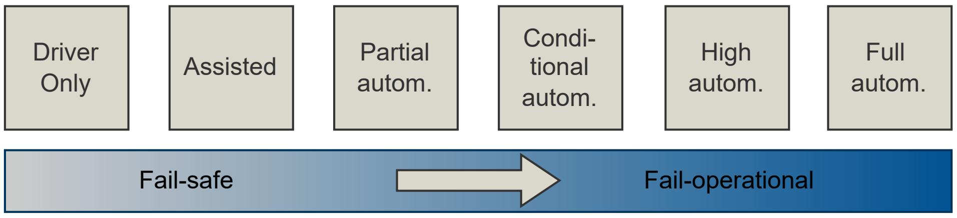


**Figure 1.5:** Development of fail-operational requirements with an increasing level of automation.

monitor the system, or use additional redundancy on the hardware level. Next to redundancy concepts, hardware systems provide essential functionality to detect, monitor, and cope with failures or faults. Typically, this includes a segregation of applications by memory and access permission or a control of storage elements protected with Error Correcting Code (ECC) [21]. Semiconductor companies such as Infineon develop microcontrollers such as the AURIX product family offering several safety and security features specifically addressed to automotive safety requirements [22].

### 1.3.1 Fail-Operational Concepts

In the following, we present the two widely known fail-operational concepts Triple Modular Redundancy (TMR) and Two-out-of-Two Diagnosis Fail-Safe system. These are general concepts to achieve a fail-operational behavior which could be implemented in hardware, software, or in a composition of both.

#### 1.3.1.1 Triple Modular Redundancy

TMR is a well-known redundancy technique to achieve a fail-operational behavior (Figure 1.6) [23]. The inputs are forwarded to three redundant computational units. A majority voter compares and validates the results. In case one result differs, the other results overrule the faulty result. This concept can be implemented in software or hardware and the units can be implemented using diverse redundancy to reduce common-cause failures. Separated power supplies eliminate the risk of shutting down all units in case of a power outage.

As an example, TMR has been applied to the fly-by-wire system of the Boeing 777 consisting of the computing system, power systems, and all communication paths (Figure 1.7) [2]. The triple redundant Primary Flight Computers (PFCs) each contains triple redundant components such as power supplies, microprocessors, and interfaces. As the costs of a plane are four to five orders of magnitude higher than the costs of a car, the additional hardware costs are only a minor factor of the total costs. The automotive industry differs as it is much more cost-sensitive regarding hardware components. Therefore, new solutions are required to reduce the need for additional hardware components.

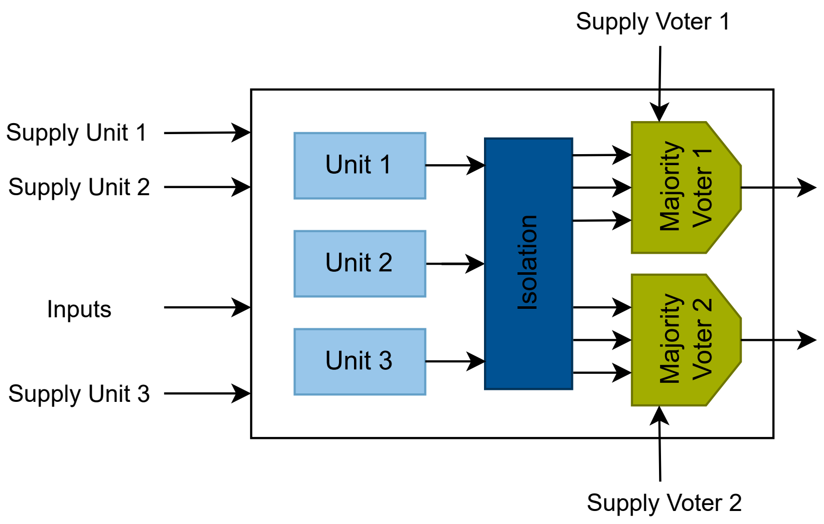


**Figure 1.6:** Architecture of a TMR system, with three identical modules providing redundant computation and voting logic for increased reliability. [21]

#### 1.3.1.2 Two-out-of-Two Diagnosis Fail-Safe System

A 2oo2DFS system consists of two subsystems which each have error detection capabilities and which can switch into a fail-safe mode (Figure 1.8) [21]. The systems monitor each other so that only correct output is being forwarded in a failure scenario. The two subsystems usually have independent power supply and communication interfaces. In a symmetric setup, the subsystems are identical, so the full functionality is preserved in a failure scenario. In an asymmetric setup, the second controller could be less powerful and only execute safety-critical functionality or a degraded version of safety-critical functions. For example, a fail-safe behavior within each subsystem can be achieved with a lock-step architecture within an MCU.

### 1.3.2 Graceful Degradation

Graceful Degradation is the ability of a system to maintain functionality when parts of the system malfunction [24]. In a failure scenario, a minimum of functionality is guaranteed to be operational. This implies that some functionality is prioritized by design to maintain operation over other functionality. In literature functional degradation is sometimes referred to as switching to a function with only the most essential features. In our context, we distinguish between safety-critical and non-critical applications and examine graceful degradation on a system level instead of a functional level. A system with applications of different criticality levels running on the same hardware platform is known as a mixed-critical system in literature [16].

To guarantee a fail-operational behavior, the system's survivability must be ensured at a lower performability, and this can be achieved through graceful degradation [25, 26, 27]. Instead of only guaranteeing the execution of critical functionality, our approach presented in this thesis also allows the active shutdown of non-critical applications to

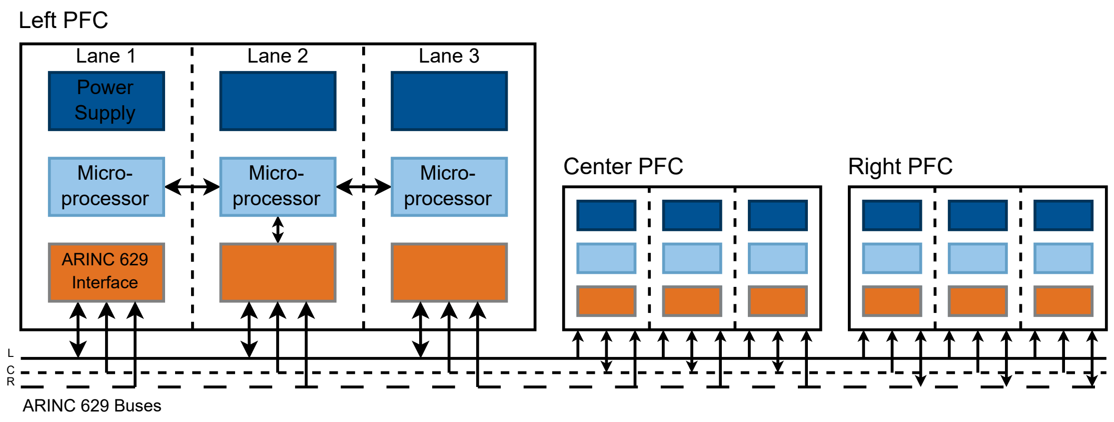


**Figure 1.7:** The three PFCs of the Boeing 777 with triple redundant computational channels.[2]

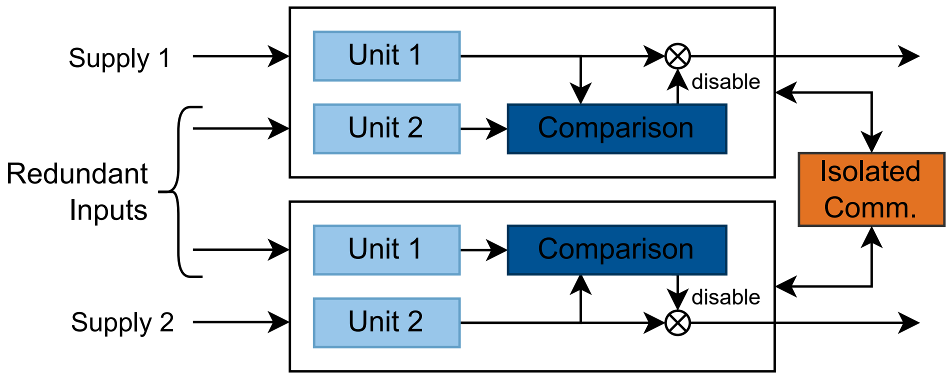


**Figure 1.8:** The architecture of a Two-out-of-Two Diagnosis Fail-Safe (2oo2DFS) system according to [21].

save critical applications if sufficient resources are unavailable. If a critical application for autonomous driving is affected directly by an ECU failure it could be restarted on another ECU while shutting down non-critical functionality from the infotainment domain to free resources. This approach has the advantage that resources already used in the car are repurposed instead of withholding resources that would be otherwise unused for this scenario.

Instead of using active redundancy, where a task replica is actively executed, or conventional passive redundancy, where a backup task instance is activated on failure, we propose to use graceful degradation. With conventional passive redundancy, the system has to be oversized to make sufficient resources available once a passive task is started. With graceful degradation, system resources are dynamically repurposed at run-time to ensure sufficient resources are available for critical applications. Resources allocated by non-critical applications are used as a backup guarantee for critical applications. As a result, non-critical functionality might have to be shut down in a failure scenario to keep critical functionality operational. Therefore, by applying graceful degradation, fail-operational requirements can be fulfilled while lowering the hardware demand at the cost of risking the loss of non-critical functionality in a failure scenario. However, to apply such an approach many research challenges have to be solved which we address in the next section.

## 1.4 Research Challenges

The trend toward a more homogeneous hardware and software architecture opens exciting possibilities for exploring new fail-operational strategies. In this context, the main research goal of this thesis is to explore a more cost-efficient and dynamic fail-operational solution such that the system can react more flexibly compared to static redundancy approaches. As anticipated, we explore graceful degradation as a redundancy approach to achieve these goals. This raises many research challenges, which we list in the following:

- In a safety-critical automotive system, unpredictable system behavior can not be tolerated. Therefore, a gracefully degrading system has to be designed in a way such that the degradation outcome for any failure can be predicted and that sufficient resources are available for re-starting critical applications. Furthermore, timing constraints must be respected even after the system switches to a new configuration. Correspondingly, the gracefully degrading system has to be designed so that the timing behavior can be analyzed. The configurations have to be found and pre-validated before any failure occurs. There is not enough time to find a new configuration and distribute the tasks after a failure. Here, the question arises of how new configurations can be found at run-time using only in-car resources such that the system can be reconfigured after a failure or when the system is updated. The corresponding control of resources and the mechanism to find new mappings has to be fail-operational by design.

- After a failure, the system has to switch to a new configuration and passive task instances must be re-started first. Here, the question arises if a gracefully degrading system could perform the failover and degradation process quickly enough to reach a new system configuration within the FTTI. New methodologies are required to evaluate this failover behavior.

- Many tasks in the automotive domain have a state that would be lost in a failure scenario. If redundant tasks are not actively executed, a solution must ensure they receive this state on time so the system can operate again within the FTTI.

- Last, the question arises of how a graceful degradation approach can be properly evaluated and compared to existing approaches. It is to be expected that graceful degradation negatively impacts non-critical applications. Here, a solution is required to quantitatively evaluate the impact of graceful degradation on the reliability and resource consumption of critical and non-critical applications.

## 1.5 Contributions

To address these research challenges we make the following contributions, which are described in further detail in the following subsections, and reference our associated publications connected to this thesis:

- We contribute our fail-operational architecture in Chapter 2. Here, we present our graceful degradation approach based on our new resource allocation and reservation system. We introduce an agent-based backtracking approach to set up our gracefully degrading system and find suitable resource allocations and reservations at run-time. Utilizing the composable architecture of our fail-operational system, we present a performance analysis for gracefully degrading systems that can verify that even during a critical ECU failure, a backup solution is always available that adheres to end-to-end timing constraints. [28, 29]

- To guarantee that the system can switch to a pre-validated configuration during a failover within the FTTI, we present our worst-case failover timing analysis in Chapter 3. [30]

- To save the state of a task we present our approach to periodic checkpointing with rollback recovery in Chapter 4. Here, the state of a task is periodically sent to another ECU so that the task can be restarted after a failure. Our approach allows us to derive a maximum achievable checkpointing period analytically to achieve a cost-efficient and safe behavior of automotive systems. [31]

- We investigate the impact of graceful degradation on the reliability of critical and non-critical applications in Chapter 5. Here, we also evaluate our graceful degradation approach based on resource costs and compare it to an active redundancy approach. [32]

### 1.5.1 Fail-Operational Automotive Architecture

The contributions to our fail-operational automotive architecture can be divided into three topics graceful degradation, agent-based decentralized control, and predictable timing behavior. These are presented in more detail in the following three subsections.

#### 1.5.1.1 Graceful Degradation

In this thesis we present a graceful degradation approach based on a new resource allocation and reservation system [28, 29]. Here, individual resources can be allocated and reserved at the same time. A reservation indicates that a resource might be used by a critical application after a failover leading to the resource withdrawal from a non-critical application. This design of our graceful degradation approach allows a predictable outcome of the degradation process for any failure in the system. Furthermore, it enables an isolated timing analysis of applications and allows a quick failover process. For the allocation and reservation of resources, we present three new strategies [29]. The *Random* strategy randomly allocates and reserves resource slots without considering existing allocations and reservations. The *FreeFirst* strategies aim at allocating and reserving resource slots that are not allocated or reserved yet. By contrast, the *FreeLast* strategies aim at allocating and reserving resource slots first which are already allocated or reserved leading to increased degradation effects. Our graceful degradation approach is evaluated based on resource savings and the impact on reliability. Furthermore, we compare it to active redundancy to conclude its efficiency.

#### 1.5.1.2 Agent-Based Decentralized Control

To find application mappings at run-time we present an agent-based mapping approach and a decentralized control of resources [28, 29]. Here, resources are managed in a decentralized way by each component. This approach and the decentralized control of resources enable a dynamic and adaptable behavior and a fail-operational control by eliminating any single point of failure. Our agent-based mapping approach ensures that active and redundant passive task instances are created and resources are allocated and reserved accordingly to ensure a gracefully degrading behavior. Our mapping approach also takes non-functional properties such as a predictable timing behavior into account [29]. A backtracking approach explores different mapping solutions. In a failure scenario, the decentralized resource control ensures a quick failover to a new resource configuration. Our approaches ensure a fail-operational behavior and a predictable gracefully degrading outcome for any ECU failure. After an immediate and quick failover to passive task instances which ensures the continuous operation of all critical applications, the redundancy has to be re-established to ensure a fail-operational behavior. A reconfiguration can be performed during a safe halt as running tasks have to be migrated to new ECUs. We evaluate our approaches based on an in-house developed simulation environment.

#### 1.5.1.3 Predictable Timing Behavior

It is an essential aspect of a fail-operational automotive architecture to guarantee a predictable timing behavior. A composable architecture design has to be considered during the design of the gracefully degrading system to allow an isolated timing analysis of individual applications. In the field of predictable timing analysis, we present a performance analysis that can analyze timing constraints of fail-operational distributed applications using graceful degradation [29]. Our method can verify that even during a critical ECU failure, a backup solution is always available that adheres to end-to-end timing constraints. Embedded in our mapping approach timing constraints can be solved a run-time. In our experimental setup, we compare the successful chance of finding a valid mapping and resource impact with an active redundancy approach and analyze the number of required exploration steps.

### 1.5.2 Worst-Case Failover Timing Analysis

While our approach to predictable timing ensures a behavior within timing constraints before and after a failure, an application might produce no output for a certain period during a failover. Here, it also has to be ensured that a failover within the FTTI can be guaranteed. For this purpose, we analyze the impact of failover on the timing behavior of distributed fail-operational applications and derive an upper bound for the worst-case failover time [30]. Instead of performing time-consuming experiments, our formal analysis can be used to evaluate whether a mapping would meet the failover timing constraints or not such that an evaluation at run-time is possible. We support our formal analysis by conducting experiments on a hardware platform using a distributed fail-operational neural network.

### 1.5.3 Checkpointing Period Optimization

Another major challenge is that most applications have a state that might get lost during a failure such that a recovery might be impossible. Thus, in a system with passive redundancy, periodic checkpointing with rollback recovery can send the state to another ECU, where the application can be restarted after a failure. Here, the challenge is to find a suitable checkpointing period such that network and processing overhead caused by sending the checkpoint is minimized but to ensure that application-specific constraints are still met. We present an approach to analytically derive the maximum checkpointing period by giving an upper bound on the number of missed computational steps due to failure effects [31]. As the implications of the state data age are application-specific, we use a SLAM algorithm, an application commonly used in autonomous systems such as robots or self-driving cars, as a real-world example to determine the effects on the quality of the application. Overall, using our approach, a maximum achievable checkpointing period can be determined to reduce network overhead to achieve a cost-efficient and safe behavior of autonomous systems.

### 1.5.4 Reliability Analysis

Last, we investigate the impact of graceful degradation on the reliability of critical and non-critical applications [32]. As resources are shifted dynamically, non-critical applications not directly affected by an ECU failure might get shut down to free resources for restarting critical tasks. Thus, with a graceful degradation approach, the reliability of critical applications is increased at the cost of a decrease in reliability of non-critical applications. To design such a system efficiently, it is crucial to quantify and understand the effect of a graceful degradation approach on critical and non-critical applications to increase predictability. Therefore, we present our approach to formally analyze the impact of graceful degradation on the reliability of critical and non-critical applications. We then quantify the effect of graceful degradation in distributed automotive systems and compare the achieved cost reduction with conventional redundancy approaches.

In summary, our agent-based fail-operational approach enables dynamic and safe behavior in automotive applications. Our analytical approaches help developers to create more cost-efficient and predictable systems. By following our methodologies, developers can efficiently analyze the impact of graceful degradation and meet a trade-off according to their system requirements. Our findings confirm that using graceful degradation can tremendously reduce cost compared to conventional redundancy approaches with no negative impact on the redundancy of critical applications if a reliability reduction of non-critical applications is acceptable.

## 1.6 List of Publications

Overall, the research that has led to this thesis has resulted in the following relevant publications:

- [28] P. Weiss, A. Weichslgartner, F. Reimann, and S. Steinhorst. Fail-Operational Automotive Software Design Using Agent-Based Graceful Degradation. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1169–1174, 2020. doi:10.23919/DATE48585.2020.9116322

- [33] P. Weiss, S. Nagel, A. Weichslgartner, and S. Steinhorst. Adaptable Demonstrator Platform for the Simulation of Distributed Agent-Based Automotive Systems. In *2nd International Workshop on Autonomous Systems Design (ASD 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020. doi:10.4230/OASIcs.ASD.2020.3

- [29] P. Weiss and S. Steinhorst. Predictable Timing Behavior of Gracefully Degrading Automotive Systems. *Design Automation for Embedded Systems*, pages 1–36, 2023. doi:10.1007/s10617-023-09271-x

- [30] P. Weiss, S. Elsabbahy, A. Weichslgartner, and S. Steinhorst. Worst-Case Failover Timing Analysis of Distributed Fail-Operational Automotive Applications. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1294–1299, 2021. doi:10.23919/DATE51398.2021.9473950

- [31] P. Weiss, E. Daporta, A. Weichslgartner, and S. Steinhorst. Checkpointing Period Optimization of Distributed Fail-Operational Automotive Applications. In *2021 24th Euromicro Conference on Digital System Design (DSD)*, pages 389–395, 2021. doi:10.1109/DSD53832.2021.00066

- [32] P. Weiss, A. Younessi, and S. Steinhorst. Reliability Analysis of Gracefully Degrading Automotive Systems. *Preprint on arXiv*, 2023. doi:10.48550/arXiv.2305.07401

## 1.7 Outline

This thesis is organized as follows. We first introduce our fail-operational automotive architecture in Chapter 2. Here, we present our system model which we use throughout this thesis. Afterwards, we present our approach to fail-operational automotive software design using agent-based graceful degradation. As it is critical for a fail-operational architecture to guarantee a predictable timing behavior, we apply our concept of graceful degradation to a scheduling approach and present a formal analysis to derive worst-case timing estimations for gracefully degrading systems. We extend the agent-based mapping approach to include a backtracking approach and respect time constraints at run-time. For our experiments we conduct simulations on our in-house developed simulation platform and compare our approach to active redundancy. In Chapter 3 we present an analytical approach to derive the worst-case failover time and perform failover experiments with a distributed neural network on a hardware platform. In Chapter 4 we apply graceful degradation to stateful applications and investigate the effect of the state data loss. Furthermore, we present an approach to determine an optimal checkpointing period to reduce networking and processing overhead. Afterwards, we perform a case study with a SLAM application and conduct experiments on our distributed hardware platform. In Chapter 5 we present a formal analysis to derive the reliability of critical and non-critical applications in a gracefully degrading system. Here, we analyze the effects of graceful degradation and our allocations and reservation strategies on the reliability of critical and non-critical applications. In our experiments, we further analyze the resource consumption of the approaches and compare it to active redundancy. Last, we conclude our findings and propose further research directions for future work in Chapter 6.

# 2 Fail-Operational Automotive Architecture

## Contents

## 2.1 Introduction

In numerous applications for self-driving vehicles, achieving a safe state through deactivation and isolation is not feasible. Consequently, the prevailing fail-safe and fail-silent methodologies are inadequate. A fail-safe system ensures a safe shutdown state in the event of a malfunction, whereas a fail-operational system requires the full operation of safety-critical functions. To attain fail-operational capabilities, duplication of the relevant components is indispensable [34]. However, integrating additional hardware resources to maintain the fail-operational status of safety-critical applications is expensive. When coupled with the predicted resource requirements of autonomous driving functionalities, hardware expenses would substantially increase.

As a result of the need to manage escalating costs and complexity, applications are now being integrated into more powerful multicore control units. This has resulted in the consolidation of existing ECUs and a shift towards a more centralized E/E architecture [1]. This trajectory is anticipated to persist, with future E/E architectures potentially comprising just a handful of highly capable central controllers.

On the other hand, when viewed from the software standpoint, this trend results in decentralization. In contrast to the current, state-of-the-art monolithic ECU software, the software of the future is expected to adopt a modular approach. This modular design will enable the dynamic repositioning of software components during runtime, including the ability to activate and deactivate components across different ECUs. This perspective allows new strategies to enable fail-operational behavior.

Instead of using active redundancy, where a task replica is actively executed, or passive redundancy, where the system is oversized such that sufficient resources are available once a passive task is started, graceful degradation can be applied. Here, passive redundant tasks with higher priority can reuse the allocated resources of tasks with lower priority. Once a passive task with higher priority is started, lower priority tasks are disabled to free resources. This way, the non-critical functionality of an automotive system can be degraded in a failure scenario to keep safety-critical functionality [35]. The advantage of such an approach is that existing hardware resources in the system can be repurposed at run-time to lower the hardware costs.

State-of-the-art design-time methods presented in [35] are not applicable for highly customizable automotive systems as they require a re-evaluation for every change in the software system. Furthermore, a configuration for optimal mapping and task activation must be evaluated and stored for each single failure combination. With customizable and frequently updated software, considering every unique system configuration is impossible.

Thus, dynamic resource management, which maps applications at run-time as part of the software platform, is required. Dynamic resource management allows integrating new applications at run-time with unique solutions for an individual mix of applications. Furthermore, it enables a gracefully degrading system behavior and allows the system to react to unplannable changes, such as the defect of a hardware unit.

In the following, we introduce our fail-operational architecture which consists of a graceful degradation approach to allocate and reserve resources. We present our decentralized agent-based approach to find a valid application mapping. In an agent-based system, the system's control is decentralized, so there is no single point of failure, and the system can still act after any ECU failure. In dynamic mapping, an agent is responsible for finding the mapping of a single application or task at run-time. In our work, we combine graceful degradation with an agent-based system. Furthermore, our fail-operational architecture ensures a predictable timing behavior of real-time applications by using our performance analysis for gracefully degrading systems to evaluate mappings.

This chapter is structured as follows. First, we introduce our system model in Section 2.2, which applies to all chapters in this thesis. In some sections, we make some additions to the model that are only relevant to the corresponding section. Then, we present our agent-based graceful degradation approach in Section 2.3 to dynamically find application mappings while enabling a gracefully degrading system behavior. Afterward, we extend our graceful degradation approach in Section 2.4 to a scheduling approach as a basic module to enable a further timing analysis of the system. Based on this approach, we present our performance analysis of a gracefully degrading system, which evaluates if an application with a given mapping adheres to timing constraints and considers passive task instances. This ensures that in any single ECU failure scenario, a valid mapping is available that respects the timing constraint of the application. Furthermore, we extend our agent-based approach such that a backtracking algorithm is applied if the performance analysis of a mapping evaluates negatively. While this approach solves the mapping problem reasonably, we discuss shortcomings and limitations regarding scalability and reconfiguration time in Section 2.5. Last, we conclude our findings about our fail-operational architecture in Section 2.6.

## 2.2 System Model [1]

In the following, we introduce our system model used throughout this chapter which builds the base system model for the remaining chapters. This system model is slightly expanded in some sections, but these additions remain only valid for the corresponding chapter or section. We first introduce our hardware architecture and motivate our reasoning for the decision in Subsection 2.2.1. Afterwards, we define our classification system with two criticality levels in Subsection 2.2.2. In Subsection 2.2.3 we introduce our application model consisting of tasks and message instances. Last, we present our failure and failover model in Subsection 2.2.4.

### 2.2.1 Architecture

In the past, automotive vendors added a new ECU for each new functionality in the vehicle. Today, cars often consist of more than 100 ECUs to control functions in the domains such as infotainment. Now the automotive industry is aiming towards zonal or more centralized architectures. Some vendors such as Tesla prefer a centralized architecture, where most functions are executed on a single ECU such as the Full Self-Driving Computer of Tesla [6]. Bosch is developing a vehicle-centralized, zone-oriented E/E architecture with a few centralized, powerful vehicle computers integrating cross-domain functionality similar to [5, 7]. These vehicle computers are connected to actuators and

[1]Major parts of this section have been published in [29].

sensors via zone ECUs. This reduces the required wiring and weight in vehicles but also system complexity.

In our work, we focus on deploying bigger applications on a future system architecture consisting of a set of a few ECUs $e \in E$ which are interconnected via switches and a set of Ethernet links $l \in L$. The ECUs and Ethernet links use Time-Division Multiplexing (TDM) scheduling with pre-determined time slices which can be allocated or reserved. We implemented a middleware based on SOME/IP [36], an automotive middleware solution to dynamically activate, deactivate, and move tasks on the platform at run-time. This middleware includes a decentralized service discovery to dynamically find services in the system and a publish/subscribe scheme to publish and subscribe to events.

### 2.2.2 Criticality

In our work, we are exploring graceful degradation methodologies. Here, critical applications, e.g., for autonomous driving can be restarted after a failure on another ECU. Instead of exclusively reserving resources for this scenario, non-critical applications, e.g., from the infotainment domain can be shut down to free resources.

According to the ISO 26262 standard, applications can be assigned one of four Automotive Safety Integrity Levels (ASILs) (A to D) [37]. However, we do not differ between criticality levels of critical applications in our work as there is no justification in shutting down applications with an assigned ASIL of A for an application with an assigned ASIL of B as the failure of any critical application can have safety-critical consequences. Instead, it has to be ensured that all safety goals are met for any critical application. Therefore, we only distinguish between critical and non-critical applications. In our work, we assume that each critical application has fail-operational requirements. This means that the application has to remain operational even if a failure occurs that affects this application.

- *Critical application*: An application that has fail-operational requirements. To ensure a fail-operational behavior passive redundancy on task level is applied. All critical applications have the same priority.

- *Non-critical application*: An application without specific safety requirements. Non-critical applications can be shut down to free resources for critical applications even if they are not directly affected by a failure.

### 2.2.3 System Software

Our system software consists of a set of applications $a \in A$ which can be subdivided into a set of safety-critical applications $A_S$ and a set of non-critical applications $A_N$: $A_S \cup A_N \subseteq A$. Applications are executed periodically with a period $P_a$ and we assume each application has to meet a deadline $\delta$, with the period $P_a$ being at least as long as the deadline $\delta$. We assume that the Worst-Case Execution Time (WCET) $W(t)$ is known for every task.

We model each application $a$ by an acyclic and directed application graph $G_A(V, E)$. Safety-critical applications must fulfill fail-operational requirements and, thus, must remain operational even during critical ECU failures. Therefore, we assume that redundant passive task instances are required for our safety-critical applications. The vertices $V = T_a \cup T_p$ of the application graph $G_A(V, E)$ are composed of the set of active task instances $T_a$ and the set of passive task instances $T_p$.

- *Active task instance*: A task instance of a critical or non-critical application. This is the default task instance actively executing any workload. A binding $\alpha : T \to E$ assigns an active task instance $t \in T$ to an ECU $\alpha(t) \in E$.
- *Passive task instance*: A backup instance $t \in T_p$ of an active task instance which is part of a critical application. The passive task instance is only activated if its active counterpart is affected by a failure. The binding $\beta : T \to E$ assigns a passive task instance $t \in T_p$ to an ECU $\beta(t) \in E$.

The edges $E = M_a \cup M_b$ of the application graph $G_A(V, E)$ are composed of the set of active messages $M_a$ and the set of backup messages $M_b$.

- *Active message instance*: A routing is required for each active message instance $m \in M_a$ which is part of a critical or non-critical application. A routing $\rho : M \to 2^L$ assigns each message $m \in M_a$ to a set of connected links $L' \subseteq L$ that establish a route $\rho(m)$.

We use the shortest path routing obtained through Dijkstra's algorithm such that there is only one route between two ECUs $e$ available [38].

- *Backup message instance*: For critical applications three backup message instances $m \in M_b$ are required of which one will get activated after an ECU failure depending on which passive task instances get activated. A routing $\sigma : M \to 2^L$ assigns each backup message $m \in M_b$ to a set of connected links $L' \subseteq L$ that establish a route $\sigma(m)$.

The application graph of non-critical applications consists only of active task instances $t \in T_a$ and active messages $m \in M_a$.

Figure 2.1 presents a non-critical and a critical application according to our system model. The non-critical application consists of two tasks and one message being sent between the two tasks. The graph also includes two passive task instances and three backup message instances for the safety-critical application. Three backup message instances are required to ensure that communication between two task instances is always possible regardless of which task instances are affected by a failure. The message instances $m_{0,ab}$ and $m_{0,ba}$ are required if only one active task instance fails. In contrast, the message instance $m_{0,bb}$ is required if both active task instances are affected by a failure, e.g., because they are mapped onto the same failing ECU.

Figure 2.2 shows the binding $\alpha$ of the active task instances $t \in T_a$ and the binding $\beta$ of the passive task instances $t \in T_p$ onto a system architecture. The routing $\rho$ of the active message instance $m \in M_a$ and the routings $\sigma$ of the backup message instances $m \in M_b$ are also marked by colored arrows.

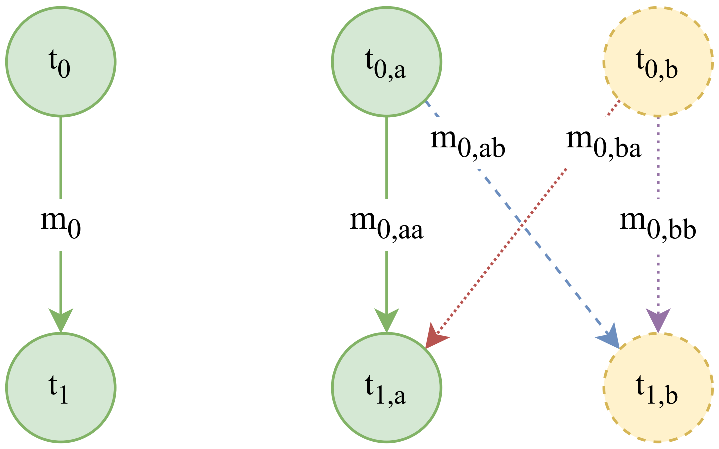


**Figure 2.1:** An exemplary application graph $G_A(V, E)$ of a non-critical application (left) and a critical application (right). The application graph of the critical application consists of two active task instances $t_{0,a}, t_{1,b} \in T_a$ and two passive task instances $t_{0,b}, t_{1,b} \in T_p$. Furthermore, it contains one active message instance $m_{0,aa} \in M_a$ and three backup message instances $m_{0,ab}, m_{0,ba}, m_{0,bb} \in M_b$, which are required to ensure there is always a communication path between the tasks instances available.

### 2.2.4 Failures

This work and our experiments focus on mitigating ECU failures detected by watchdogs and heartbeats. Here, we follow the definitions of [37] and [39], distinguishing between the terms fault, error, and failure. A fault is the cause of an error, which might potentially manifest as a failure. It can appear for a short period (transient), intermittently, or permanently. A fault is an abnormal system condition that can only be detected by the system when it causes an error. An error describes a discrepancy between actual and theoretically correct values. An error may become intolerable and result in failure. In this thesis, we focus on mitigating random hardware failures and failures caused by physical and permanent faults. Our approach could also mitigate failures caused by transient faults if hardware or software supports the corresponding failure detection mechanisms, e.g., through a lock-step architecture. If possible, a failure caused by transient faults should be handled locally. Our solution is intended to be used as a last resort, as the isolation of an ECU and a failover could lead to the shutdown of non-critical applications when graceful degradation is applied.

We define a failure $f \in F$ with $F \subseteq E$, where $f$ identifies the failed ECU. Our graceful degradation approach ensures that safety-critical applications can keep running after an ECU failure while there is no guarantee for non-critical applications. As redundancy is used for all safety-critical tasks, the system could withstand any single ECU failure and perform failover for affected task instances. After a failover, any critical application can remain operational such that any hazards can be avoided. After a time-critical failover, fail-operational capabilities must be re-established with a new mapping process. Applications might have to be stopped temporarily if active task instances have to be

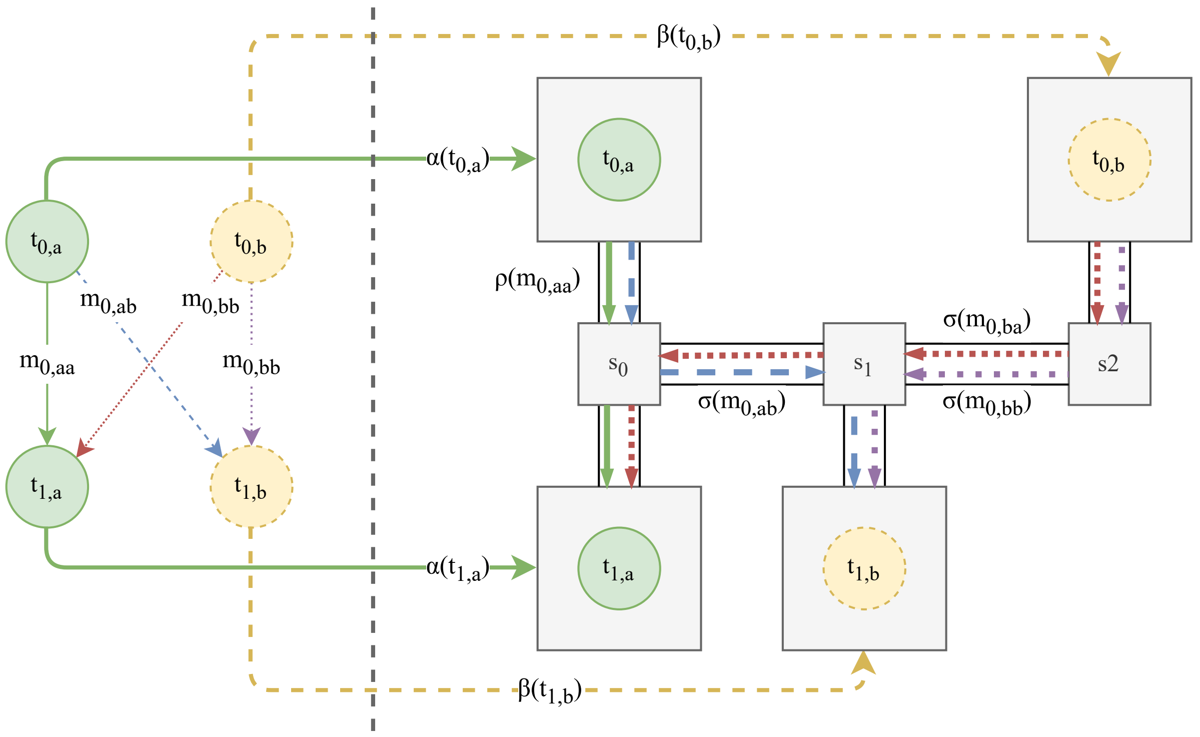


**Figure 2.2:** Exemplary mapping of a safety-critical application onto a hardware architecture consisting of four ECUs $e_0$, $e_1$, $e_2$, and $e_3$, and three switches $s_0$, $s_1$, and $s_2$. The green arrows indicate the active bindings of tasks $t_0$ and $t_1$, while the dashed yellow arrows indicate the passive task bindings. The same arrow color and style indicate the routings of the message instances as in the application graph.

remapped such that a safe mapping can only be performed during a halt. Methods such as those proposed in [40] to perform a safe real-time task migration could be applied to prevent this. While our approach considers that redundant message instances are required to ensure communication is possible after a failover, mitigating network or switch failures is out of the scope of this work. We recommend the work of [41] on this topic for the interested reader.

In the case of an ECU failure we consider that the current application execution might not finish if an active task instance is affected directly by the failure and that application execution might be interrupted for a specific time interval. Here, it is vital that a failover within the FTTI can be guaranteed [42]. We cover the topic of failover timing analysis in Chapter 3. To save critical application states, checkpoints can be periodically transmitted from active to passive task instances to save essential state data covered in Chapter 4. After the failure recovery computation can be continued with the latest transmitted checkpoints. Now, that we have introduced our system model we present our fail-operational architecture in the following two sections.

## 2.3 Agent-Based Graceful Degradation [2]

After presenting our system model in Section 2.2, we introduce our agent-based graceful degradation approach. While we introduce our basic degradation approach in this section, we extend it in Section 2.4 to a scheduling approach and adjust our agent-based approach to handle mappings that do not respect timing constraints. In this section, we make the following contributions:

- We analyze related work in the domains of fail-operational systems, graceful degradation, and dynamic mapping approaches in Subsection 2.3.1 and identify that combining graceful degradation and dynamic mapping approaches have not been considered in literature yet.
- Based on our additions to the system model introduced in Subsection 2.3.2, we present our agent-based approach, which is depicted in Figure 2.3, to find task mappings and activations at run-time in Subsection 2.3.3. The fail-operational requirements of safety-critical applications can be satisfied by using graceful degradation. Here, the dynamic nature of the agent-based system allows an easy reconfiguration to re-establish the fail-operational behavior after ECU failures.
- We use our in-house developed simulation framework to evaluate the approach in Subsection 2.3.4 and summarize our findings in Subsection 2.3.5.

### 2.3.1 Related Work

The related work for our approach can be mainly separated into the three domains of fail-operational systems, graceful degradation, and dynamic mapping approaches. To our knowledge, no other work in literature combines graceful degradation with a dynamic agent-based approach.

#### 2.3.1.1 Fail-Operational Systems

Ensuring fail-operational behavior is crucial in emerging automotive systems to meet safety requirements. This importance is amplified by the ongoing trend in automotive technology towards autonomous driving, where there may be no human driver to take over in the event of a failure. Consequently, relying solely on fail-safe and fail-silent approaches, which aim to minimize disruption in the event of a failure, is no longer adequate. Traditionally, achieving fail-safe behavior involves continuous system monitoring, hardware redundancy (resource replication), or implementing specific shutdown procedures, as discussed in reference [43]. However, with the shift towards autonomous driving, more advanced fail-operational strategies are essential to ensure the safety and reliability of these systems.

The solutions mentioned earlier, come with trade-offs in cost, required physical space, and manufacturing complexity. These trade-offs primarily arise due to the need for

[2]Major parts of this section have been published in [28].

additional hardware components and redundancy, as discussed in [44]. To address these trade-offs and provide a comprehensive understanding of fault tolerance in automotive systems, the authors [18] offer an overview of diverse fault-tolerant designs. These designs aim to achieve varying levels of fail-operational, fail-silent, and fail-safe systems by employing different degrees of redundancy, particularly in the context of drive-by-wire systems. This research helps explore a spectrum of fault tolerance options to strike a balance between system reliability and associated costs.

In [34], the authors provide an overview of existing fail-operational hardware approaches and introduce concepts for implementing a multi-core processor. This work focuses on hardware-based fault tolerance strategies. On the other hand, in [45], the authors review common fault-tolerant architectures in System-on-Chip (SoC) solutions, including lock-step architectures, loosely synchronized processors, and triple modular redundancy. They conduct a trade-off analysis to assess the strengths and weaknesses of these architectures. These detection and mitigation mechanisms presented in both references serve as a foundation for enabling dynamic fail-operational solutions at the software level. However, it's important to note that these approaches primarily address fail-operational aspects at the device level and may have limitations in terms of flexibility and system-wide implementation. In contrast, our system-wide graceful degradation approach goes beyond the device level. It leverages the ability to restart applications and utilize non-critical application resources to reduce resource overhead introduced by redundancy. This approach offers a more comprehensive and flexible solution for achieving fail-operational behavior at a system-wide scale.

The RACE project represents a significant attempt in the development of fault-tolerant software and system architectures designed specifically for future electric vehicles. This architecture was not just a theoretical concept but was practically implemented in two prototype vehicles, highlighting its real-world applicability. Within the project, a centralized computation architecture was proposed, as detailed in [46]. This architecture is based on a cross-domain system topology and incorporates a Runtime Environment that executes real-time applications of mixed-criticality. The primary objective is to enable fail-operational behavior, ensuring the continued operation of critical vehicle functions even in the presence of faults, failures, or errors, as discussed in [47]. The assumption made by these authors, and similarly by us, is that applications are executed on a limited number of powerful and homogeneous ECUs within a mixed-critical environment. This approach streamlines the design and management of complex automotive systems, with a focus on ensuring reliable and fault-tolerant operation.

In [48], a method outlines a hardware architecture designed to comply with the rigorous functional safety standards of IEC61508 and ISO26262. This method achieves its goal by incorporating a pre-certified hardware fault supervisor, which helps optimize costs and streamline the certification process. However, it's worth noting that this approach primarily concentrates on the hardware aspect of functional safety and fault tolerance. It may not fully address the growing complexity of functions and applications running on these ECUs. As automotive systems become increasingly sophisticated, ensuring safety and reliability requires not only robust hardware but also comprehensive strategies for managing software complexity and fault tolerance, which may extend beyond the hardware level.

Approaches like the ones outlined in [49] and [50] adopt a dual lock-step architecture that employs two identical Central Processing Units (CPUs) to execute the same software tasks. This architecture serves as a fault tolerance mechanism. In this setup, the first CPU operates in a live mode and is responsible for controlling the system during normal operational conditions when no fault is present. Simultaneously, the second CPU continuously monitors the status of the first CPU at every clock cycle. This monitoring ensures that potential faults or discrepancies in the first CPU's behavior are promptly detected. The authors propose a trade-off between system performance and fault coverage. This dual lock-step architecture can function as two fail-silent channels, ensuring that any faults are handled in a way that does not disrupt the system's operation. Alternatively, it can be configured as a single fail-operational unit, allowing the system to continue functioning even in the presence of faults.

In [51], the authors introduce a coded processing approach that operates by coding both data and instructions within a computing system. This approach is designed to enhance fault tolerance. In a system employing this technique with two coded channels, both channels are simultaneously active on either different partitions of a core or, in the case of a multi-core system, on multiple cores. This redundancy ensures that even if one channel experiences a failure, the service or application can seamlessly continue running on the other channel. The approach described in [51] offers notable improvements in Mean Time to Failure (MTTF), which represents the system's reliability.

Another approach, as presented in [52] and later extended by [53], adopts a simplex architecture to ensure system safety. A simplex architecture achieves safety at the application level by simplifying complexity, following the principle of "using simplicity to control complexity" [54]. In this architecture, two key subsystems are employed: a safety controller subsystem and a high-performance control subsystem. During regular system operation, the high-performance subsystem is utilized to maximize system performance. However, in the event of a fault or failure, the system can switch to using the safety subsystem to ensure stability and reliability. Using the simplex architecture, the authors propose a hardware/software approach that guarantees fail-operational behavior and a fault-tolerant system. This method is capable of handling logical application-level faults and faults in dependent layers, including real-time operating systems.

In [55], the authors introduce a system-level simplex architecture designed to ensure the fail-operational behavior of applications while safeguarding the underlying operating system, middleware, and microprocessor from failures. In this architecture, a complex subsystem takes charge of driving the system as long as no failure is detected. A safety subsystem and a decision controller run on a dedicated microcontroller to provide fault tolerance. Similarly, authors in [56] present a fail-operational simplex architecture and address the challenge of inconsistent states in CAN controllers during a failover. To resolve this issue, they employ an atomic function that stores the state and sends messages to prevent inconsistencies, particularly in communication with peripherals. Their work demonstrates that dynamic reconfiguration of a set of functions at the system level is feasible, and redundancy in safety-critical functions with dynamic reconfiguration can reduce the hardware redundancy required in E/E architectures. While these approaches bring fail-operational capabilities to the system level, they do have limitations in terms of flexibility and cost. The inclusion of a dedicated hardware component is necessary, which can add to the overall system expenses.

In [41], researchers address the automatic optimization of redundant message routings within automotive Ethernet networks to enable fail-operational communication. Their work primarily focuses on enhancing communication component redundancy to ensure reliable network operation. In contrast, our approach concentrates on mitigating ECU failures rather than communication component failures. Within our methodology, redundant tasks are distributed across the system, and redundant communication routes are established. This ensures that when a task is restarted, there is always at least one communication path with preceding and succeeding tasks.

Overall, hardware components with failure detection and mitigation mechanisms are the base for enabling a dynamic fail-operational solution on a software level. However, the presented approaches lack flexibility and do not deal with the problem of providing a dynamic fail-operational behavior for an entire system consisting of many applications distributed over multiple ECUs.

#### 2.3.1.2 Graceful Degradation

Graceful degradation, also known as functional degradation, is a strategy that involves suspending the execution of less critical applications within a system to ensure the continued functionality of more important applications, as described in [57]. In this approach, critical applications are prioritized and kept operational, while non-critical applications may be temporarily halted or slowed down. One of the key advantages of graceful degradation over other redundancy approaches, such as active redundancy, is its potential to reduce hardware costs significantly. By strategically managing the allocation of resources and prioritizing critical functions during fault or failure conditions, graceful degradation can help maintain system functionality while minimizing the need for redundant hardware components.

In [58] and [59], the authors present an architectural framework for reliable autonomous vehicles, which includes mechanisms for fault tolerance and degradation. The SAFER architecture achieves fault tolerance through various redundancy strategies, such as cold standby slaves, hot standby slaves, and task reruns. They also consider a form of degradation that involves reducing processor utilization by extending the execution period of tasks, resulting in less frequent task execution and a potential decline in quality of service. In contrast, our degradation approach differs in its focus. We emphasize the complete shutdown of non-critical applications to free up system resources for critical applications. Our approach does not specifically address the degradation of functionality within a single application by extending task execution periods. Instead, it prioritizes the availability and performance of critical applications over non-critical ones by releasing resources when necessary.

The SafeAdapt project, as described in [60], introduces a generic adaptation mechanism as a functional safety concept to enable fail-operational capabilities in automotive systems. This mechanism involves deploying redundant application instances as hot-standby and cold-standby versions. In the event of a failure, the system can switch from a functional primary version to a simplified, degraded version to maintain functionality. Additionally, in [43], the authors propose a meta-modeling technique for describing architectural patterns for fail-operational systems. They introduce a graceful degradation pattern based on patterns from [61] and integrate it into a pattern meta-model library. Comparatively, our approach differs in several aspects. We consider degradation at a system level, allowing for resource optimization by reallocating resources from less critical applications to critical ones. In contrast, the approach presented in [60] requires exclusive resource allocation for standby versions. Moreover, our approach could be integrated into a pattern meta-model library, similar to [61], to provide developers with a broader set of options for designing fail-operational systems. This flexibility allows for more efficient resource utilization and enhanced fault tolerance.

There are similar concepts in the literature that focus on degrading the performance of an application in the event of a failure rather than shutting down non-critical applications, as discussed in [62] and [63]. Additionally, in [64], the authors propose a reconfiguration strategy based on the available hardware resources following a failure. This approach involves adapting the system's configuration to make the best use of the remaining resources after a failure occurs, ensuring continued operation.

In [65], the author provides a formal definition of graceful degradation by specifying a comprehensive set of system constraints that outline the tasks a system can perform based on these constraints. This formal definition helps establish what a system can achieve while degrading gracefully under specified conditions. On the other hand, [25] introduces a method for calculating the utility of a system by decomposing it into feature subsets, which can be defined by various functional or non-functional attributes. This decomposition enables an analysis of the system's utility in different degradation modes, providing insights into how the system's performance or capabilities can vary under different conditions. However, it's important to note that neither of these works proposes a specific approach for designing a system with graceful degradation behavior. Instead, they contribute to the formalization and analysis of graceful degradation concepts and metrics, which can be valuable for assessing and understanding system behavior under different conditions.

In [66], the authors introduce a degradation-aware reliability analysis that considers different degradation modes for tasks of varying safety levels. They use design space exploration to optimize the reliability of these degradation modes. At runtime, an algorithm monitors resource states and selects appropriate task mappings based on the observed conditions. On the other hand, [35] presents a design-time analysis focused on finding valid application mappings in mixed-critical systems. In this context, applications can have multiple redundancies based on their fail-operational level, and the system can be degraded by shutting down optional software components. Contrasting with both of these approaches, our method prioritizes re-establishing lost redundancy after a failover, rather than relying on a predetermined level of redundancy. We dynamically adapt to fault conditions, optimizing resource utilization to maintain critical functions. While the approaches in [66] and [35] use design-time analysis, our approach provides more flexibility for highly customized automotive systems by adapting to run-time conditions and resource availability.

#### 2.3.1.3 Dynamic Mapping

There has been a lot of work on dynamic mapping approaches. In [67], the authors introduce a decentralized mapping algorithm for Network-on-Chip (NoC) architectures. This algorithm involves predecessor tasks mapping tasks and considers constraints such as computational capacities. It optimizes routing based on a chosen goal function. Their approach uses a best-neighbor strategy that focuses on the closest search space around a task.

The work in [68] presents an agent-based runtime mapping approach for heterogeneous NoC architectures. The system employs global agents that contain system state information and cluster agents responsible for assigning resources. The primary motivation here is to reduce computational effort and global traffic when mapping distributed applications.

The authors in [69] introduce a centralized run-time mapping approach for reducing network load in NoC based Multiprocessor-System-on-a-Chip (MPSoC) systems. In their approach, a dedicated manager processor takes responsibility for initially mapping tasks of each application to specific clusters within the MPSoC. As the workload dynamically changes, subsequent tasks are mapped at runtime based on communication requests. To efficiently distribute tasks and communication across the system, the authors propose and evaluate multiple heuristics that consider the channel load within the NoC architecture. They argue that using greedy algorithms is reasonable, as these algorithms can quickly provide mapping solutions, even if they do not explore the entire search space exhaustively. The authors conclude that, compared to static optimization methods, the moderate overhead associated with solutions found through dynamic methods is acceptable, considering the increased flexibility gained from dynamic mapping to adapt to runtime conditions.

In addition to purely dynamic approaches, hybrid mapping schemes, combining design-time analysis and run-time reconfiguration, have been explored in research. One example is presented by the authors in [70], where they consider task migration as part of their approach. This involves allocating resources at runtime to migrate tasks as needed. However, in contrast to reactive schemes, the authors in [71] propose a proactive approach, where task mappings are changed at runtime to prevent imminent hazards. In our approach, we focus on finding task mappings to establish passive redundancy, enabling us to react to failures as they occur.

The state-of-the-art approaches discussed have limitations, especially when applied to highly customized automotive systems due to their reliance on design-time analysis. Additionally, centralized approaches, such as in [69], introduce single points of failure into the system. Furthermore, these approaches do not consider fail-operational or timing requirements, and none of the existing dynamic or hybrid mapping approaches have built-in support for graceful degradation. As a result, they are not suitable for efficiently achieving fail-operational behavior in safety-critical applications.

### 2.3.2 System Model Additions

In the following, we assume that each of the ECUs $e \in E$ has a CPU budget $C(e)$. Each bi-directional link $l \in L$, that connects an ECU $e$ with the gateway, has a bandwidth budget $BW(l)$. Furthermore, we assume that the resource consumption of the CPU $c(t)$ per task is known. Similarly, we assume the bandwidth requirement $bw(m)$ of a message $m$ as given.

### 2.3.3 Agent-Based Degradation

To cope with the high number of customized configurations in future automotive architectures, we investigate the effectiveness of applying agent-based strategies on the task level to achieve gracefully degrading system behavior.

In contrast to active redundancy, where redundant tasks would actively run and use CPU resources, passive redundant tasks only reside in the memory. Only when they are activated and replace a failed task, will they require the same amount of resources. Thus, it has to be ensured that sufficient resources are available on the startup of the passive redundant task. Instead of allocating the required resources without using them, graceful degradation can deactivate other less critical tasks to free the required resources. With this approach, the system loses some of its non-critical functionality. On the other hand, this allows us to save costs as our graceful degradation approach completely avoids the computational overhead that active redundant tasks would induce.

The FTTI describes the time that a fault can be present in the system before a hazard could occur and has to be determined for each safety goal according to the International Organization for Standardization (ISO) 26262 [37]. We assume that all tasks can be restarted within their assigned FTTI and, thus, focus on the aspect that sufficient resources must be provided once a passive redundant task is activated to ensure a predictable system behavior. We address the issue of failover timing analysis in Chapter 3. From a timing perspective, to achieve the restart within the FTTI, it is necessary for a redundant task to operate on the same data as the active task. For passive redundant tasks, a checkpointing approach has to provide the active task's status in a periodic fashion such that a restart within the FTTI can be ensured. We present a checkpointing-based approach in Chapter 4.

#### 2.3.3.1 Agent-Based System

With our agent-based methodology on task level we provide a way to ensure that all safety-critical tasks in the system have a passive redundancy and, once they are started, sufficient resources are provided to execute the task. Furthermore, the approach maintains communication with preceding and succeeding tasks. To mitigate multiple failures, the system can reconfigure and re-establish the redundancy of safety-critical tasks. The system has a predictable behavior as the mapping ensures the reservation of sufficient resources for the passive redundant tasks, such that they can start in a failure scenario by disabling non-critical tasks.

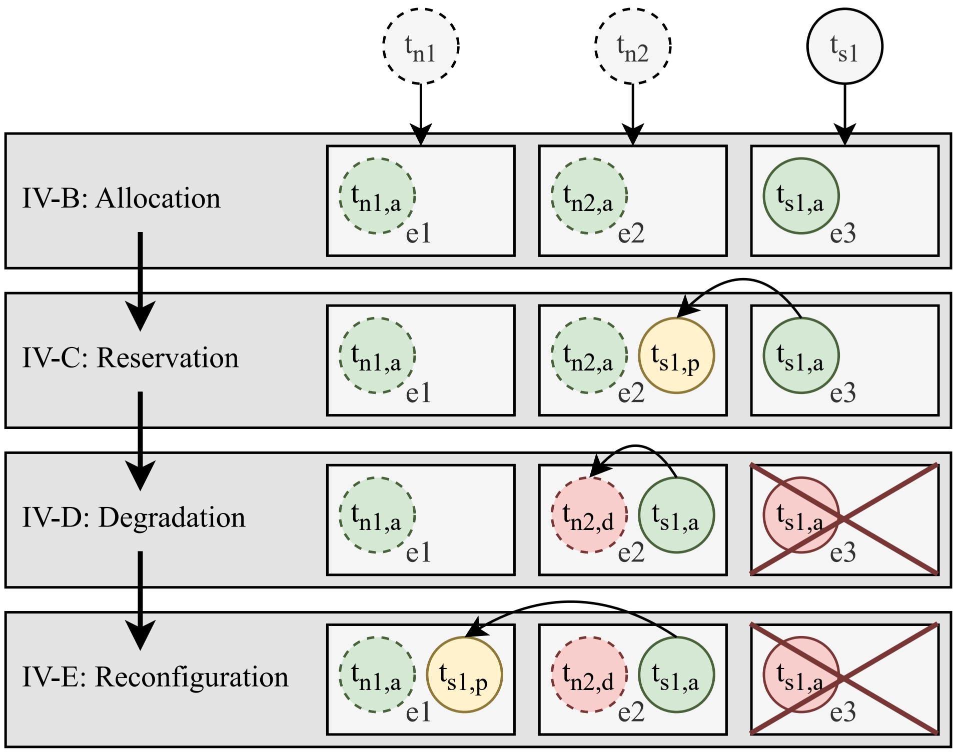


**Figure 2.3:** The safety-critical task $t_{s1}$ and the two non-critical tasks $t_{n1}$ and $t_{n2}$, each being part of a distinct application, is deployed on three ECUs using our agent-based approach as will be described in Subsection 2.3.3. The task states are depicted in green (active), yellow (passive), and red (failed/deactivated).

In our approach, both active and passive tasks are wrapped with an agent, which is in the following referred to as a *passive* or *active task agent*. The task agents are responsible for the proper execution of their respective tasks and for fulfilling their fail-operational requirements. The task agents themselves are always active and able to react to failures. We also use the task agents to find a valid mapping for both the active and the passive tasks. To achieve a dynamic behavior, the task agents can move on the system from one ECU to another.

Furthermore, each of the ECUs is running an *ECU agent*, which starts the task agents on startup. In addition, they handle the requests from task agents to start a new redundant task agent or to move to the corresponding ECU. Note that, in contrast to a design-time optimization, agent-based approaches can dynamically and continuously optimize the mapping concerning metrics such as link load at run-time.

The flow of our agent-based approach to finding task mappings and activations at run-time is depicted in Figure 2.3 and includes the following steps:

- Allocation 2.3.3.2: In the initial mapping process the active task agents allocate resources to find a valid mapping.

- Reservation 2.3.3.3: The task agents reserve resources at other agents, determining how the system will be degraded.
- Degradation 2.3.3.4: As an immediate failure reaction passive tasks are started and the system is degraded.
- Reconfiguration 2.3.3.5: By repeating the reservation process, the fail-operational behavior can be re-established.

### 2.3.3.2 Resource Allocation

Initially, the ECU agents start all task agents depending on a configuration. On startup, the active task agents send requests to the ECU agents to allocate CPU resources and link resources for their incoming messages. The amount of allocated CPU resources $c_{alloc}(t,e)$ and link resources $bw_{alloc}(m,l)$ is stored at the corresponding ECU agent. If it is impossible to allocate sufficient resources for mapping on the current ECU, the task agents will request other available ECU agents. When a valid mapping can be found, a task agent moves to the corresponding ECU and starts its task. Succeeding task agents wait on their predecessors to find a valid mapping to allocate the correct link resources for their incoming messages. This allocation process ensures that the allocated CPU resources of all tasks in the system do not exceed the CPU budget of any ECU:

$$\forall e \in E : \sum_{a \in A} \sum_{t \in T_a} c_{alloc}(t,e) \leq C(e) \tag{2.1}$$

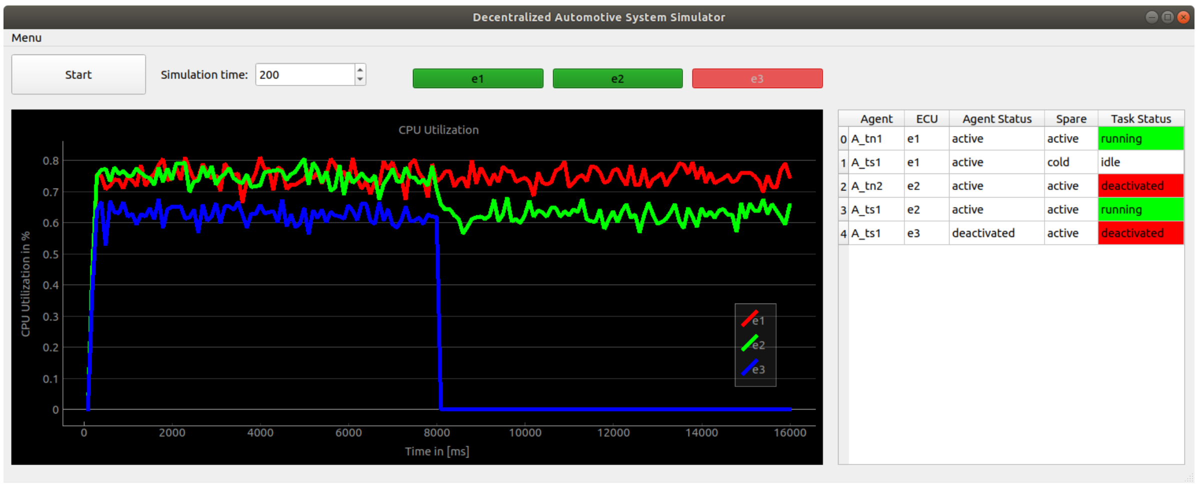
Similarly, it is ensured that the allocated bandwidth of all messages does not exceed the bandwidth budget of any link:

$$\forall l \in L : \sum_{a \in A} \sum_{m \in M_a} bw_{alloc}(m,l) \leq BW(l) \tag{2.2}$$

### 2.3.3.3 Resource Reservation

Any safety-critical task agent will start a redundant passive task agent on a different ECU. The responsibility of the passive task agents is to ensure that sufficient resources are freed in case the task has to be activated. For that, task agents can *reserve* resources, that have been previously allocated, at non-critical task agents or so far unused resources at ECU agents. The agents at which the resources are reserved *promise* to free the resources if requested. If a passive task agent has to activate its task, it *claims* the reserved resources at the corresponding agents. Once a promising agent frees the claimed resources, it deactivates its task. Similar to the allocation process succeeding passive task agents wait on their predecessors to find a valid mapping and allocate a route.

Furthermore, active task agents must reserve a second route to preceding passive task agents (next to the allocated routes to the preceding active task agents) to ensure that a valid route is available at any time. On the other hand, assuming that only one ECU fails at a time, passive task agents only have to reserve one route to either the preceding



Decentralized Automotive System Simulator

Simulation time: 200

| | Agent | ECU | Agent Status | Spare | Task Status |
|---|---|---|---|---|---|
| 0 | A_tn1 | e1 | active | active | running |
| 1 | A_ts1 | e1 | active | cold | idle |
| 2 | A_tn2 | e2 | active | active | deactivated |
| 3 | A_ts1 | e2 | active | active | running |
| 4 | A_ts1 | e3 | deactivated | active | deactivated |

**Figure 2.4:** Simulation framework performing simulation of the example described in Figure 2.3. The plots show the CPU utilization of the three ECUs $e_1$ (red), $e_2$ (green), and $e_3$ (blue). The table on the right displays the situation at the end of the simulation. After the failure of ECU $e_3$ at $8000ms$, task $t_{s1}$ restarted on ECU $e_2$, leading to the shutdown of $t_{n2}$. As $t_{s1}$ requires slightly fewer resources than $t_{n2}$, the change can be examined in the plot.

passive or the preceding active task agents. If the active task agent and the preceding active task agent have the exact mapping, they would both fail simultaneously. In this case, the passive task agent must only reserve a route to the preceding passive task agent. If the active task agent and the preceding active task agents have different mappings, only one can fail simultaneously. Thus, if the active task agent fails, the passive task agent only needs to reserve a route to the preceding active task agent.

### 2.3.3.4 Degradation

Once an ECU failure is detected, passive task agents, that lost their active task agent, immediately claim their resources, update the allocation status at the ECU agent, and start their task. Promising task agents, whose resources are being claimed, free the resources and deactivate their tasks. In addition, the allocation and reservation status at all task agents and ECU agents is updated, so the allocated or reserved resources of failed agents are not lost.

### 2.3.3.5 Reconfiguration

The procedure from Subsection 2.3.3.3 can be repeated for any task agent that lost its redundant counterpart or a resource reservation. Here, the advantage of the agent-based approach is that no additional algorithm is required.

With this approach, the task agents ensure the fail-operational behavior of their application. The task-based reservation of resources has two specific advantages. First, deciding which tasks are shut down in a degradation scenario does not have to be met in the time-critical phase after a failure. Second, this approach allows us to predict the system's behavior if all passive task agents of an application reserve the required resources, a fail-operational behavior can be guaranteed.

#### 2.3.3.6 Example

In our example in Figure 2.3, a safety-critical and two non-critical tasks are deployed on three ECUs using our agent-based approach. After each task agent finds an ECU and allocates the resources, the safety-critical task agent responsible for $t_{s1,a}$ starts a passive task agent on $e_2$. This passive task agent reserves the required resources at the task agent responsible for $t_{n2,a}$. After the failure of ECU $e_3$, the passive task $t_{s1,p}$ on ECU $e_2$ is immediately started and its task agent claims its resources at the task agent of $t_{n2,a}$, who deactivates its task. In the last step, the fail-operational behavior of task $t_{s1,a}$ is re-established by starting another passive task agent on $e_1$ and repeating the reservation process.

### 2.3.4 Evaluation

We have implemented the approach described in Subsection 2.3.3 in our in-house developed time-discrete and event-based simulation environment. The framework has been developed to simulate an automotive hardware architecture and system software according to our model as described in Subsection 2.3.2. We use this framework to evaluate our agent-based approach from Subsection 2.3.3.

#### 2.3.4.1 Simulation Framework

The system parameters describing the hardware architecture and system software can be provided by a specification that uses the XML schema for specifications from the OpenDSE framework [72]. For the simulation environment, we chose a process-based Discrete-Event Simulation architecture based on the SimPy framework [73].

To dynamically activate, deactivate, and move tasks on the platform at run-time, we implemented a middleware based on SOME/IP [36], an automotive middleware solution. This middleware includes a decentralized service discovery to dynamically find services in the system and a publish/subscribe scheme to publish and subscribe to events. In addition, this middleware allows remote procedure calls. All tasks in the system communicate via this middleware and are modeled as clients and/or services. Tasks that have outgoing edges in our application graph $G_a$ are offered as a service to the system, which also publishes their messages as subscribable events. Tasks with ingoing edges behave as clients requesting the corresponding services and subscribing to the events. This service-oriented approach allows a dynamic reconfiguration of the system software at run-time, such that tasks can be restarted on other ECUs and still be found by their subscribers.

Hardware access to a CPU or Ethernet link is managed by interchangeable schedulers. For the simulation, a static-priority preemptive scheduler was used for access to CPUs and a static-priority non-preemptive scheduler for access to Ethernet links. Tasks in the system are triggered periodically if they are the anchor task of an application otherwise on the arrival of incoming messages. Once a task is triggered, it is scheduled for execution on the CPU. After it has been granted access to the resource, it keeps the resource busy and sends out its messages as soon as it has finished execution. The unicast messages are put on the link and forwarded by the central switch. On arrival at its destination, succeeding tasks waiting for the message are triggered.

Furthermore, we implemented the proposed agent-based system from Subsection 2.3.3 in our simulation framework. The agents use the same middleware and network interface for communication as the tasks in our system. The significant load the agents impose onto the system occurs during the initialization phase when all agents allocate and reserve resources. Furthermore, the size of the agent messages is relatively small, mainly consisting of a few bytes, compared to the message size of typical automotive applications. During the normal execution phase, the agents do not impact the system. They are only triggered by ECU failures, where the system has to immediately react to the failure and be reconfigured. With example applications implemented, a more detailed analysis of the overhead imposed by the agents on the system is possible.

To simulate ECU failures, the framework offers the possibility to shut down ECUs . ECU failures are detected with periodic heartbeats and watchdogs. Each ECU has a running service to offer its heartbeat, whose periodic event is subscribed by the watchdogs of other ECUs.

Figure 2.4 shows our simulation framework performing the simulation of the example from Figure 2.3, which has been described in Subsection 2.3.3.6. From the CPU utilization on the three ECUs and the status information shown, it can be observed that $t_{n2}$ was shut down on ECU $e_2$ and instead $t_{s1}$ has been restarted, which was formerly running on ECU $e_3$.

### 2.3.4.2 Results

To evaluate our agent-based approach we used a simulation setup of 6 ECUs and 25 applications, of which each consisted at least of ten tasks. We used the OpenDSE framework [72] to generate synthetic applications with workloads based on typical automotive applications. All applications had a period of 10 ms and the message sizes were set to 1500 bytes. The link speed of all links was set to 100 Mbit/s. To obtain feasible mappings in the initial mapping process, the combined CPU usage of all applications was set to 90% of the available system resources. We equalized the sum of required computational resources $c(t)$ within each application to obtain a better comparison. Although the actual workload is varied with a random distribution at run-time, we use the required computational resources $c(t)$ as the worst-case estimation for our allocation and reservation process.

Each configuration was run 50 times and the results (Figure 2.5 and Figure 2.6) show the corresponding mean values and standard deviation. We conducted the simulations with an Intel Xeon Gold 6130 CPU consisting of 16 cores running at 2.1 GHz and 128 GB of Random-Access Memory (RAM). The simulations were run for a simulation

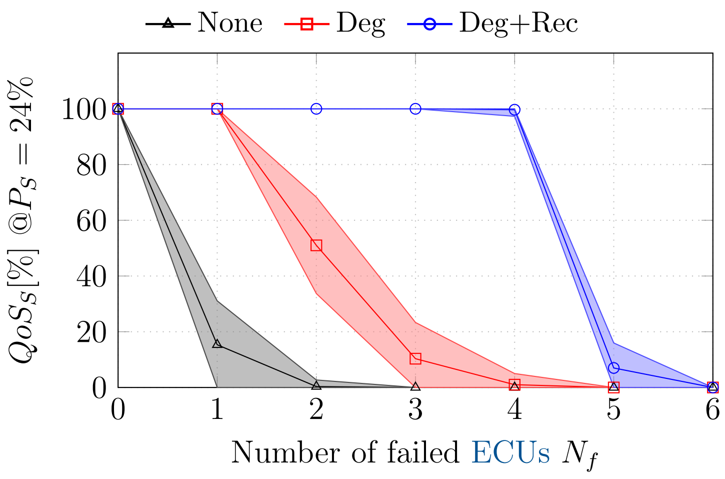


**Figure 2.5:** Simulated results with $|A| = 25$, $\forall a \in A : |T_a| \geq 10$, $|E| = 6$, $P_S = 24\%$ and 50 runs per configuration. Our agent-based approach using both degradation and reconfiguration significantly improves the percentage of operational safety-critical applications $QoS_S$ and the amount of ECU failures tolerated by the safety-critical applications.

time of 6000 $ms$. To simulate the failures we successively shut down a random ECU every 1000 $ms$. On average a single simulation run with a setup of 6 ECUs and at least 250 tasks took about 119.7 $s$ on one of the cores.

For our evaluation we define the metric

$$QoS_S(f) = \frac{|A_{S,f}|}{|A_S|}, f \in [0; |E| - 1] \tag{2.3}$$

where $A_{S,f}$ is the set of safety-critical applications which are running after the failure $f$ and $QoS_S(f)$ the percentage of operational safety-critical applications after the failure $f$. Similar, we use the notation $A_{N,f}$ and $QoS_N(f) = \frac{|A_{N,f}|}{|A_N|}$ for the non-critical applications. Furthermore, we define $P_S = \frac{A_S}{A}$ as the percentage of safety-critical applications in the system.

Our simulation results show that the amount of tolerated ECU failures increases significantly with our agent-based approach using both degradation and reconfiguration (Figure 2.5). In this scenario, the first degradation of the safety-critical system occurs in the majority of the cases after the fifth ECU failure compared to one failure tolerated by the approach without reconfiguration and zero failures tolerated without any replication.

Furthermore, we can observe that with an increasing percentage $P_S$ of safety-critical applications in the system, fewer failures can be tolerated until a safety-critical application fails (Figure 2.6a). This is plausible as an increasing amount of safety-critical applications have to share the same amount of resources. It can be also noticed that configurations with higher $P_S$ reach a $QoS_S$ close to 0% earlier while there would still be resources available. This comes from the fact that with more safety-critical tasks in the system, less passive task agents are able to reserve resources. Once a passive task

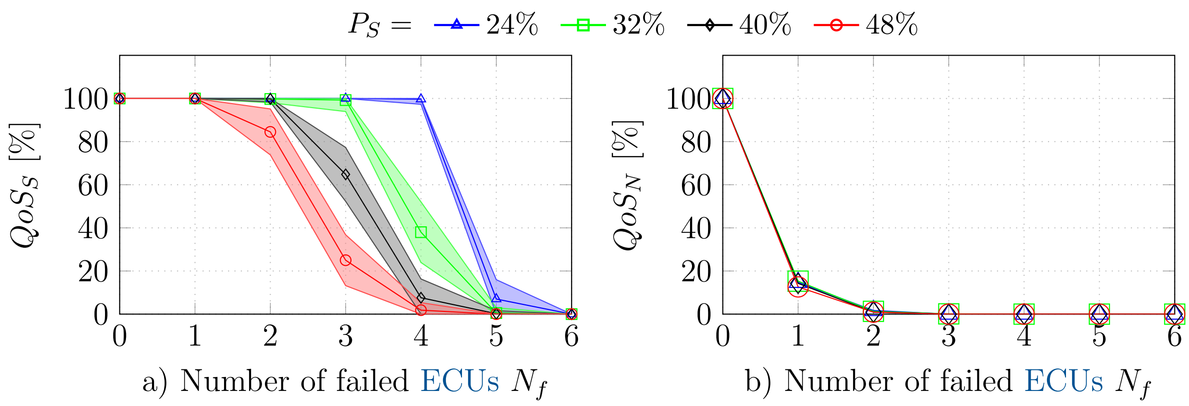


**Figure 2.6:** Simulated results with $|A| = 25$, $\forall a \in A : |T_a| \geq 10$, $|E| = 6$ and 50 runs per configuration. With an increasing percentage of safety-critical applications $P_S$ in the system, the percentage of operational safety-critical applications $QoS_S$ decreases earlier, and fewer ECU failures can be tolerated by the safety-critical system. There is no measurable impact of $P_S$ on the percentage of operational non-critical applications $QoS_N$.

agent is unable to reserve sufficient resources it is shut down and will not be restarted. Thus, even if resources would become available with the shutdown of other safety-critical applications after the next ECU failure, no new passive task agents are started.

There is no observable impact of $P_S$ on the percentage of operational non-critical applications $QoS_N$ (Figure 2.6b) as the curves behave relatively similarly. We explain this with the fact that the tasks of an application are distributed in the system and that an ECU failure leads to a high probability of the shutdown of multiple applications. This effect overlaps the number of applications that are being shut down due to degradation. We conclude that in a system with highly distributed tasks, the degradation has little impact on $QoS_N(f)$ as the non-critical applications would fail nevertheless.

Overall, the results indicate that our presented approach is able to significantly improve the tolerance of safety-critical applications against ECU failures. This improvement depends on the percentage of resources allocated by all safety-critical applications in the system.

### 2.3.5 Summary

In this section we have introduced an agent-based approach utilizing graceful degradation to ensure the fail-operational behavior of safety-critical automotive applications. The system finds task mappings and activations at run-time and is able to predict if the fail-operational behavior of an application can be guaranteed. Furthermore, the agent-based system is able to reconfigure itself after ECU failures and re-establish fail-operational behavior. In an experimental evaluation, we have shown that the number of tolerated ECU failures until a safety-critical application fails, can be improved significantly without using additional hardware resources. In the following section, we extend this approach to scheduling algorithms to allow a performance analysis of the system such that timing guarantees can be given to applications.

## 2.4 Predictable Timing Behavior [3]

The main challenge for a dynamic and decentralized system, as presented in Section 2.3, is to achieve a predictable system behavior. Most safety-critical applications have to meet real-time requirements, where a complete application execution has to finish within a deadline. Therefore, it is essential for a fail-operational architecture to ensure that timing constraints are respected not only by the actively running part of the application but also by the backup solution to which the system will switch during a failover.

However, no approach yet ensures a predictable timing behavior of gracefully degrading systems where passive task instances are activated after a failure. To achieve a fail-operational behavior, the timing constraints must be met under any circumstance. Finding a new task binding after a failure is unrealistic as the backup solution must be available immediately. Thus, an application binding can only be considered feasible if the deadline can be met after restarting any passive task instances and a viable backup solution is available for any possible failure.

In this section, we extend our graceful degradation approach from Section 2.3 to a scheduling approach to enable a further timing analysis of the system. To provide real-time guarantees, performance analysis has to be based on a composable system such that the interference between applications can be bounded [17]. The composable architecture allows us to estimate an upper bound of the execution time for active and passive tasks once they are started. Based on this approach, we present our performance analysis of a gracefully degrading system, which evaluates if an application with a given mapping adheres to timing constraints and considers passive task instances. Furthermore, we extend our agent-based approach such that a backtracking algorithm is applied if the performance analysis of a mapping evaluates negatively. Our approach includes passive task instances in the mapping search such that a backup solution is available that fulfills the real-time constraints under any ECU failure.

The mapping approach is intended to be performed while the car is not actively in use such that most system resources are available for the search. Once the system is stable and running, the agents do not perform any action, putting no additional strain on the system resources. Heartbeat messages and watchdogs are monitored during run-time to detect ECU failures. The solutions found by the mapping approach include valid backup solutions for critical applications to which a fast failover can be performed after a failure has been detected. Here, we follow our central hypothesis: a safe state can be reached when the failure occurs with minimal communication and computation. While we cover the topic of timing behavior in this section, a failover scenario is a unique scenario during which the system has to switch to a new, stable, and valid state. We cover the analysis of the failover time during which no application output might be produced in Chapter 3. After a failover has been performed, safety-critical applications can remain operational. However, the fail-operational behavior must be re-established to ensure a safe continuation. Depending on the remaining resources, re-mapping the applications could be performed during a safe halt in a parking space.

[3]Major parts of this section have been published in [29].

In this section, we make the following contributions:

- We analyze related work in Subsection 2.4.1 and provide an overview of related approaches in the field of predictable timing behavior. We conclude that no work allows a predictable timing or failover analysis of distributed gracefully degrading systems.
- We introduce and adapt a state-of-the-art performance analysis based on composable scheduling in Subsection 2.4.2.
- We present our performance analysis for fail-operational systems, which supports a gracefully degradable system behavior in Subsection 2.4.3. This analysis also considers backup solutions and can evaluate whether the worst-case end-to-end application latency can still meet the deadline after switching to a backup solution during a failover. Furthermore, we introduce our gracefully degrading scheduling scheme.
- We present our agent-based run-time mapping procedure in Subsection 2.4.4 using our performance analysis to find feasible mappings that meet real-time requirements. Our approach includes passive tasks in the search to ensure that all backup solutions meet the real-time constraints. Here, we also introduce three strategies that can strongly influence the degradation behavior. The system can be reconfigured after any failure to re-establish the fail-operational behavior of safety-critical applications.
- We evaluate our graceful degradation approach in our simulation framework in Subsection 2.4.5. Here, we compare our approach to active redundancy by measuring the success rate and resource utilization over multiple experiments. Furthermore, we evaluate and discuss our three allocation and reservation strategies. Results show that around twice as many critical applications can be mapped onto the same architecture when using our graceful degradation approach compared to active redundancy approaches. We conclude that graceful degradation can significantly increase the success rate in scenarios where resources are limited if the risk of losing non-critical functionality in a failure scenario is acceptable.

### 2.4.1 Related Work

As our work combines aspects from various research areas, we organized the related work section into four subsections. The topics of fail-operational systems, graceful degradation, and dynamic mapping are covered in Subsection 2.3.1. In the following, we focus on existing approaches to predictable timing behavior and failover timing analysis in literature.

The authors of [17] describe the concepts of predictability and composability, which can be used to reduce complexity and verify real-time requirements. In composable systems, applications are isolated to not influence each other, allowing us to verify their timing behavior independently. Furthermore, using formal analysis, lower bounds on performance can be guaranteed. The work of [74] uses hybrid application mapping to combine design-time analysis with run-time application mapping. The spatial and temporal isolation techniques and performance analysis are based on the concepts from [17]. At design time, a design space exploration with a formal performance analysis finds Pareto-optimal configurations. A run-time manager then searches for suitable mappings of these optimized solutions. In contrast to their work, we extend the scheduling techniques and include passive tasks and messages in the performance analysis to enable graceful degradation. Furthermore, our application mapping is performed entirely at run-time with an agent-based approach.

In the real-time mixed-criticality systems community, work has guaranteed reduced service to low-criticality tasks after switching to a safety mode when a critical task can not meet its deadline [75]. By contrast, we do not switch between two modes in our work but only shut down tasks if their resources have been reserved and are claimed by a critical task. Furthermore, the approaches in the literature mainly do not focus on distributed systems and do not consider fail-operational aspects.

The authors in [58] present a worst-case timing analysis for hot and passive standby tasks. However, the topic of graceful degradation is not addressed in this work.

Related to this topic, authors in [70] and [40] have analyzed the timing behavior of task migration at run-time if a new mapping has to be found. Their deterministic mapping reconfiguration mechanism identifies efficient migration routes and determines the worst-case reconfiguration latency. Run-time mechanisms combine offline design space exploration to find new application mappings and transition predictably to new configurations. In our work, we do not migrate the tasks to optimize the mapping but assume that the tasks are already deployed redundantly, making a direct failover possible. However, our approach ensures that new configurations meet timing constraints, and we analyze recovery time and failover effects on the timing behavior. In [76], the same authors and others present a general overview of hybrid application mapping techniques and composable many-core systems.

In [77], a fail-operational function-specific E/E architecture for brake and steering control is introduced that supports dynamic configuration. Several simulations are executed to derive the requirements for failure detection and fault reaction time, against which the presented architecture is tested. The authors further developed their approach in [56] by adding a hardware extension to prevent state loss that relies on CAN messages to communicate its state. They then integrate the architecture into a service-oriented architecture. Similarly to this approach, our architecture relies on communication through a service-oriented middleware. However, we do not rely on extra hardware and use periodic Ethernet heartbeat messages to detect the state of operation of the hardware devices.

In [78], the authors target distributed real-time embedded systems and aim to provide an automated design process for software reconfiguration. To approach this task, they define several "mode structures" characterized by a set of structured component types, each comprising several configuration instances. The transitions from one example to another are triggered by events related to system constraints or variations in the infrastructure. Our work goes beyond the system reconfiguration scope and covers the timing behavior analysis.

Overall, there has not been any work in predicting timing behavior that considers a gracefully degrading system architecture. Furthermore, none of the work discussed concerning the timing properties of safety-critical embedded systems has yet covered the topic of failover timing analysis. For this purpose, we build mainly upon our previous work in Section 2.3 and the result from [74] to create, for the first time, a comprehensive approach that enables a safe and efficient fail-operational behavior of time-critical applications in distributed systems by predictably analyzing the execution time behavior and allowing a gracefully degrading system behavior.

### 2.4.2 Performance Analysis

In this subsection, we contribute our concept of performance analysis for distributed applications based on the work from [74] and [17]. Using this performance analysis, we present our performance analysis of gracefully degrading systems in Subsection 2.4.3. Instead of targeting NoC architectures as in [74], we target a distributed electronic system consisting of multiple ECUs, which are connected via switches and Ethernet links. The goal of the performance analysis is to find an upper bound for the end-to-end application latency such that it can be verified whether a mapping adheres to a timing constraint. Here, we formally introduce the end-to-end application latency in Subsection 2.4.2.1 and present our derived analytical formulas for the distributed and composable system. Afterward, we present composable scheduling with our adapted analytical formulas for task and message scheduling in Subsection 2.4.2.2. Compared to the work in [74], we chose TDM instead of a Round-Robin (RR) scheme for task scheduling. The disadvantage of RR scheduling is that the execution latency depends on the number of other tasks in the schedule. The execution latency might change once tasks are added to the schedule later. By contrast, when using a schedule based on TDM, an upper bound of service intervals that tasks can allocate is already predefined, which can be used for a worst-case estimation of the execution latency. For message scheduling, we also use time-division multiplexing, where we design the system such that precisely one Ethernet frame can be sent per slot and assume that all messages can be sent within one frame. If messages can be split into multiple slots, we refer the interested reader to [79] and [74].

#### 2.4.2.1 End-to-End Application Latency

For time-critical applications, a mapping can only be considered feasible if the worst-case end-to-end application latency $\mathcal{L}_{wc}(\alpha, \rho)$ does not exceed a given deadline $\delta$ such that the following constraint has to be met:

$$\mathcal{L}_{wc}(\alpha, \rho) \leq \delta. \tag{2.4}$$

A distributed application's end-to-end latency is influenced by executing computational tasks $t$ and sending messages $m$ between the tasks. However, task execution and message transmission times will differ between each iteration. This highly depends on the paths in a program and interference caused by other applications executed concurrently in the system. The interference time is mainly influenced by the scheduling and admission algorithms used for the CPU, the network infrastructure, and other shared resources. The worst-case end-to-end execution latency $\mathcal{L}_{wc}(\alpha, \rho)$ is determined by the critical path as

$$\mathcal{L}_{wc}(\alpha, \rho) = \max_{\forall path \in paths(G_A(V,E))} PL(path, \alpha, \rho), \tag{2.5}$$

where the critical path is the path through an application with the highest aggregated latency. The path latency $PL(path, \alpha, \rho)$ itself can be calculated as

$$PL(path, \alpha, \rho) = \sum_{\forall t \in path \cap T} TL(t, \alpha(t)) + \sum_{\forall m \in path \cap M} CL(m, \rho(m)), \tag{2.6}$$

by summing up all task latencies $TL(t, \alpha(t))$ and communication latencies $CL(m, \rho(m))$ of tasks and messages which lie in this path. To predictably calculate these worst-case task latencies $TL(t, \alpha(t))$ and worst-case communication latencies $CL(m, \rho(m))$, it is required to analytically determine an upper bound. To reduce complexity, composability is required to ensure that applications have only a bounded effect on each other. Well-known scheduling approaches such as RR or TDM temporally isolate task execution or message transmission on a resource.

Figure 2.7 presents an exemplary mapping of a non-critical application with annotated worst-case task and communication latencies. The path latencies for the two paths $p_0 = (t_0 - m_0 - t_1)$ and $p_1 = (t_0 - m_1 - t_2)$ in the application graph can be calculated as $PL(p_0, \alpha, \rho) = 30\ ms$ and $PL(p_1, \alpha, \rho) = 35\ ms$. The critical path $p_1$ leads to a worst-case end-to-end application latency of $\mathcal{L}_{wc}(\alpha, \rho) = 35\ ms$. With a deadline of $\delta = 40\ ms$, Equation 2.4 would be still fulfilled.

#### 2.4.2.2 Composable Scheduling

Using a weighted RR scheme, the processing time of a CPU can be partitioned into service intervals $SI$ with a fixed time duration $\tau_{SI}$. In a RR schedule, each task is executed for the same amount of time and is put at the end of a waiting queue after the execution such that every task gets an equal amount of time for execution. An additional weight would then determine how many service intervals a task is executed, but tasks would still be put back at the end of the waiting queue after the execution. This approach has the slight disadvantage that the interference time changes with the number of tasks scheduled on the CPU. Adding another task to a schedule would increase the waiting time for the tasks already in the schedule. Therefore, an analysis of the worst-case end-to-end latency would be only possible if the binding of all applications is known. However, we would like to allow a partitioned analysis, where applications can be mapped and analyzed independently. Instead, defining a maximum available number of service intervals $SI_{max}$ beforehand would provide a guaranteed bound for the time a task would spend in the waiting queue. Using a limited number of time slots is

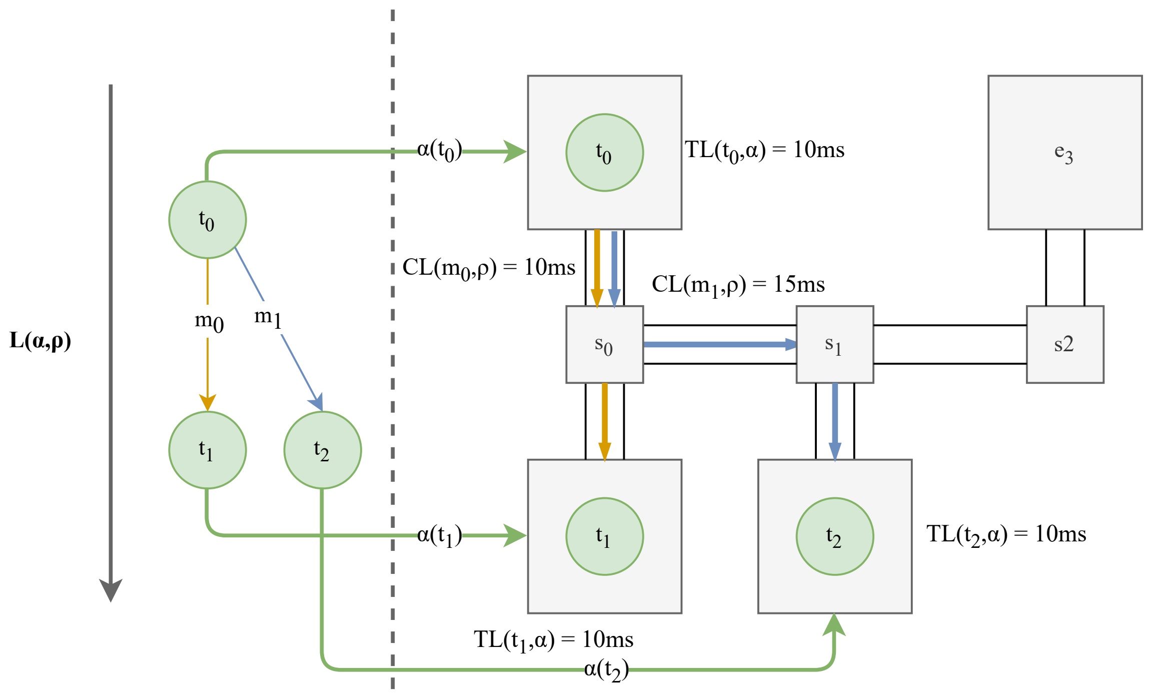


**Figure 2.7:** Exemplary mapping of a non-critical application onto a hardware architecture consisting of four ECUs $e_0$, $e_1$, $e_2$, and $e_3$, and three switches $s_0$, $s_1$, and $s_2$. The green arrows indicate the binding of a task to an ECU. The message routings are marked in the color of the corresponding message in the application graph. There are two paths $p_0 = (t_0 - m_0 - t_1)$ and $p_1 = (t_0 - m_1 - t_2)$ in the application graph. Taking the task and communication latencies from the figure, the path latencies can be calculated as $PL(p_0, \alpha, \rho) = 30\ ms$ an $PL(p_1, \alpha, \rho) = 35\ ms$. With $p1$ as the critical path, the worst-case end-to-end application latency can be calculated as $\mathcal{L}_{wc}(\alpha, \rho) = 35\ ms$. With a deadline of $\delta = 40\ ms$, the constraint in Equation 2.4 would be met.

the same principle as in TDM scheduling with an Asynchronous Time Division (ATD), where messages would be guaranteed several time slots but not a fixed time slot, as [74] has used for communication scheduling.

**Task Scheduling** In general, the task latency $TL(t, \alpha(t))$ consists of the actual task execution time $TL_{exec}(t, \alpha(t))$ and the task interference time $TL_{inter}(t, \alpha(t))$, which a task spends waiting e.g. due to scheduling:

$$TL(t, \alpha(t)) = TL_{exec}(t, \alpha(t)) + TL_{inter}(t, \alpha(t)). \tag{2.7}$$

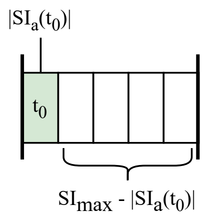
Thus, the worst-case task execution time without interference $TL_{exec}(t, \alpha(t))$ is a multiple of the service interval time $\tau_{SI}$ such that we can calculate it using the WCET $W(t, \alpha(t))$ as

$$TL_{exec}(t, \alpha(t)) = \lceil \frac{W(t, \alpha(t))}{\tau_{SI}} \rceil \cdot \tau_{SI}. \tag{2.8}$$

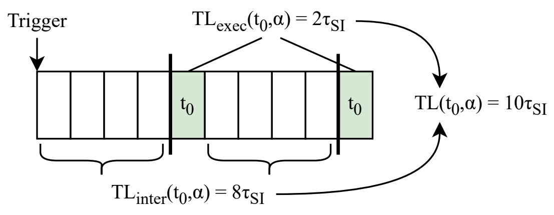
Given a number of service intervals $SI_a(t)$ that are allocated for a task $t$ and a defined maximum number of service intervals $SI_{max}$ we can determine the worst-case interference time $TL_{inter}(t, \alpha(t))$ as

$$TL_{inter}(t, \alpha(t)) = \lceil \frac{W(t, \alpha(t))}{|SI_a(t)| \cdot \tau_{SI}} \rceil \cdot (SI_{max} - |SI_a(t)|) \cdot \tau_{SI}. \tag{2.9}$$

Here, $SI_{max} - |SI_a(t)|$ reflects the number of service intervals a task would have to wait until it is executed again. The first factor represents how often the task would have to wait until its turn in the worst case. An exemplary task schedule with the derivation of the worst-case task latency $TL(t, \alpha(t))$ is presented in Figure 2.8.



**(a)** Task schedule with allocation.


Trigger
$TL_{exec}(t_0,\alpha) = 2\tau_{SI}$
$t_0$
$t_0$
$TL(t_0,\alpha) = 10\tau_{SI}$
$TL_{inter}(t_0,\alpha) = 8\tau_{SI}$

**(b)** Derivation of the worst-case task latency.

**Figure 2.8:** Example of a task schedule with a maximum amount of allocatable service intervals of $SI_{max} = 5$. One service interval $|SI_a| = 1$ is allocated for the task $t_0$. With a WCET of $TL_{exec}(t_0, \alpha) = W(t_0, \alpha) = 2\tau_{SI}$, two full execution cycles are required in the worst case for the task to finish execution. With the corresponding worst-case task interference time of $TL_{inter}(t_0, \alpha) = 8\tau_{SI}$, the worst-case task latency can be calculated as $TL_{inter}(t_0, \alpha) = 10\tau_{SI}$.

**Message Scheduling** We assume that messages are sent over ethernet links instead of the work of [74], where smaller links connect multiple processing elements on a NoC architecture. For the worst-case communication latency, we can proceed likewise as for the task scheduling by using an ATD schedule to calculate the worst-case communication latency $CL(m, \rho)$ with the worst-case message transmission time $CL_{trans}(m, \rho)$ and the worst-case communication interference time $CL_{inter}(m, \rho)$:

$$CL(m, \rho(m)) = CL_{trans}(m, \rho(m)) + CL_{inter}(m, \rho(m)) \tag{2.10}$$

For the message scheduling, we use the notation $SL$ to describe the time frame of a slot interval. We design the system so that one ethernet frame with a maximum frame size of 1518 bytes can be sent in one time slot. We assume that message sizes do not exceed the Maximum Transmission Unit (MTU) of an ethernet frame such that only one slot has to be allocated per message. Using this, we can calculate the transmission time of a message over one ethernet link $CL_{trans}(m, l)$ as

$$CL_{trans}(m, l) = \tau_{SL}. \tag{2.11}$$

To calculate the interference time of a message over one link $CL_{inter}(m, l)$, we assume that a maximum number of slots $SL_{max}$ is defined. As the transmission of the message requires only one slot, a message has to wait one transmission round in the worst case:

$$CL_{inter}(m, l) = (SL_{max} - 1) \cdot \tau_{SL}. \tag{2.12}$$

Combining these two worst-case latencies, we can calculate the worst-case communication latency $CL_(m, l)$ of a message $m$ over one link $l$ as

$$CL_(m, l) = CL_{trans}(m, l) + CL_{inter}(m, l) = (SL_{max} - 1) \cdot SL + SL = SL_{max} \cdot \tau_{SL}. \tag{2.13}$$

This approach allows us to analyze the worst-case communication latency over each link individually as

$$CL(m, \rho(m)) = \sum_{\forall l \in \rho(m)} CL(m, l) \tag{2.14}$$

Under the assumption that all links in the system are designed equally, the communication latency only depends on the number of links that the message $m$ is passing on its route $\rho$, further denoted as $hops(\rho(m))$, which allows us to further simplify the formula to

$$CL(m, \rho(m)) = hops(\rho(m)) \cdot SL_{max} \cdot \tau_{SL}. \tag{2.15}$$

Our formulas are based on the assumption that a message requires exactly one slot to be transmitted. If the system should be designed granularly such that a message could require multiple slots for transmission, we refer the interested reader to [79] and [74].

## 2.4.3 Performance Analysis of Gracefully Degrading Systems

Using a state-of-the-art analysis presented in Subsection 2.4.2 it is possible to analyze whether a configuration of an application consisting of active tasks is meeting a deadline $\delta$. However, we need to ensure that critical applications can still continue full operation after any ECU failure without violating Equation 2.4. Here, we add passive tasks as a backup solution that are started once the active task is affected by a failure. Furthermore, we are enabling a gracefully degrading behavior such that resources allocated by non-critical applications can be used by critical applications if required. The advantage of using our passive backup solution compared to active redundancy is that no overhead is added regarding the required computational power during operation.

Theoretically, searching for a new valid configuration that satisfies the timing constraint would be possible after an ECU failure occurred. However, this approach would have multiple disadvantages accompanied by highly unpredictable behaviors. First, it can not be guaranteed that sufficient resources are available for task computations and message transmissions after a failure, even if a degradation approach is used. The ECU failure reduces the system-wide resource pool such that not every application can find sufficient resources. Even if the system was over-designed, the subsequent failure would increase the uncertainty further. Second, even if sufficient resources were still available, it is uncertain if a mapping could be found that would satisfy the timing constraint. Third, finding a valid solution could take a lot of time. Even if a valid solution was available, finding one could take an unknown amount of time. Although the unavailability of an application might be tolerable for a certain amount of time (FTTI) during a failover, it would not be predictable how long it would take to find a solution, and most likely it would not be found in time. Fourth, it would be unpredictable which non-critical applications would be shut down due to degradation as this would be only decided after the occurrence of the failure. Most importantly, it is uncertain if a solution can be found in time yet if one exists, which is unacceptable for safety-critical applications. Furthermore, it would be at least desirable to also allow a more predictable degradation behavior of non-critical applications.

Therefore, improving uncertainty and achieving a more predictable behavior is necessary. To bypass having to find new solutions after an ECU failure, we have to ensure that a suitable backup solution is already available for any ECU failure in the system. To ensure this, we add redundant tasks such that there is at least one instance of each task available after an ECU failure that has sufficient resources to continue operation and to communicate with other tasks. Here, we present our performance analysis of gracefully degrading systems to verify that these backup solutions always meet timing constraints in Subsection 2.4.3.1. This solution allows predictable system behavior as it is known as a valid backup solution that meets resource and timing constraints before any failure is available and allows a quick switch to this backup solution. Last, we present our composable scheduling of gracefully degrading systems in Subsection 2.4.3.2, allowing us to independently derive worst-case latencies while enabling a gracefully degrading system behavior. This solution also allows us to predict in which failure scenario a non-critical application will be shut down due to graceful degradation.

#### 2.4.3.1 End-to-End Application Latency

A valid binding needs to ensure that the deadline $\delta$ is not only met by the active part of the application but also in case any backup solutions have to be used in a failure scenario. Activating passive task instances could lead to a new critical path in the application, which might not meet the deadline $\delta$. The worst-case end-to-end application latency $\mathcal{L}_{wc}(\alpha, \beta, \rho, \sigma)$ considers not only the worst-case end-to-end application latency $\mathcal{L}_{wc}(\alpha, \rho)$ of currently active task instances but also of any possible future configurations, where passive task instances are activated. To achieve a fail-operational behavior, a valid mapping of a safety-critical application has to fulfill the following constraints:

$$\mathcal{L}_{wc}(\alpha, \beta, \rho, \sigma) \leq \delta. \tag{2.16}$$

As any activation of a passive task instance could potentially lead to a violation of this constraint, any path through the instance graph $G_B(V, E)$, including passive task instances $t \in T_b$ and backup messages $m \in M_b$, has to be considered. Therefore, we can define the worst-case end-to-end application latency $\mathcal{L}_{wc}(\alpha, \beta, \rho, \sigma)$ as the critical path through the instance graph $G_B(V, E)$, which also includes all possible backup solutions:

$$\mathcal{L}_{wc}(\alpha, \beta, \rho, \sigma) = \max_{\forall path \in paths(G_B(V,E))} PL(path, \alpha, \beta, \rho, \sigma), \tag{2.17}$$

The path latency $PL(path, \alpha, \beta, \rho, \sigma)$ depends not only on the worst-case latency of active task instances $TL(t, \alpha(t))$ and the worst-case communication latency of active messages $CL(m, \rho(m))$, but also on the potential worst-case latency of passive task instances $TL(t, \beta(t))$ and the possible worst-case latency of backup messages $CL(m, \sigma(m))$, such that it can be calculated as

$$\begin{aligned} PL(path, \alpha, \beta, \rho, \sigma) = &\sum_{\forall t \in path \cap T_a} TL(t, \alpha(t)) + \sum_{\forall t \in path \cap T_b} TL(t, \beta(t)) + \\ &\sum_{\forall m \in path \cap M_a} CL(m, \rho(m)) + \sum_{\forall m \in path \cap M_b} CL(m, \sigma(m)). \end{aligned} \tag{2.18}$$

can From this formula, it can be observed that it is also necessary to find a bound on the potential worst-case latency of passive task instances $TL(t, \beta(t))$ and the potential worst-case latency of backup messages $CL(m, \sigma(m))$.

For illustration, let us assume that a deadline $\delta = 35\ ms$ is given for the safety-critical application as presented in Figure 2.9. When disregarding passive task and message instances and using the annotated worst-case latencies from the figure, state-of-the-art performance analysis as presented in [74] would conclude a worst-case end-to-end application latency of $\mathcal{L}_{wc}(\alpha, \rho) = 30\ ms$ which would meet the deadline $\delta = 35\ ms$. However, in a failure scenario where $t_{0,a}$ is affected by a failure and $t_{0,b}$ activated, the deadline can no longer be met, leading to a configuration that violates the timing constraint. By contrast, our performance analysis takes passive task and messages instances into account and, therefore, can identify the highlighted critical path $p_2 = (t_{0,b} - m_{0,ba} - t_{1,a})$ with a worst-case path latency of $PL(p_2, \alpha, \beta, \rho, \sigma) = 40\ ms$. A violation of the timing constraint with a worst-case end-to-end application latency of $\mathcal{L}_{wc}(\alpha, \beta, \rho, \sigma) = 40\ ms$ and the deadline $\delta = 35\ ms$ is visible. Here, our performance analysis can quickly evaluate different configurations and help design automation algorithms to find a valid configuration.

#### 2.4.3.2 Composable Scheduling of Gracefully Degrading Systems

We extend state-of-the-art scheduling, such as presented in [74] by introducing the concept of graceful degradation. For our analysis and experiments, we apply graceful degra-

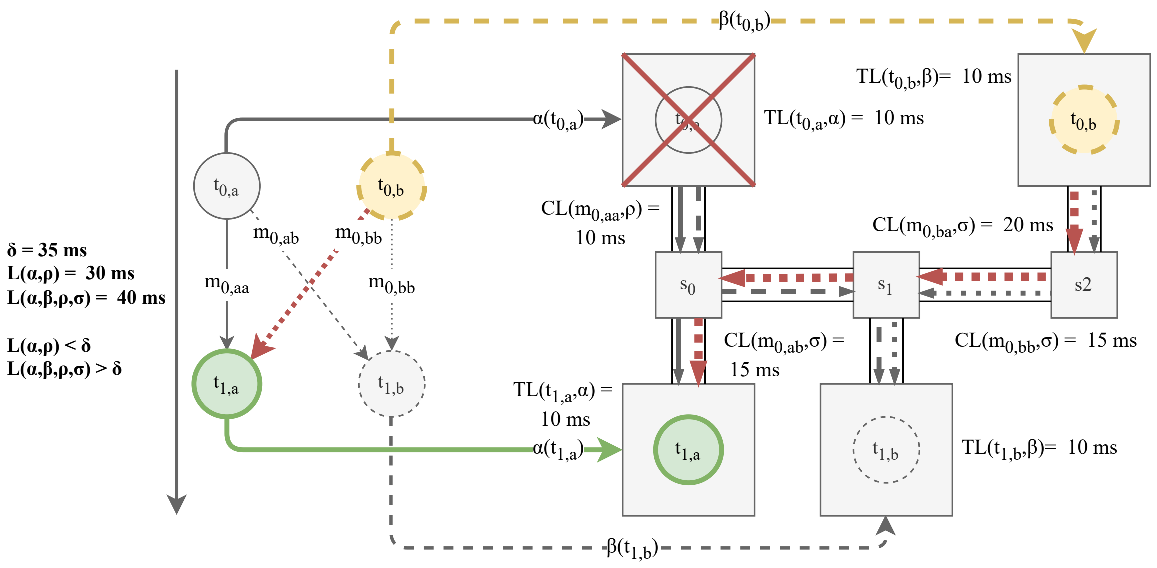
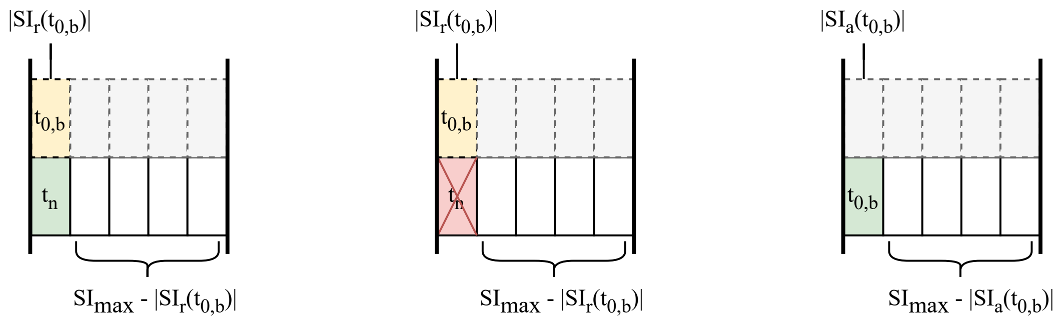


**Figure 2.9:** Exemplary mapping of a safety-critical application onto a system architecture with annotated worst-case latencies. When using state-of-the-art performance analysis and disregarding the passive task and message instances, the worst-case end-to-end application latency $\mathcal{L}_{wc}(\alpha, \rho) = 30\ ms$ would meet the deadline $\delta = 35\ ms$. However, in a failure scenario where $t_{0,a}$ was affected by a failure and $t_{0,b}$ activated, the deadline could no longer be met, leading to a configuration that violates the timing constraint. Our performance analysis takes the highlighted critical path $p_2 = (t_{0,b} - m_{0,ba} - t_{1,a})$ with a worst-case path latency of $PL(p_2, \alpha, \beta, \rho, \sigma) = 40\ ms$ into account leading to a worst-case end-to-end application latency of $\mathcal{L}_{wc}(\alpha, \beta, \rho, \sigma) = 40\ ms$. A timing constraint violation with the deadline $\delta = 35\ ms$ can be identified here. Therefore, our performance analysis can quickly evaluate different configurations and help design automation algorithms to find a valid configuration.

dation to CPU resources, although the concept can also be used to link resources. Applied to our composable schedules, service intervals can be allocated and reserved for a task. A reservation indicates that the corresponding service interval is currently not in use but might be used and turned into an allocation once the backup instance is used. Service intervals that can be reserved are empty service intervals that have not been allocated yet or service intervals already allocated by non-critical applications. The allocation of slots works the other way; non-critical applications can allocate service intervals that are free or that are already reserved by critical applications. On the other hand, critical applications can only allocate completely free service intervals. The graceful degradation approach is applied if a non-critical application allocates a service interval and is also reserved by a critical application. In case there is a failure in the system and critical passive task instances have to be started to mitigate a failure, the reservation of the resources will be turned into an active allocation, and any non-critical tasks which formerly held an allocation of the corresponding slots are shut down. Depending on the application, it could then be decided if the degraded non-critical application keeps running in a degraded mode or is completely shut down.



**(a)** Task schedule with reservation and allocation. **(b)** Non-critical task $t_n$ being shut down. **(c)** Reservation turned into an allocation.

**Figure 2.10:** Example of a task schedule with a maximum amount of allocatable (lower) and reservable (upper) service intervals of $SI_{max} = 5$. One service interval $|SI_r| = 1$ is reserved for the critical task instance $t_{0,b}$. The same service interval is allocated by the non-critical task instance $t_n$. In a failure scenario where $t_{0,b}$ has to be activated, $t_n$ is shut down in the first step. Afterward, $t_{0,b}$ takes over the allocation of the service interval.

For the latency analysis of the backup solutions, it is not relevant whether a reserved slot is also allocated and graceful degradation is applied as the reservation will be turned into an allocation. This procedure allows us to predict before the occurrence of any failure if a valid backup solution can be found but also allows a predictable behavior of the graceful degradation approach. Furthermore, we can reuse our analytical formulas from Subsection 2.4.2.2 to calculate $TL(t, \beta(t))$ based on the number of reserved service intervals $SI_r(t)$ and the binding of the passive task instance $\beta(t)$:

$$TL(t, \beta(t)) = TL_{exec}(t, \beta(t)) + TL_{inter}(t, \beta(t)), \tag{2.19}$$

$$TL_{exec}(t, \beta(t)) = \lceil \frac{W(t, \beta(t))}{\tau_{SI}} \rceil \cdot \tau_{SI}, \tag{2.20}$$

$$TL_{inter}(t, \beta(t)) = \lceil \frac{W(t, \beta(t))}{|SI_r(t)| \cdot \tau_{SI}} \rceil \cdot (SI_{max} - |SI_r(t)|) \cdot \tau_{SI}. \tag{2.21}$$

To calculate the worst-case communication latency $CL(m, \sigma(m))$ of backup message instances, under the same assumptions as in Subsection 2.4.2.2, we can reuse Equation 2.15 based on the routing of the backup message instances $\sigma$:

$$CL(m, \sigma) = hops(\sigma(m)) \cdot SL_{max} \cdot \tau_{SL}. \tag{2.22}$$

Figure 2.10 presents an exemplary task schedule with reservations and allocations. The upper service intervals indicate a reservation, while the lower service intervals indicate an allocation of the same service interval. In this example, the first service interval is allocated by the non-critical task $t_n$ and reserved by the critical task instance $t_{0,b}$. In a failure scenario where the task instance $t_{0,b}$ is activated to serve as a backup solution, the non-critical $t_n$ first loses its allocation and, thus, is degraded. Afterward, the reservation of the task instance $t_{0,b}$ is turned into an active allocation such that this resource can now be used exclusively by the critical task instance.

The decision of which service interval should be allocated or reserved is not straightforward. When a completely free service interval is always taken first, then the schedule might run out of service intervals earlier, so insufficient resources might be left for other tasks. On the other hand, when trying to overlap service intervals as much as possible but with more than sufficient service intervals being available for all tasks, an avoidable degradation might occur in a failure scenario. We present and discuss three different allocation and reservation strategies in Subsection 2.4.4.4 and evaluate them in Subsection 2.4.5. Before that, we introduce our agent-based mapping approach and the components required to implement such a gracefully degrading behavior at run-time in the following section.

### 2.4.4 Agents - Finding Feasible Solutions at Run-Time

We use a dynamic agent-based approach to perform mapping and constraint checking at run-time. In the following, we describe the components involved in the mapping process and the algorithms used by the agents. Each agent controls one task instance and is responsible for allocating resources and meeting constraints. Together, the agents enable a decentralized control without a single point of failure. This concerns both active and passive task instances. The agents can communicate in a predefined order with each other to manage the mapping process of all task instances of an application. To dynamically activate, deactivate, and move tasks on the platform at run-time, we implemented a middleware based on SOME/IP [36]. This middleware includes a decentralized service discovery to dynamically find services in the system and a publish/subscribe scheme to publish and subscribe to events.

We assume that most system resources are available for the search and that the mapping approach is performed while the car is not actively used. Once a stable mapping is found, the agents do not perform any action and, thus, do not require any additional resources. As backup mappings are already found and readily available, we follow our central hypothesis. When the failure occurs, a safe state can be reached with a minimum amount of communication and computation. After a failover, the fail-operational behavior has to be re-established to ensure safe travel, as further failures could lead to potential hazards. A re-mapping of the applications has to be performed during a safe halt.

Our system to find feasible mappings at run-time consists mainly of agents and resource managers. Resource managers manage the resources associated with an ECU or switch. The resource managers handle the allocation or reservation requests and assign the corresponding service intervals or slots. By assigning a slot, a resource manager implicitly decides how the system will be degraded in a failure scenario and which task instances will lose resources. Therefore, changing the algorithm for assigning the slots can impact the degradation behavior and the success rate of finding a mapping. An agent is responsible for finding a suitable mapping for its task that meets all mandatory constraints by allocating and reserving resources at the resource manager or moving to another ECU. Furthermore, each agent verifies if an application-wide constraint, such as the end-to-end application latency, is still met. As a sequential mapping order is required to fulfill or verify these constraints, agents can communicate with each other to synchronize the mapping flow.

In the following, we first provide an overview and formalize the constraints that have to be solved to consider a given mapping as a valid solution in Subsection 2.4.4.1. We then present a solution for evaluating the timing constraints at run-time using our performance analysis in Subsection 2.4.4.2. Afterward, we present the mapping process of an application using agents in Subsection 2.4.4.3. Here, we first detail in which order the agents find a mapping for their task to meet all constraints. Afterwards, we present a code listing and describe how the search space is searched for a valid solution by the agents, which solves all constraints using a backtracking algorithm. Furthermore, we describe the rAfterwardole of resource managers in Subsection 2.4.4.4 responsible for performing allocations and reservations by choosing the corresponding service intervals. We also present our allocation and reservation strategies that can be used by a resource manager, which has a direct effect on how the system will be degraded. Last, we describe the recovery and reconfiguration process with the immediate failure reaction to ensure a safe fail-operational behavior in Subsection 2.4.4.5.

#### 2.4.4.1 Constraints

The following constraints must be respected to consider any found mapping a valid solution. Agents or resource managers respect these constraints at run-time as described below.

C.1 The application-wide worst-case end-to-end application latency must meet the deadline $\delta$:

$$\mathcal{L}_{wc}(\alpha, \beta, \rho, \sigma) \leq \delta \tag{2.23}$$

C.2 An active task instance $t_a$ and its corresponding passive task instance $t_b$ may not be mapped onto the same ECU as this would contradict our fail-operational goal:

$$\alpha(t_a) \neq \beta(t_b). \tag{2.24}$$

C.3 The number of allocated service intervals $N_{SI,a}(e)$, reserved service intervals $N_{SI,r}(e)$ and service intervals with both allocation and reservation $N_{SI,ar}(e)$ of an ECU $e$ must not exceed the number of maximum available service intervals $SI_{max}$:

$$N_{SI,a}(e) + N_{SI,r}(e) + N_{SI,ar}(e) \leq SI_{max} \tag{2.25}$$

C.4 The number of allocated slots $N_{SL,a}(l)$, reserved slots $N_{SL,r}(l)$ of a link $l$ must not exceed the number of maximum available slots $SL_{max}$:

$$N_{SL,a}(l) + N_{SL,r}(l) \leq SL_{max} \tag{2.26}$$

C.5 Service intervals that are allocated by a critical task $t$ must not be allocated or reserved by any other task:

$$\forall SI \in SI_a(t) \wedge \forall t' \in T : SI \notin SI_a(t') \wedge SI \notin SI_r(t') \tag{2.27}$$

C.6 Service intervals which are reserved by a critical task $t$ must not be allocated or reserved by another critical task but may be allocated by a non-critical task:

$$\forall SI \in SI_r(t) \wedge \forall t' \in T_c : SI \notin SI_a(t') \wedge SI \notin SI_r(t') \tag{2.28}$$

C.7 Service intervals which are allocated by a non-critical task $t$ must not be allocated by any other task but may be reserved by a critical task:

$$\forall SI \in SI_a(t) \wedge \forall t' \in T : SI \notin SI_a(t') \tag{2.29}$$

C.8 Slots which are allocated or reserved by a message $m$ must not be allocated or reserved by another message:

$$\forall SL \in SL_a(m) \wedge \forall m' \in M : SL \notin SL_a(m') \wedge SL \notin SL_r(m') \tag{2.30}$$

$$\forall SL \in SL_r(m) \wedge \forall m' \in M : SL \notin SL_a(m') \wedge SL \notin SL_r(m') \tag{2.31}$$

Subsection 2.4.4.2 describes how solutions that respect Constraint C.1 can be solved at run-time by splitting the validation problem into smaller sub-problems the agents solve. Constraint C.2 is respected by constraining the mapping order of the agents and by limiting the search space accordingly, as described in Subsection 2.4.4.3. Constraints C.3 to C.8 are implicitly respected by the resource manager when assigning service intervals and slots to task and message instances as described in Subsection 2.4.4.4.

#### 2.4.4.2 Run-Time Timing Constraint Solving

As described in Subsection 2.4.3.1, the application graph's critical path must be identified to verify whether Constraint C.1 can be met. Instead of verifying the timing constraint once all task instances are mapped, we continuously evaluate whether the application meets the timing constraint with the task instances mapped thus far. This is done at run-time as the mapping of task instances and their predecessors is not known at design time. We split the bigger problem of verifying every single path into smaller sub-problems solved by the agents such that the constraint is continuously being verified, and the search can be interrupted earlier if the mapping process is running into a dead-end.

As only the critical path is of interest, each agent calculates and stores the worst-case path latency to its task $\mathcal{L}_{wc}(t, \alpha, \beta, \rho, \sigma)$ as

$$\mathcal{L}_{wc}(t, \alpha, \beta, \rho, \sigma) = \max_{\forall t_p \in pred(t)} (\mathcal{L}_{wc}(t_p, \alpha, \beta, \rho, \sigma) + CL(m_{t_p - t}, \rho, \sigma)) + TL(t, \alpha(t), \beta(t)). \tag{2.32}$$

Here, an agent requires only the worst-case path latencies $\mathcal{L}_{wc}(t_p, \alpha, \beta, \rho, \sigma)$ of its predecessor tasks $t_p$ and the corresponding worst-case communication latencies to its own task $CL(m_{t_p - t}, \rho, \sigma))$ together with the worst-case task latency $TL(t, \alpha(t), \beta(t))$. The maximum of all paths from preceding tasks $t_p$ to this task $t$ is then the worst-case path latency to task $\mathcal{L}_{wc}(t, \alpha, \beta, \rho, \sigma)$. Accordingly, instead of verifying Constraint C.1 once all tasks are mapped, each agent can verify the following constraint on its own:

$$\mathcal{L}_{wc}(t, \alpha, \beta, \rho, \sigma) \leq \delta \tag{2.33}$$

This has the advantage that if multiple sink tasks exist in an application graph, each agent can verify the constraint independently. Furthermore, suppose any tasks can not meet the constraint during the mapping process. In that case, the process can be stopped, and a backtracking algorithm can be applied, saving the agents from evaluating invalid solutions.

#### 2.4.4.3 Agent

In our work, each task instance is assigned to an agent responsible for finding a mapping by allocating and reserving resources at resource managers and coordinating the mapping process with other agents. In the following, we present a pre-defined mapping order in which the agents of an application find a mapping. Afterward, we describe the search space exploration of an agent using a code listing. Last, we present the backtracking approach agents require to explore different solutions.

**Mapping Order** For the mapping process of the tasks onto the ECUs, a predefined mapping flow is required to ensure that constraints are met and only valid mappings are generated. Theoretically, it would be preferable to map all tasks in parallel. However, there are limitations to the level of parallelism due to constraints and their verification at run-time. To adhere to Constraint C.2, a sequential mapping flow between active and passive task instances is required to avoid mapping both task instances onto the same ECU. Therefore, we define that active task instances must be mapped before passive task instances. Second, to allocate and reserve resources for communication, the routing from a predecessor task to its successor has to be known. We decided that the succeeding tasks allocate and reserve the resources for all incoming messages. Consequently, a task instance can only be mapped onto an ECU once all preceding task instances are mapped. This sequential mapping order is also required to perform the run-time constraint solving described in Subsection 2.4.4.2.

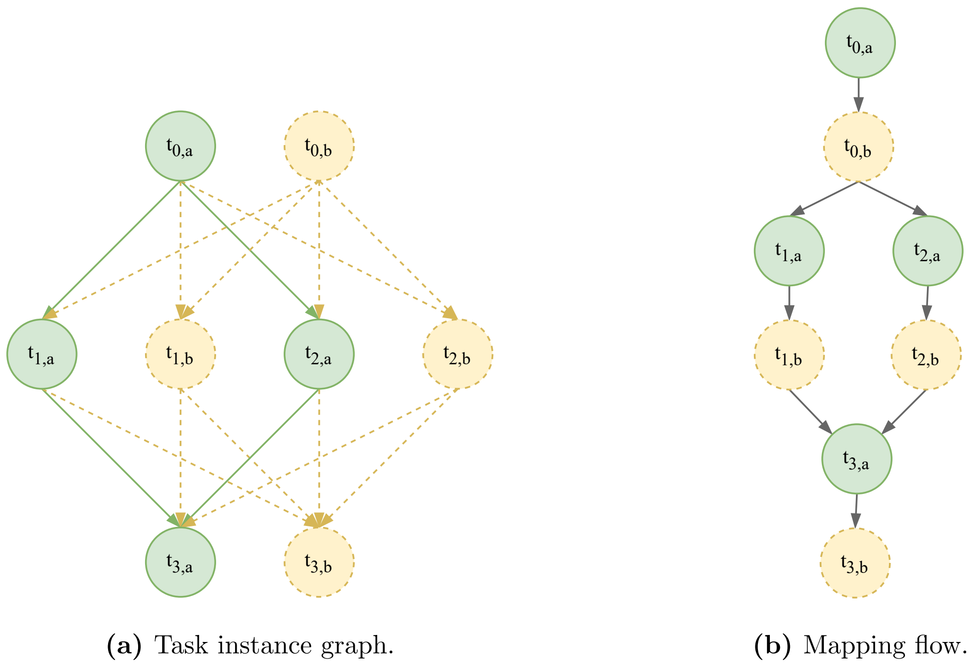


**Figure 2.11:** An exemplary task instance graph with the corresponding mapping flow. Passive task instances are mapped after active task instances. Active task instances have to wait until the passive task instances of their predecessors are mapped. As the task instances of $t1$ and $t2$ do not depend on each other, they can be mapped in parallel. However, the active task instance of $t3$ has to wait for both branches to finish to continue with the mapping process.

Figure 2.11 shows the mapping flow of an exemplary task instance graph consisting of four tasks. The mapping process starts with the active task instance of the source task $t_{0,a}$. All other task instances are initially waiting until they receive a message from all their mapping predecessors. Once the active task instance has found a mapping, it will inform its passive counterpart $t_{0,b}$ that it found a valid mapping. The passive task instance $t_{0,b}$ that has been waiting then finds a mapping itself. It ensures that the ECU to which the active task instance is mapped is removed from the search space of the passive task instance. Afterward, the passive task instance informs all active task instances of directly succeeding tasks. In the example $t_{0,b}$ informs $t_{1,a}$ and $t_{2,a}$ about a successful mapping. Active task instances might have to wait not only for one but multiple passive task instances of their preceding tasks depending on the dependencies in the application graph. In the figure $t_{3,a}$ has to wait for both $t_{1,b}$ and $t_{2,b}$ to finish. By contrast, neighboring tasks with shared predecessors can search for a mapping in parallel as they have no direct dependency and open a new branch, as is the case for the task instances of $t1$ and $t2$ in the example. Any task instances belonging to different branches can also work in parallel. Therefore, the level of parallelism in the mapping search is bound by the number of branches or the width of the application graph. Tasks with predecessors in multiple branches reduce the level of parallelism and represent a bottleneck.

**Agent Mapping** Once an agent turns to find a mapping, it is its responsibility to evaluate the mapping options and ensure that resources are allocated and reserved accordingly. Listing 2.1 shows the pseudo-code for the mapping process of an agent. Each agent starts in the Wait() function (Line 1), waiting until all predecessors send a signal that they found a mapping and transmitting their worst-case latencies and mappings. Afterward, the agent becomes active and is searching for a mapping. At first, the search space is set up consisting of all available ECUs in the system (Line 6). Then, in case a passive task instance $t_b$ is being mapped, the ECU to which the corresponding active task instance $t_a$ is mapped is removed from the search space $B(t_b)$ to adhere to Constraint C.2 (Lines 7 and 23):

$$B(t_b) = \{e \in E | e \neq \alpha(t_a)\} \tag{2.34}$$

To find the most suitable solution, the different options for mapping the task are evaluated (Lines 8 and 28). Here, the worst-case path latency $\mathcal{L}_{wc}(t, e)$ for each available ECU $e \in E$ is calculated. Afterward, the search space can be filtered for invalid solutions that do not adhere to Equation 2.4.4.2 and, therefore, would violate Constraint C.1 (Lines 9 and 34). Using a heuristic, the possible solutions are sorted and calculated by an increasing worst-case latency (Line 10). The agent then tries to allocate resources for the most fitting solution by sending allocation and reservation requests to resource managers (Lines 11 and 40). Here, service intervals have to be allocated or reserved for the executing tasks, and slots allocated or reserved for all incoming messages on the links on the corresponding routing paths. The resource managers request an agent to give or reserve a certain amount of service intervals for CPU resources or slots for a link. The resource manager then executes the allocation or reservation and answers the agent whether the allocation or reservation could be performed successfully or not. The resource manager also implicitly decides how the system will be degraded by choosing the corresponding service intervals for the agent. However, this behavior is entirely transparent to the agent, and it does not know whether the activation of its passive task instance will lead to a degradation of another task and vice versa. Multiple strategies of the resource manager for allocating and reserving resources are described in Subsection 2.4.4.4. Suppose all resources could be successfully allocated and reserved. In that case, a valid solution has been found such that the agent can update the binding for its task, move to the corresponding ECU, and inform all succeeding agents about its success (Lines 18-21).

```
1  Wait():
2      $\alpha, \beta, \mathcal{L}_{wc}(t_p) \leftarrow$ WaitForPredecessors()
3      FindMapping($\alpha, \beta, \mathcal{L}_{wc}(t_p)$)
4
5  FindMapping($\alpha, \beta$):
    $E \leftarrow$ GetSearchSpace()
    $E \leftarrow$ RemoveActiveBindingFromSearchSpace($\alpha$, $E$)
    $\mathcal{L}_{wc} \leftarrow$ EvaluateSearchSpace($\alpha, \beta, \mathcal{L}_{wc}(t_p), E$)
    $E \leftarrow$ FilterSearchSpace($\mathcal{L}_{wc}, E$)
    $E \leftarrow$ SortSearchSpaceByLatency($\mathcal{L}_{wc}, E$)
    $\epsilon \leftarrow$ TrySolutions($E$)

    if $\epsilon == \varnothing$
        for $t_p$ in pred(t):
            $t_p$.Backtrack()
        Wait()
    else:
        UpdateBindings($\alpha, \beta, \epsilon$)
        $\mathcal{L}_{wc}(t) \leftarrow \mathcal{L}_{wc}(t, \epsilon)$
        MoveToECU($\epsilon$)
        InformSuccessors($\alpha, \beta, \mathcal{L}_{wc}(t)$)

RemoveActiveBindingFromSearchSpace($E$):
    if $t \in T_b$:
        $E$.remove($\alpha(t)$)
    return $E$

EvaluateSearchSpace($\alpha, \beta, E$):
    for $\epsilon \in E$:
        $\rho, \sigma \leftarrow$ GetRoutings($\alpha, \beta, \epsilon$)
        $\mathcal{L}_{wc}(t, \epsilon) = \max_{\forall t_p \in pred(t)} (\mathcal{L}_{wc}(t_p) + CL(m_{t_p - t}, \rho, \sigma)) + TL(t, \epsilon)$
    return $\mathcal{L}_{wc}, E$

FilterSearchSpace($\mathcal{L}_{wc}, E$):
    for $\epsilon \in E$:
        if $\mathcal{L}_{wc}(t, \epsilon) > \delta$:
            $E$.remove($\epsilon$)
    return $E$

TrySolutions($E$):
    while $\epsilon \in E$ and not $\zeta$:
        $E$.remove($\epsilon$)
        $\zeta \leftarrow$ AllocateAndReserveResources($\epsilon$):

    if not $\zeta$:
        return $\varnothing$
    else:
        return $\epsilon$

Backtrack():
    FreeResources()
    InvalidateSuccessors()
    FindMapping()

InvalidateSuccessors():
    for $t_s$ in succ(t):
        $t_s$.Invalidate()
```

```
58
59  Invalidate():
60      FreeResources()
61      InvalidateSuccessors()
62      Wait()
```

**Listing 2.1:** Code listing of our agent-based mapping approach.

**Backtracking** In case either all solutions in the search space have been filtered out or not sufficient resources could be allocated or reserved for any of the solutions, the mapping algorithm ran into a dead end. Here, a backtracking algorithm is performed. The agent informs its predecessor in the mapping flow that no solution could be found and then goes back into its initial waiting status (Lines 14-16 and 49). The informed agent then frees all resources it had previously allocated or reserved and the ECU to which it is currently mapped is removed from the search space. Afterward, this agent has to in turn inform all other succeeding agents in the mapping flow that its mapping has been invalidated. This prohibits the succeeding agents in other branches from wasting time and unnecessarily occupying resources. Even if the other branches were to find a valid solution, the solution would be based on a mapping of their predecessor that is no longer valid. Furthermore, they are informed that they have to wait again for their predecessor in the mapping flow. After invalidating its successors the backtracking agent tries to find another valid solution (Line 53) and will then again inform its predecessors about the success. If there is no valid solution left or no other solution can be found, it will start the backtracking process of its own predecessors.

#### 2.4.4.4 Resource Manager

In the following we describe how an allocation and reservation request of an agent is performed by a resource manager. Furthermore, we introduce the three allocation and reservation strategies *Random*, *FreeFirst*, and *FreeLast* and describe their respective advantages and disadvantages over each other. These three strategies are evaluated in detail in Subsection 2.4.5.

**Resource Allocation and Reservation** The resource managers receive a request from an agent to allocate or reserve a certain amount of service intervals for CPU resources or slots for a link. It is the job of the corresponding resource manager to execute the allocation or reservation and to answer whether the allocation or reservation could be performed successfully or not. During this process, the resource manager decides which exact service intervals or slots will be allocated or reserved. Here, it ensures that constraints C.3 to C.8 are met as it is programmed to only operate within these bounds. This allocation and reservation process also decides how the system will be degraded in a failure scenario. Subsection 2.4.3.2 describes in detail how a degradation would be performed on the level of service intervals. Suppose a service interval is reserved by a critical passive task instance and allocated by a non-critical task instance. In that case, the service interval is automatically assigned to the critical task instance once a failure occurs. In our system, a critical agent can pass the information in which failure scenarios this activation should occur. A watchdog then informs the resource manager about any failures occurring in the system so that it can immediately react. This behavior is entirely transparent to the agent, and it does not know whether the activation of its passive task instance will lead to a degradation of another task.

**Strategies** In the following, we discuss the three strategies we developed to allocate and reserve resource slots. By default, the resource managers use the *Random* strategy, which randomly chooses the service intervals assigned to an allocation or reservation request. Changing this algorithm can impact the degradation behavior and the success rate of finding a mapping. In the following, we propose two alternative strategies *FreeFirst* and *FreeLast* for assigning service intervals.

The *FreeFirst* strategy aims at minimizing the overlap between reservations and allocations by assigning service intervals first that have not been allocated or reserved. Only if no free service interval is available, the algorithm will allocate or reserve other service intervals. The algorithm has the advantage of reducing the degradation effect as allocations and reservations overlap as little as possible. The downside is that in scenarios where resources are constrained, the algorithm might lead to lower success rates of finding mappings as fewer free service intervals will be available. On the other hand, in more relaxed scenarios, it uses all available resources to reduce the degradation effect with little effect on the success rate.

The *FreeLast* strategy aims to utilize resources more efficiently by assigning service intervals that have already been allocated or reserved, maximizing the overlap between reservations and allocations. The advantage is that even in resource-constrained scenarios, the resources are used efficiently so that the success rate of finding mappings is increased. On the other hand, as the overlap between reservations and allocations is maximized, there will be a stronger degradation effect in case of a failure scenario. In scenarios where resources are less constrained, reservations and allocations will overlap even if additional free service intervals are available, leading to avoidable degradation effects.

When choosing one of the two opposing strategies, a trade-off must be made between degradation effect, success rate, and resource efficiency. The *FreeLast* strategy can lead to lower resource utilization and higher success rates in scenarios where degradation is desired or tolerated. In scenarios where many resources are available and degradation should be avoided as much as possible, the *FreeFirst* strategy maximizes resource utilization to minimize degradation impact. We continue this discussion with our experimental results in Subsection 2.4.5, which gives further insights into these three strategies.

#### 2.4.4.5 Recovery and Reconfiguration

In a failure scenario with a critical application, an immediate failure reaction and failover to the backup solution are required to keep the application operational. Failures can be detected via watchdogs and heartbeats. In Chapter 3, we will present a formal analysis to derive the worst-case application failover time for distributed systems. Here, we analyze the impact of failure detection and recovery times on the timing behavior of distributed applications. This analysis guarantees an upper bound on the time an application would take to generate a new output after the failover of one or multiple task instances. The work in this section can be used as a base for the analysis in Chapter 3 as our system provides an upper bound on task execution and message transmission times.

In case an active task instance of a critical application is affected by the failure, the watchdogs notice a timeout and notify both the resource manager and the agent of the passive task instance. The resource manager immediately turns the agent's reservation into an allocation and performs a degradation if required. The agent of the passive task instance then starts its task. As we use a service-oriented middleware with a publish/-subscribe pattern, this task instance has to subscribe to its predecessor task instances to receive the corresponding messages. In case the preceding task instances have been affected by the failure, they first have to advertise their service before succeeding task instances can subscribe to them. If a passive task instance of a critical application is affected by a failure, no immediate failure reaction is required. If a failover has been performed by a preceding task instance, it also has to wait until the corresponding service is offered until it can re-subscribe to it.

If a task instance of a non-critical application is affected by a failure, no immediate reaction is required. If a non-critical task instance is affected by a degradation, the agent is automatically notified. The degraded non-critical task can run in a degraded mode with fewer available resources or completely shut down.

After the immediate failover, which ensures a safe fail-operational behavior of critical applications, the system can be reconfigured. During this reconfiguration, resources can be freed by shutting down task instances of failed non-critical applications. Furthermore, it is essential to re-establish the fail-operational behavior of critical applications. Here, the agent of the source task invalidates the mapping of all its successors in the mapping order, which in turn also informs all their successors. Afterward, the mapping process can be repeated until a valid configuration is found.

### 2.4.5 Evaluation

We evaluate our performance analysis and our agent-based approach using our in-house developed simulation framework. The framework has been developed to simulate automotive hardware architectures and the execution and communication of the system software according to our system model. On top of the simulation framework, we implemented the agents, resource managers, and strategies as described in Subsection 2.4.4. For the simulation framework, we chose a process-based Discrete-Event Simulation architecture based on the SimPy framework [73]. The hardware architecture and system software are described in a specification file using the XML schema from the OpenDSE framework [72]. The simulation framework supports any hardware architecture consisting of ECUs, switches, and links. To allow a dynamic behavior where tasks and agents are moving between ECUs at run-time, we use a communication middleware based on the SOME/IP standard [36]. The middleware consists of a service discovery that dy-

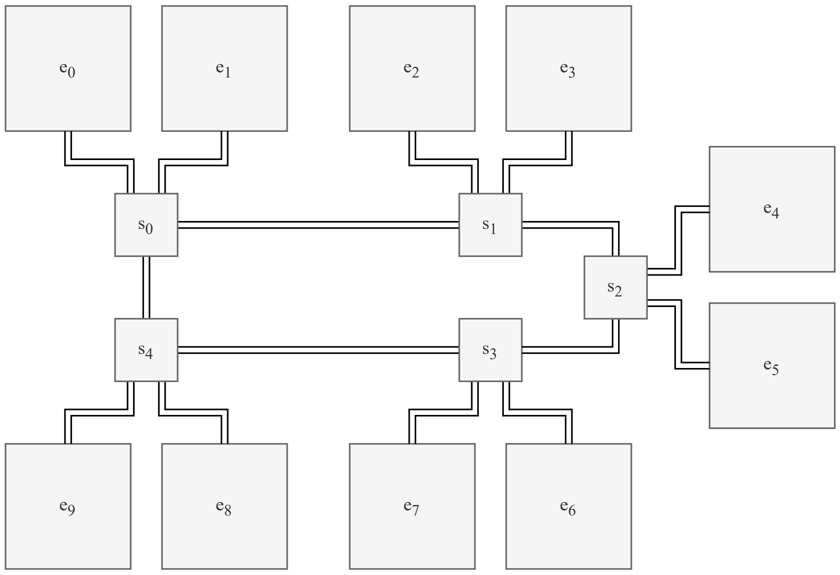


**Figure 2.12:** The simulated hardware architecture used in our experiments. Ten homogeneous ECUs $e \in E$ are connected in pairs via ethernet links $l \in L$ to one switch $s \in S$, forming a ring architecture. Messages are routed using Dijkstra's algorithm [38]. In the worst case, four hops are required to enable communication between two ECUs. Using ethernet links with a data rate of 1 $Gbit/s$, a maximum slot number of $SL_{max} = 1000$ and a slot interval of $\tau_{SL} = 12.5\ \mu s$, the worst-case communication latency for sending a message $m$ over one link is $CL(m, l) = 12.5ms$.

namically finds services at run-time. Communication participants are either modeled as clients or services. Furthermore, the middleware supports remote procedure calls and includes a publish/subscribe scheme. The framework also offers the possibility to simulate ECU failures by shutting down ECUs. ECU failures are detected via heartbeats that are periodically sent between all ECUs. Once a watchdog does not receive the heartbeat within a specific timeout interval it reports the corresponding ECU failure.

#### 2.4.5.1 Setup

The hardware architecture that we use in our experiments, depicted in Figure 2.12 consists of ten homogeneous ECUs connected in pairs to one switch. The switches are connected in a ring architecture, resulting in a maximum hop count of four between each ECU. We use shortest path routing based on Dijkstra's algorithm for routing the messages [38]. For our experiments, we use applications synthetically generated by the OpenDSE framework using the TGFF algorithm [72, 80]. All applications consist of exactly ten tasks with a WCET of $W(t, \alpha(t)) = 2.5ms$ and the requirement to allocate or reserve five service intervals. We design the system so that one ethernet frame with a maximum frame size of 1518 bytes can be sent in one time slot. Using ethernet links with a data rate of 1 $Gbit/s$, we can calculate the transmission time over one link $l$ as

$$\tau_{trans} = \frac{1518\ byte}{1\ Gbit/s} = 12.144\ \mu s. \tag{2.35}$$

Rounding this value, we design our slot interval as $\tau_{SL} = 12.5\ \mu s$. Thus, we can calculate the transmission time of a message over one ethernet link $CL_{trans}(m,l)$ as

$$CL_{trans}(m,l) = \tau_{SL} = 12.5\ \mu s. \tag{2.36}$$

Choosing $SL_{max} = 1000$, we can calculate the worst-case communication latency $CL_(m,l)$ over one link according to Equation 2.13 as

$$CL(m,l) = SL_{max} \cdot \tau_{SL} = 12.5ms. \tag{2.37}$$

The resource managers on the ECUs are managing access to a schedule with $SI_{max} = 250$ service intervals and a service interval time of $\tau_{SI} = 0.5ms$ resulting in a total number of 2500 service intervals.

#### 2.4.5.2 Experiments

We set the deadline $\delta$ for the worst-case end-to-end application latency to $1200ms$, creating scenarios with tight and relaxed timing constraints for the randomly generated applications. For the experiments, we chose 20 non-critical applications and deviated the number of critical applications $N_c$ between 10 and 30 to construct scenarios where resources are constrained and scenarios where the resource situation is relaxed. Results present the average values of 500 runs per configuration. For each single run, a new set of applications was synthetically generated. As the search process could take a lot of time in cases with tight timing constraints or where resources are limited, we set an upper bound of 10,000 backtrack operations as a stopping criterion. Cases that hit this upper bound are considered as failed. We conducted the simulations on a server with an Intel Xeon Gold 6130 CPU with 16 cores running at 2.1 GHz and 128 GB of RAM. Each simulation run was assigned a single CPU and 8 GB of RAM. Simulation runs finished within a few minutes in the majority of cases. In the worst case, it took 76 minutes to complete the simulation.

In the following, we present the mapping success rates of our constraint-solving approach over a deviating number of critical applications. We compare these results to our graceful degradation approach from Section 2.3, where no timing constraints are considered and to an active redundancy where no degradation is applied. Afterward, we evaluate the success rate of our three allocation and reservation strategies from Subsection 2.4.4.4. Then, we further analyze one scenario by presenting Cumulative Distribution Functions (CDFs) over the number of explorations performed for both the successful and failed cases. Furthermore, we present the number of free service intervals and the number of overlapping service intervals for all our three strategies and compare these results to an active redundancy approach. Last, we summarize all findings from these experiments about the difference in our allocation and reservation strategies be-

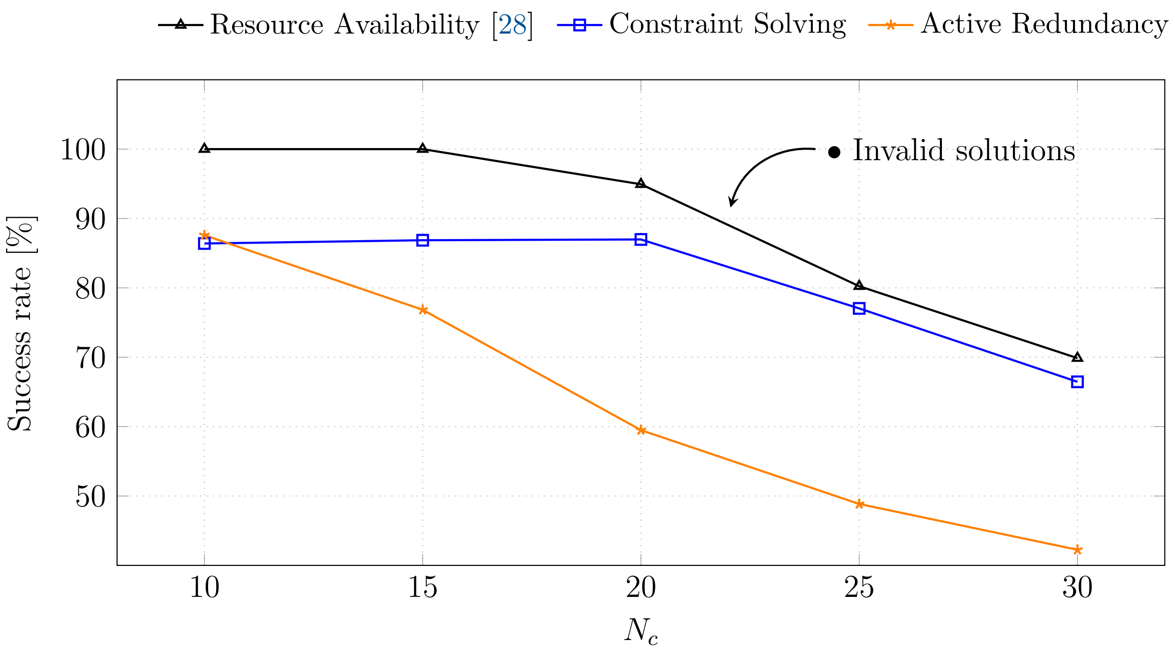


**Figure 2.13:** Experimental results presenting the average mapping success rate of critical applications for deviating numbers of critical applications $N_c$. Using our constraint-solving approach with graceful degradation (blue curve with square marks), the success rate is significantly higher than active redundancy (orange curve with star marks). When not considering timing constraints and only taking resource availability (black curve with triangle marks) into account, as presented in Section 2.3, only invalid solutions would be found, which is not an option.

tween each other and draw a conclusion about the importance of graceful degradation compared to state-of-the-art approaches.

**Success Rate - Constraint Solving vs. State-of-the-Art** Figure 2.13 presents the average success rate of the application mapping processes of critical applications for a deviating number $N_c$ of critical applications. We compare our constraint-solving approach (blue curve with square marks) to our approach in Section 2.3, where timing constraints are not considered and only resource availability is considered. Furthermore, we conducted experiments where constraint solving is applied, but no degradation is allowed (orange curve with star marks), representing approaches with active redundancy or where passive redundancy is used, but resources are allocated exclusively to one task. While the approach based only on resource availability (black curve with triangle marks) can map applications with a success rate of 100% in cases where the resource situation is relaxed, the success rate steadily decreases with an increasing amount of critical applications down to a success rate of 69.7% at a number of $N_c = 30$ critical applications. Looking at our run-time constraint-solving approach, the success rate is lower even in scenarios with a relaxed resource situation, as some timing constraints are too tight to find a valid solution. With an increasing number of critical applications $N_c$, the resources

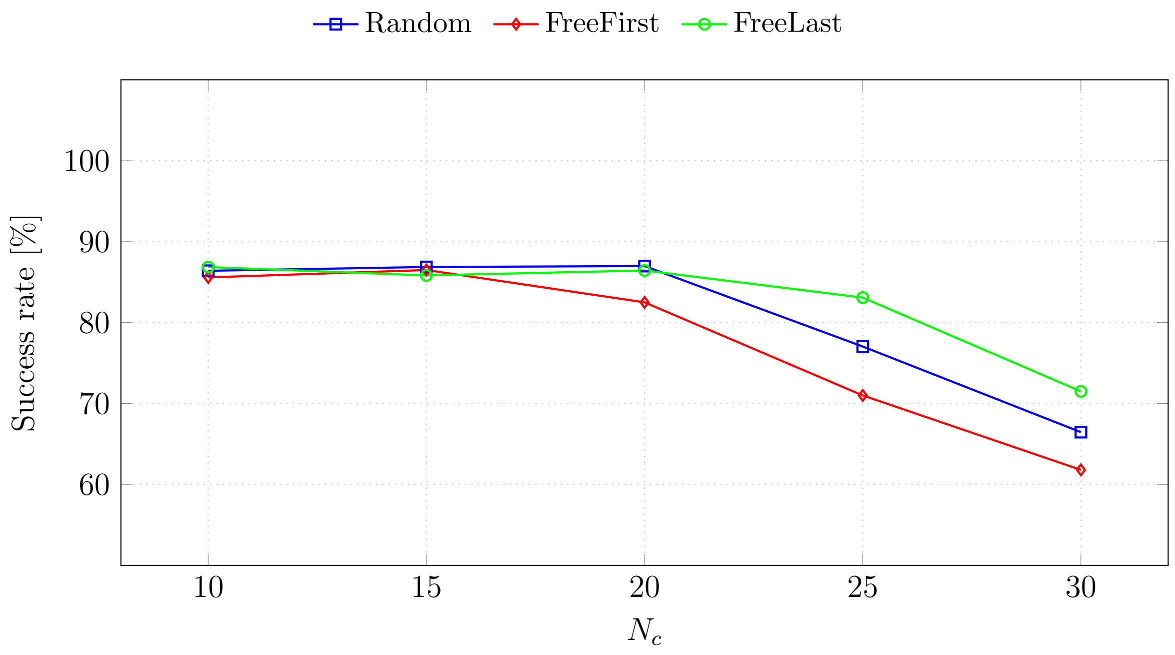
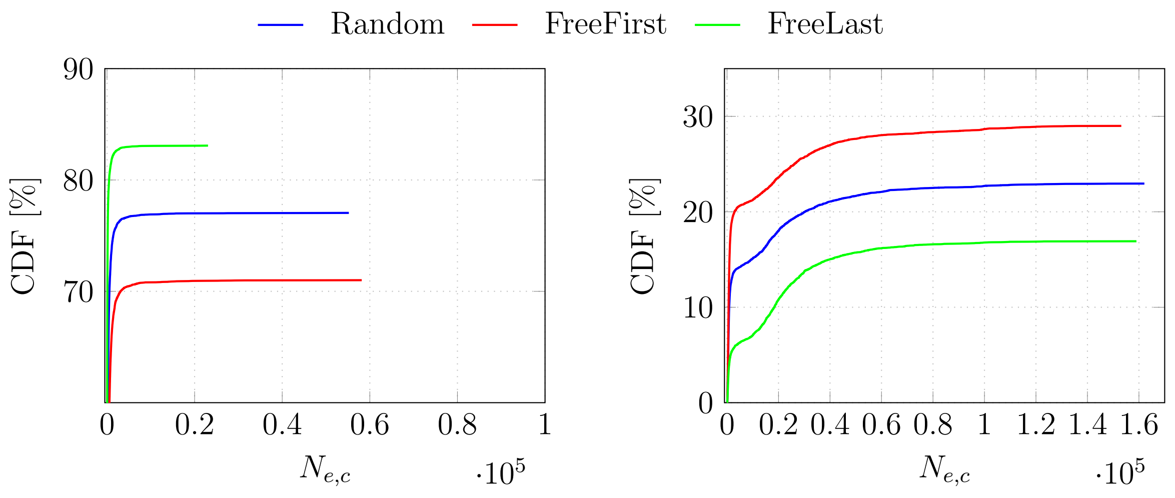


**Figure 2.14:** Experimental results presenting the average mapping success rate of critical applications of our strategies for different numbers of critical applications $N_c$. The *FreeLast* heuristic (green curve with circle marks) leads to increased success rates compared to the random constraint solving (blue curve with square marks), while the *FreeFirst* heuristic (red curve with diamond marks) leads to lower success rates. With lower numbers of critical applications, the heuristics have no impact on the success rate, which is visible as sufficient resources are available to allocate and reserve all required service intervals. With an increasing number of critical applications, resources become more constrained, and the advantages and disadvantages of heuristics become more visible.

become the more limiting factor such that the success rate decreases further but also comes closer to the resource availability curve. These results confirm that an approach based only on resource availability would find mappings that would violate timing constraints in many cases in this setup. The active redundancy approach has a similar success rate to our constraint-solving approach with a number of $N_c = 10$ critical applications. However, with increasing critical applications, the success rate drops steadily with significantly worse results than our constraint-solving approach with degradation. It is visible that graceful degradation can greatly increase the success rate in scenarios where resources become more limited. While the approach based on resource availability appears to have a higher success rate, it mainly generates invalid solutions, which is not an option.

**Success Rate - Strategies** We also evaluate our strategies for allocating and reserving resources presented in Subsection 2.4.4.4. Figure 2.14 presents the success rates of critical applications of the default *Random* strategy (blue curve with square marks) and our two heuristics *FreeFirst* (green curve with circle marks) and *FreeLast* (red curve with diamond marks). For non-resource-constrained cases with 10 and 15 critical applications, the values of all three algorithms are close to each other. Afterward, in scenarios where resources become more limited with an increasing number of critical applications, the success rate decreases for all three strategies. However, the three curves diverge, with the *FreeLast* strategy having the highest success rates and the *FreeFirst* strategy leading to the lowest success rates. In resource-relaxed scenarios, the *FreeLast* heuristic is not advantageous as there are sufficient resources available to reserve and to allocate for both the *Random* and the *FreeFirst* algorithms. In more resource-constrained scenarios, the advantage of the *FreeLast* strategy regarding the more efficient use of resources becomes visible, while the *FreeFirst* heuristic leads to sub-optimal results. We discuss in Subsection 2.4.5.2 that there can be scenarios in which the *FreeFirst* strategy can have advantages.



**(a)** Successful mappings.

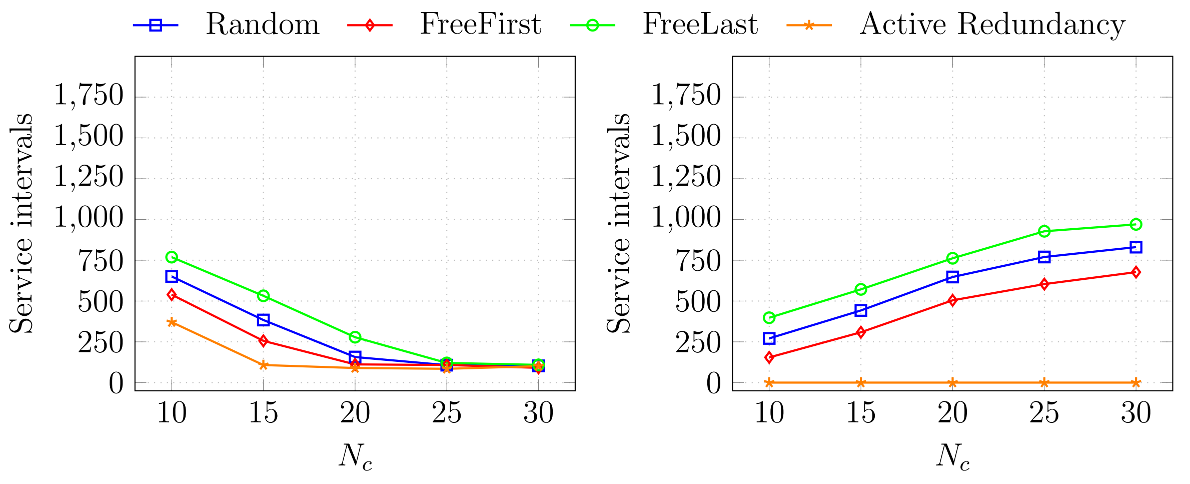
**(b)** Failed mappings.

**Figure 2.15:** CDFs of the number of total number of epxlorations of the critical applications $N_{e,c}$ in the scenario with $N_c = 25$ critical applications for both the successful cases (Subfigure 2.15a) and the failed cases (Subfigure 2.15b). Most applications that successfully found a mapping require clearly less than 500 explorations, with only a few exceptions requiring more explorations. Applications that could not find a mapping within the upper limit of 10,000 backtracking operations often required more than 500 explorations.

**Cumulative Distribution Function** To allow a more fine-grained analysis of these three strategies we present the CDFs of the number of explorations $N_{e,c}$ tested per critical application for the scenario with $N_c = 25$ critical applications split up into successful and failed mapping cases in Figure 2.15. The number of explorations $N_{e,c}$ defines how many times the agents of critical applications have explored a mapping by sending allocation and reservation requests for a potential solution. In a best-case scenario, every agent finds a mapping on the first try. In the worst-case scenario, mappings are tested until our stopping criteria of 10,000 backtrack operations is reached. It can be observed that in the case of successful mappings, almost all mappings are found with less than 500 allocation or reservation requests and only a few outliers. For applications that could not find a mapping within the upper limit of 10,000 backtrack operations many could confirm within 500 explorations that no mapping can be found. This implies that applications started backtracking relatively early in the search process as tasks that are further up in the application graph could not find a mapping due to limited resources such that it can be confirmed relatively quickly that no mapping can be found. However, for many applications it frequently took more than 500 explorations up to around 16,000 explorations. In these cases, many task instances were already mapped until running into a shortage of resources or a violation of a timing constraint such that many explorations would have to be performed to confirm that no solution exists. The curves of the *FreeFirst* and the *FreeLast* strategies appear to have a similar trend as the curve of the *Random* strategy with a positive or negative offset. In the case of failed mappings, this means for the *FreeFirst* strategy that more cases are counted as failed after relatively few explorations implying that mappings could not be found due to resource constraints. On the other hand, in the case of the *FreeLast* strategy, this implies that more applications could find a mapping compared to the other two strategies as applications did not run as often into resource constraints. Therefore, these results further confirm the findings that the *FreeLast* strategy has an increased success rate due to utilizing service intervals more efficiently, while the *FreeFirst* strategy runs into resource constraints more often.

**Service Interval Utilization** Figure 2.16 presents the number of free service intervals and the number of overlapping service intervals of the experiments with a deviating number of critical applications $N_c$. It can be observed that the curves are running almost parallel to each other with an offset until a minimum close to zero is hit. In most cases, some service intervals remain unoccupied. However, not all applications can be mapped as the remaining service intervals are distributed over multiple ECUs and might only be a fraction of the service intervals required to map an application. The *FreeLast* strategy has the most free service intervals for the scenarios with fewer critical applications. In comparison, the *FreeFirst* algorithm leads to fewer free service intervals than the *Random* strategy. The active redundancy approach leads to the lowest number of freely available service intervals. Comparing Subfigure 2.16a to Figure 2.13 and Figure 2.14 we can see that the success rate always starts to decrease when the curve hits a low in the number of freely available service intervals. This observation confirms our previous findings that the number of available resources limits the success rate.

However, this advantage is bought with a likely more reduced functionality of non-critical applications after a failover scenario. In the case of active redundancy, no service intervals overlap as no degradation is performed at all. The other three strategies lead to increasingly more overlapping service intervals with an increasing number of critical applications, with the *FreeLast* resulting in the most overlapping service intervals and the *FreeFirst* strategy resulting in the lowest number of overlapping service intervals. The more service intervals overlap, the more likely a non-critical application will be degraded if the corresponding service interval is required by a passive task instance



**(a)** Average number of free service intervals. **(b)** Average number of overlapping service intervals.

**Figure 2.16:** Experimental results of the average numbers of free and overlapping service intervals with a total number of 2500 service intervals distributed over ten ECUs. The results are consistent with success rates from Figure 2.14. A higher number of free service intervals is better, allowing for mapping more applications and increasing the success rate. A higher number of overlapping service intervals directly increases the number of free service intervals, increasing the success rate. The *FreeFirst* heuristic occupies more service intervals than the default service intervals distribution algorithm, while the *FreeLast* heuristic occupies fewer service intervals leading to higher success rates. On the other hand, the *FreeLast* heuristic has a higher number of overlapping allocations and reservations, leading to a higher degradation impact. In comparison, the number of overlapping service intervals is lower for the *FreeFirst* heuristic. The active redundancy approach requires the most service intervals and does not result in overlapping service intervals.

performing a failover. The number of overlapping service intervals also explains the difference in free service intervals between the three strategies. The *FreeLast* strategy overlaps service intervals whenever possible, keeping a maximum of free service intervals available, which can be used to map further applications. The *FreeFirst* strategy instead avoids overlapping service intervals as much as possible leading to less freely available service intervals and thus limiting the amount of applications that can be mapped. This means at the same time that the *FreeLast* strategy is more likely to result in a degradation in a failover scenario while the *FreeFirst* minimizes this risk. In scenarios with constrained resources, the *FreeLast* strategy can use the resources more efficiently by overlapping service intervals, resulting in higher success rates of finding mappings and significantly higher degradation effects on non-critical applications. In scenarios with relaxed resource situations, the *FreeLast* heuristic does not use the plentiful available resources and might result in degradation scenarios that could be avoided when using all resources. On the other hand, the *FreeFirst* heuristic has the disadvantage of having a lower success rate in resource-constrained scenarios as overlapping service intervals are avoided as far as possible. In more relaxed situations, it uses all available resources to prevent degradation as much as possible, leading to more optimal solutions.

### 2.4.6 Summary

In summary, in this section, we presented an agent-based approach that enables, for the first time, graceful degradation for real-time automotive applications. Our approach guarantees predictability for end-to-end timing constraints and enables a graceful system degradation while ensuring the fail-operational requirements of critical applications. The advantage is that resources can be utilized more efficiently in mixed critical systems if a reduced functionality of non-critical applications after a failover can be accepted. We first introduced state-of-the-art predictable timing analysis for composable scheduling to enable this behavior and adapted it to our system model. Then, we extended this predictable timing analysis for fail-operational systems to include backup solutions to ensure that there is always a solution that fulfills the end-to-end application latency constraint. Furthermore, we introduced our composable scheduling of gracefully degrading systems, which allows critical backup solutions to reserve service intervals allocated by non-critical applications. In a failure scenario, once a vital backup solution has to be started, it can take over the resources such that the system is being degraded in an intended way. Compared to active redundancy, the advantage of using our passive backup solution is that almost no overhead is added regarding the required computational power. We also presented our agent-based approach, which uses our run-time constraint-solving approach to find solutions that meet resource and timing constraints. Here, a pre-defined mapping order of the tasks is required to ensure that constraints can be met. If the mapping search runs into a dead end, backtracking is applied to explore other possible solutions. Resource managers receive requests from agents to allocate and reserve resources and decide which service intervals will be assigned. Here, we introduced three new strategies *Random*, *FreeLast*, and *FreeFirst*, for the assignment of the resources which heavily influence how the system will be degraded. We performed multiple experiments on our simulation platform to evaluate our approach.

Summarizing our observations and findings from our experimental results, we have shown that it is necessary to use our constraint-solving approach as approaches based only on resource availability would result in solutions violating timing constraints. Additionally, our assumptions about the advantages and disadvantages of the three allocation and reservation strategies *Random*, *FreeFirst*, and *FreeLast* from Subsection 2.4.4.4 have been confirmed. All results together provide a clear picture that the *FreeLast* strategy results in higher success rates by overlapping service intervals as far as possible, thus leaving more free service intervals to map other applications. While this strategy leads to increased success rates in resource-constrained scenarios, there is an increased degradation effect that could be avoided in scenarios with a relaxed resource situation. The *FreeFirst* strategy leads to lower success rates by avoiding overlapping service intervals as far as possible, leading to less freely available service intervals and, thus, fewer applications that can be mapped. While this strategy reduces the degradation effect in scenarios with plentiful resources, it results in significantly lower success rates than the other two strategies in resource-constrained scenarios. Overall, the *FreeLast* maximizes the graceful degradation effect while the *FreeFirst* strategy minimizes it with all accompanying advantages and disadvantages for both.

Most importantly, our experiments have confirmed that graceful degradation strongly increases the success rate of finding a mapping for critical applications compared to an active or passive redundancy approach without degradation in resource-constrained scenarios. When using the *FreeLast* strategy more than double the number of critical applications can fit onto the system architecture in our setup compared to an active redundancy approach before an effect on the success rate becomes observable. Our experiments have shown that graceful degradation can significantly increase the success rate in scenarios where resources are limited if the risk of losing non-critical functionality in a failure scenario is acceptable. In summary, by enabling graceful degradation for real-time applications with our predictable timing analysis and agent-based approach, resources can be utilized more flexibly and efficiently while guaranteeing a safe and dynamic behavior of automotive systems.

## 2.5 Limitations

In the following we address the limitations on scalability and the reconfiguration time of our fail-operational approach.

### 2.5.1 Scalability

Proving whether a valid mapping and scheduling solution exists is an NP-complete problem that would take exponential time in the worst case [81, 76, 67]. Our approach assumes that the problem is solved on a few powerful ECUs interconnected by high-speed Ethernet and that most of the resources are available for the mapping process. The mapping process itself is not performed during a time-critical phase and only while the car is safely parked such that longer search times might be acceptable. Our simulation times on a single CPU with 8 GB of RAM typically ranged from a few minutes up to 76 minutes. For an in-car implementation and the given problem size, we would assume that in the worst case, 18 messages (1 for the CPU, 8 for the links, two-way) are sent per exploration over 4 hops. With a transmission time of $CL_{trans}(m,l) = 12.5\ \mu s$ and the maximum number of around $N_{e,c} = 160000$ explorations (see Figure 2.15b) measured, the time taken to send all monitoring messages of a critical application would equate to $\tau = 18 * 4 * 12{,}5\ \mu s * 160000 = 144s$. Please note that the time required to find a mapping is usually smaller by multiple orders of magnitude. For our given problem size, we would assume these transmission times are acceptable. However, to address scalability issues and in-car resource limitations of our approach there is exciting research on hybrid mapping methods such as presented in [81]. Here, meta-heuristic optimization approaches are used to find multiple Pareto-optimal solutions at design time. Run-time backtracking approaches then perform constraint solving. In

[81] the backtracking approach finds a solution for a comparable problem size within a few milliseconds. However, it has to be assumed that communication within a NoC architecture is faster than on the system level. Future work could include a hybrid mapping approach with the pre-computation of possible mappings at design time such that new mappings after failover are found faster.

### 2.5.2 Reconfiguration Time

Another limitation of our approach is that after a failover has been performed, no safe operation of the critical applications can be guaranteed, as another critical failure could lead to potentially hazardous situations. Here, the car has to stop and start a re-mapping process to re-establish the fail-operational behavior. Adding another layer of redundancy could potentially resolve this issue but would increase the problem's complexity significantly as the communication between all redundant instances would need to be ensured. Hybrid mapping approaches could greatly reduce the time spent on finding a new mapping. Additionally, there is research on performing real-time task migrations [40]. Using such an approach a new mapping could be applied while the car is actively driving.

## 2.6 Conclusion

In this chapter, we have presented our fail-operational architecture, which consists of an agent-based graceful degradation approach. With our graceful degradation approach, resources allocated by non-critical applications can be utilized by critical applications as a backup reserve. With our reservation approach, we enable graceful degradation while preventing an overestimation of the remaining resources such that it can be predicted if sufficient resources are available to withstand another failure. The decentralized approach ensures that no single point of failure can affect the system's control. As it is crucial for a fail-operational architecture to respect timing constraints, we extended our base approach such that timing guarantees can be given to safety-critical applications. This approach also ensures that a valid backup solution is available after any single ECU failure which also respects timing constraints.

Using our reconfiguration approach, we have shown that our agent-based graceful degradation approach can significantly increase the number of tolerated failures without adding hardware resources to the system. In a system with timing constraints, our graceful degradation approach could fit double the number of critical applications on the same platform before influencing the success rate of finding a mapping compared to state-of-the-art approaches such as active redundancy. Furthermore, significantly fewer service intervals were required by our graceful degradation approach. Our *FreeLast* strategy further enhances the graceful degradation effect by overlapping as many service intervals as possible. In contrast, the *FreeFirst* strategy reduces the graceful degradation by avoiding overlapping service intervals.

In summary, graceful degradation can be a powerful methodology that uses resources more efficiently than common redundancy approaches and can enormously increase the number of applications mapped onto the same system architecture while providing the same fail-operational capabilities, however, when designing a system it has to be considered that more non-critical functionality is lost due to degradation in a failover scenario. Suppose the tasks of non-critical applications are widely distributed in the system. In that case, the functionality might be lost anyway due to a direct failure impact, lowering the amount of non-critical applications lost due to a degradation effect. Nonetheless, a trade-off between degradation impact, mapping success rate, and resource availability has to be carefully evaluated. We further address the impact of graceful degradation on non-critical functionality and evaluate resource utilization in more detail in Chapter 5.

As presented in this chapter, one of the main challenges when designing a dynamic fail-operational system is to achieve a predictable behavior and to find bounds within which a safe and dynamic operation is possible. Therefore, we investigate further non-functional properties such as timing, checkpointing, and reliability to increase the predictability of our approach in the following chapters.

# 3 Worst-Case Failover Timing Analysis of Distributed Fail-Operational Automotive Applications [4]

## Contents

## 3.1 Introduction

In Chapter 2.4 we presented an approach to determine if timing constraints are respected and if valid application mappings are readily available as a backup that also adhere to these timing constraints in the case of an ECU failure. In a failure scenario, the system can switch to these pre-validated backup mappings without searching or verifying them in the time-critical phase of a failover. However, it is also important that these new mappings are applied quickly and that all applications are operational again on time. This chapter covers the critical time phase during a failover until the system has switched to the pre-validated mappings. In this context, it is important that a failover within the

[4]Major parts of this chapter have been published in [30].

FTTI can be guaranteed [42]. Failing to perform a failover within the FTTI will lead to unpredictable application behavior with potentially hazardous consequences.

In this work, we analyze the impact of failover on the timing behavior of distributed fail-operational applications and derive an upper bound for the worst-case failover time. The work that we present can be used to evaluate and verify the worst-case failover timing behavior of fail-operational distributed applications. Instead of performing time-consuming experiments, our formal analysis can be used to assess whether a mapping would meet the failover timing constraints or not such that an evaluation at run-time is possible. Therefore, we make the following contributions:

- Based on the application and failover model presented in Section 3.2, we introduce a formula to derive the application failover time in Section 3.3. Here, we analyze worst-case scenarios to derive an upper bound for the failover time. Our upper bound can be used to achieve a predictable fail-over behavior and to verify application requirements on failover constraints.

- We support our formal analysis by conducting failover experiments on our demonstrator platform using a fail-operational distributed neural network in Section 3.4.

## 3.2 System Model Adaptations

In the following, we present minor adjustments and additions to our system model, which we presented in Section 2.2. Here, we mainly assume that a system with valid mappings as described in Section 2.4 is already given such that it can be analyzed.

### 3.2.1 Application Model

For our analysis, we assume that each node in the application graph $G_A(V, E)$ has, at maximum, one predecessor and one successor such that the application graph builds a task chain.

We assume a valid binding $\alpha : T \rightarrow E$ is already given, which assigns each active task instance $t \in T$ to an ECU $\alpha(t) \in E$. For our safety-critical applications $a$, we assume that a redundant passive task instance is available in the system. Here, we assume a binding $\beta : T \rightarrow E$ is given, which assigns each passive task instance $t \in T$ to an ECU $\beta(t) \in E$. Furthermore, we assume that a routing $\rho : M \rightarrow 2^L$ is given, which assigns each message $m \in M$ to a set of connected links $L' \subseteq L$ that establish a route $\rho(m)$. As the routing will also change after a failover, up to three passive routes are required, of which one will become activated depending on which tasks are affected by the failover.

We use the notation $\mathcal{L}_i(a)$ to define the end-to-end application latency of a single iteration $i$. Using composable task and communication scheduling, the interference between tasks and messages can be bounded [17, 74], such that a worst-case $\mathcal{L}_{wc}(a)$ and best-case application latency $\mathcal{L}_{bc}(a)$ can be obtained. Similarly, we define $\mathcal{L}_{bc}(t)$ and $\mathcal{L}_{wc}(t)$ as the best-case and worst-case latency from the application start until task $t$ has finished execution. We assume that the application is periodically executed with the period $P_a$ and that the application might operate in a pipeline such that $P_a$ can be smaller than the worst-case application latency $\mathcal{L}_{wc}(a)$.

### 3.2.2 Failover Model

We define a failure $f \in F$ with $F \subseteq E$, where $f$ identifies the failed ECU. To describe the bindings after the $j$-th failure, we use the notation $\alpha_j(t)$ and $\beta_j(t)$, with $\alpha_0(t)$ and $\beta_0(t)$ being the initial bindings. A failover is required once an ECU $e$ fails to which at least one active task instance of a safety-critical application $a$ has a binding: $\exists t \in T : \alpha_j(t) = e$. In a failover scenario, we assume that affected task instances are lost and that tasks are restarted using the passive task instances such that the active binding of the affected task instances is changed to the former passive task binding: $\alpha_{j+1}(t) = \beta_j(t)$. Furthermore, a new binding for the passive task instance $\beta_{j+1}(t)$ has to be found. In a scenario where only a passive task instance is lost, no restart is required such that the binding of the active task instance remains the same, and only a new binding for the passive task instance has to be found. Similarly, one of the passive routing paths between the new active task instances has to be activated. The new active routing path $\rho_j(m)$ depends on which tasks are affected by the failover and is being implicitly updated. To identify the application latencies after a failure $f$ we use the notations $\mathcal{L}_i(a, f)$, $\mathcal{L}_{wc}(a, f)$ and $\mathcal{L}_{bc}(a, f)$.

## 3.3 Worst-Case Failover Timing Behavior

With dynamic resource management that meets mapping decisions at run-time, verifying every system constraint at design-time is no longer possible. The timing behavior of applications is heavily influenced by timing interference from other applications, which is unknown at design time and can change once applications are updated and new functionality is added to the system.

It is crucial to automatically verify critical system aspects at run-time to enable dynamic resource management for safety-critical applications. In this context, an automated worst-case failover timing analysis is essential, which allows the setting of bounds on the dynamic system behavior when meeting mapping decisions to achieve a deterministic and predictable system behavior. Although our research focuses on automating the failover analysis to achieve dynamic system behavior, our equation can also be used to automatically evaluate the failover timing behavior at design time such that multiple mapping configurations can be explored.

To achieve such a predictable failover timing behavior, we first define the application failover time $\mathcal{F}(a, f)$ in Subsection 3.3.1. Afterward, we present an upper bound and approach to derive the worst-case application failover time $\mathcal{F}_{wc}(a, f)$ in Subsection 3.3.2. Step by step, we first introduce the worst-case recovery time in Subsection 3.3.3, which significantly influences the worst-case application failover time. Then, we analyze a single task failover scenario in Subsection 3.3.4 until we finally derive a generalized formula for

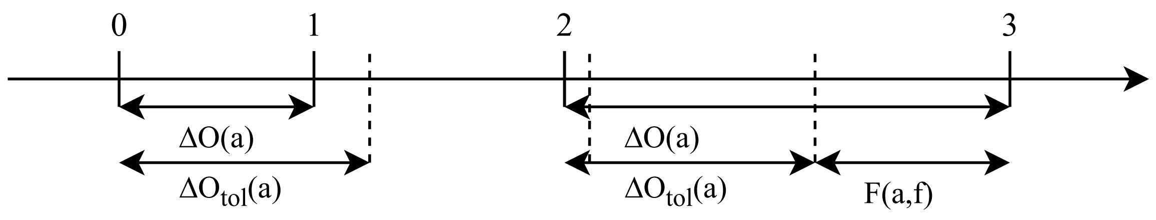


**Figure 3.1:** Depiction of the application failover time: The vertical lines indicate the output time of the corresponding iteration, while the dashed vertical lines depict the latest output time that would be tolerated.

the worst-case application failover time in Subsection 3.3.5, where multiple tasks might be affected by the failover.

### 3.3.1 Application Failover Time

In a flawless operation mode, an application will periodically produce output data with the period $P_a$. Since the application latency is not constant and might differ between two consecutive iterations $i$ and $i+1$, we can calculate the output time difference between two application outputs in a failure-free operation as $\Delta O(a) = P_a + (\mathcal{L}_{i+1}(a) - \mathcal{L}_i(a))$, where $\mathcal{L}_{i+1}(a)$ and $\mathcal{L}_i(a)$ are the corresponding end-to-end application latencies of the iterations $i$ and $i+1$.

Therefore, the maximum delay that can be tolerated after an iteration $i$ would be if it took the worst-case latency $\mathcal{L}_{wc}(a)$ to execute iteration $i+1$ such that the maximum tolerated output delay for the current iteration $i$ can be calculated as

$$\Delta O_{tol}(a) = P_a + (\mathcal{L}_{wc}(a) - \mathcal{L}_i(a)). \tag{3.1}$$

However, when a task failure occurs, the application might take longer to produce a valid output, such that we define $\Delta O(a,f)$ as the output delay during a failover scenario with

$$\Delta O(a,f) = P_a + N_t(a,f) \cdot P_a + \mathcal{L}_{i+1}(a,f) - \mathcal{L}_i(a). \tag{3.2}$$

Here, $N_t(a,f) \cdot P_a$ describes the additional time delay due to iterations that are lost or missed due to failure and recovery time effects as the application is not able to operate during the downtime. Furthermore, $\mathcal{L}_{i+1}(a,f)$ is the application latency of the following valid iteration $i+1$ after the failover with the new active task bindings.

With this we can define the application failover time $\mathcal{F}(a,f)$ as the time window between the point in time where a delay under a failure-free operation would not be tolerated anymore until the point in time where a new output is available after the recovery:

$$\mathcal{F}(a,f) = \Delta O(a,f) - \Delta O_{tol}(a). \tag{3.3}$$

From Equation 3.3 it can be observed that if $\Delta O_{tol} \geq \Delta O(a,f)$, the failover time $\mathcal{F}(a,f)$ might be masked by $\Delta O_{tol}$ and not observable at the application output at all. Putting

Equations 3.1 and 3.2 into Equation 3.3 we obtain the failover time:

$$\mathcal{F}(a,f) = N_t(a,f) \cdot P_a + \mathcal{L}_{i+1}(a,f) - \mathcal{L}_{wc}(a). \tag{3.4}$$

Figure 3.1 depicts the relation between the application failover time $\mathcal{F}(a,f)$, the output delay $\Delta O(a)$, and the maximum tolerated output delay $\Delta O_{tol}(a)$. Here, a failure occurs after iteration 2, such that one frame is missed, causing the output delay to exceed the tolerated delay.

## 3.3.2 Worst-Case Application Failover Time

Our goal is to derive an upper bound for Equation 3.4 to achieve a predictable failover behavior of the application. In a worst-case scenario $\mathcal{L}_{i+1}(a,f)$ from Equation 3.4 would be equal to the worst-case latency $\mathcal{L}_{wc}(a,f)$. Furthermore, we have to identify the worst-case number of iterations $N_{t,wc}(a,f)$ that are lost during the failover, such that we can calculate the worst-case application failover time as

$$\mathcal{F}_{wc}(a,f) = N_{t,wc}(a,f) \cdot P_a + \mathcal{L}_{wc}(a,f) - \mathcal{L}_{wc}(a). \tag{3.5}$$

Using this equation, we can observe that a change to a lower worst-case latency $\mathcal{L}_{wc}(a,f)$ after a failure can have a positive effect on the worst-case failover time as the next iteration after a failure would be able to propagate faster through the complete task chain and, therefore, reach the output earlier than expected beforehand.

For the total number $N_{t,wc}(a,f)$ of lost frames, we distinguish between the iterations $N_{l,wc}(a,f)$ that are lost immediately as they were currently held by a task that was affected directly by the failover and the iterations that are missed $N_{m,wc}(a,f)$ due to the system recovering too slow such that the tasks miss potential inputs:

$$N_{t,wc}(a,f) = N_{l,wc}(a,f) + N_{m,wc}(a,f). \tag{3.6}$$

However, before we can derive $N_{t,wc}(a,f)$, we have to introduce the worst-case recovery time $\tau_r(t,f)$ which is responsible for the number of iterations $N_{m,wc}(a,f)$ that will be missed due to task recovery. Afterward, we present an analysis to calculate $N_{t,wc}(a,f)$ in a simplified scenario where only a single task is affected directly by the failover and then generalize the formula for multiple recovering tasks.

$$\tau_r(t,f) = \begin{cases} 0 & \alpha_{j+1}(t) = \alpha_j(t) \wedge \alpha_{j+1}(p(t)) = \alpha_j(p(t)) \\ \tau_d + \tau_{sub} & \alpha_{j+1}(t) = \beta_j(t) \wedge \alpha_{j+1}(p(t)) = \alpha_j(p(t)) \\ \tau_r(p(t),f) + \tau_{off} + \tau_{sub} & \alpha_{j+1}(t) = \alpha_j(t) \wedge \alpha_{j+1}(p(t)) = \beta_j(p(t)) \\ \tau_r(p(t),f) + \tau_{off} + \tau_{sub} & \alpha_{j+1}(t) = \beta_j(t) \wedge \alpha_{j+1}(p(t)) = \beta_j(p(t)) \end{cases} \tag{3.7}$$

### 3.3.3 Worst-Case Task Recovery Time

We define $\tau_r(t, f)$ for a failover $f$ as the worst-case time frame that it takes from the occurrence of a failure until a task $t$ is ready to receive and process the following input after recovery. Our definition of the recovery time is based on our system, which uses a service-oriented middleware with a publish/subscribe pattern. Here, we make the case distinctions as presented in Equation 3.7, which can be found below.

For a task $t$, it holds that $\tau_r(t, f) = 0$ if neither its active task binding nor the active task binding of its predecessor is affected by the failover.

If a task is affected directly by the failure and has to be restarted, the passive task instance has to detect the failure. In the following, we assume that the worst-case failure detection time $\tau_d$ is equal for every failure and every ECU. After detecting the failure, the task has to subscribe to its predecessor to receive its message, where we assume a worst-case subscription time $\tau_{sub}$ resulting in $\tau_r(t, f) = \tau_d + \tau_{sub}$.

In case a predecessor task $p(t)$ is affected by the failover and not the task itself, the active task instance has to wait until the preceding task has restarted and sent a message to offer the service, while the task itself has to wait to detect the failure until it can finally subscribe. $\tau_r(t, f) = \max(\tau_r(p(t), f) + \tau_{off}, \tau_d) + \tau_{sub}$.

If a predecessor task $p(t)$ and the task $t$ itself are affected by the failover, the task has to detect the failure and wait until the offer service message has arrived until it can subscribe, just as in the previous case. With $\tau_r(p(t), f) + \tau_{off} > \tau_d$, we can then further simplify the formula for the last two cases to $\tau_r(t, f) = \tau_r(p(t), f) + \tau_{off} + \tau_{sub}$. With the worst-case recovery time, it is now possible to derive the worst-case number of iterations lost during a failover.

### 3.3.4 Single Task Failover Scenario

In a scenario where only a single task is affected directly by a failure, the worst point in time where most iterations are lost is when the affected active task instance has just finished the execution and the frame $l_0$ is lost together with the task such that for a single failing task $N_{l,wc}(t, f) = 1$.

If the passive task instance that is being started can recover on time $\tau_r(t, f)$ before the following frame $m_0$ has reached $p(t)$, no additional frame will be missed. Otherwise, at least one other frame $m_0$ will be missed depending on how fast the task can recover $\tau_r(t, f)$.

To calculate the earliest point in time $\tau_{m_0}$ where $m_0$ would reach the predecessor at the end of the predecessor $p(t)$ of task $t$, we have to know how far the next frame $m_0$ was behind the lost frame $l_0$. The frames start with a time difference of exactly $P_a$. In the worst case, it took the lost frame $l_0$ the worst-case latency $\mathcal{L}_{wc}(t_l)$ to reach the output of the task, such that the following frame $m_0$ had the most time to minimize the time distance between them. In turn, $m_0$ would in the worst case propagate with the best-case latency $\mathcal{L}_{bc}(p(t))$ such that we can calculate $\tau_{m_0}$ as

$$\tau_{m_0} = P_a + \mathcal{L}_{bc}(p(t)) - \mathcal{L}_{wc}(t). \tag{3.8}$$

We assume that in the worst case, every following frame $m_i$ would propagate with the best case latency and therefore be exactly behind by $P_a$ to the previous frame $m_{i-1}$:

$$\forall i > 0 : \tau_{m_i} = \tau_{m_{i-1}} + P_a \tag{3.9}$$

With this, we can make the following case distinction for the worst-case number of missed frames:

$$N_{m,wc}(t,f) = \begin{cases} 0 & \tau_{m_0} > \tau_r(t,f) \\ 1 + \lfloor \frac{\tau_r(t,f) - \tau_{m_0}}{P_a} \rfloor & \tau_{m_0} \leq \tau_r(t,f) \end{cases}. \tag{3.10}$$

If the frame $m_0$ would reach the end of $p(t)$ before the task $t$ has recovered with the worst-case recovery time $\tau_r(t,f)$, then $m_0$ will be missed. As the following frames follow $m_0$ by $P_a$ in the worst case, an additional number of $\lfloor \frac{\tau_r(t,f) - \tau_{m_0}}{P_a} \rfloor$ frames will be missed until the recovery is completed.

### 3.3.5 Multi Task Failover Scenario

For a scenario where multiple tasks must recover, more than one frame might get lost immediately due to the failure if the application uses pipelining. In the following, we denote task $t_l$ as the last task and $t_f$ as the first task in the chain to recover. We assume that any frames that at the point of failure were between the predecessor task $p(t_f)$ of $t_f$ and $t_l$ are immediately lost. In the worst case, at least one frame $l_0$ is lost at $t_l$, which just has completed execution with the corresponding frame. To understand how many other frames between $p(t_f)$ and $t_l$ are lost, we must make similar considerations as in Subsection 3.3.4. Most frames are lost if the frame $l_0$ at $t_l$ is propagated with the worst-case latency, and the following frames are propagated with the best-case latency. Therefore, if $l_1$ propagated faster to $p(t_f)$ with $P_a + \mathcal{L}_{bc}(p(t_f))$ than it took for $l_0$ to reach the end of $t_l$ with $\mathcal{L}_{wc}(t_l)$, at least one additional frame is immediately lost during the failure. Therefore, we define

$$\tau_{l_1} = P_a + \mathcal{L}_{bc}(p(t_f)) - \mathcal{L}_{wc}(t_l) \tag{3.11}$$

as the time where $l_1$ would reach the end of execution at $p(t_f)$. Similar as before, all following frames $l_i$ are following their predecessor by $P_a$:

$$\forall i > 1 : \tau_{l_i} = \tau_{l_{i-1}} + P_a \tag{3.12}$$

If $\tau_{l_1}$ is positive, which means $l_1$ reached $p(t_f)$ after the failure event, no other frame than $l_0$ is lost. Otherwise, $l_1$ and possibly other frames are lost. Thus, we can calculate the number of lost frames as

$$N_l(a,f) = \begin{cases} 1 & \tau_{l_1} > 0 \\ 2 + \lfloor \frac{-\tau_{l_1}}{P_a} \rfloor & \tau_{l_1} \leq 0 \end{cases} \tag{3.13}$$

We assume that $\forall t \in T : \tau_{sub} + \tau_{off} < l_{t,bc}(t, \alpha(t, f)) + l_{c,bc}(m_{in}, \rho(m_{in}, f))$, such that the system is dominated by the failover of the first task $t_f$ in the chain that failed. If $t_f$ has recovered, we assume that all the following tasks in the chain can recover on time such that no additional frames are lost except the frames $N_m(t_f, f)$ missed by $t_f$.

$$\tau_{m_0}(t_f, f) = (N_l(a, f) - 1) \cdot P_a + \tau_{l_1} \tag{3.14}$$

Here, we can reuse Equation 3.10 to calculate $N_{m,wc}(t_f, f)$ with $\tau_{m_0}(t_f, f)$, such that we can calculate the worst-case number of missed frames. By combining and simplifying the complete formula for $N_t(a, f)$ we obtain

$$N_t(a, f) = \lfloor \frac{\tau_r(t, f) + \mathcal{L}_{wc}(t_l) - \mathcal{L}_{bc}(p(t_f))}{P_a} \rfloor + 1. \tag{3.15}$$

By putting Equation 3.15 into Equation 3.5, we can finally derive the generalized formula to calculate the worst-case application failover time.

## 3.4 Evaluation

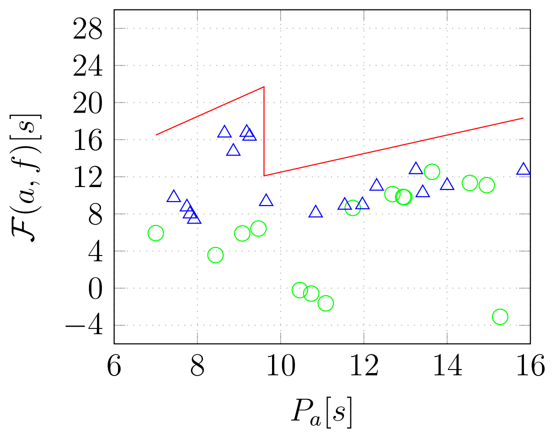
In the following, we present the results of our experiments conducted on a demonstrator setup to support our worst-case failover analysis using a fail-operational distributed neural network. In a usual system setup, we assume that multiple distributed applications run simultaneously. Here, our analysis can be applied for each application individually as long as worst-case and best-case latencies can be determined.

### 3.4.1 Setup

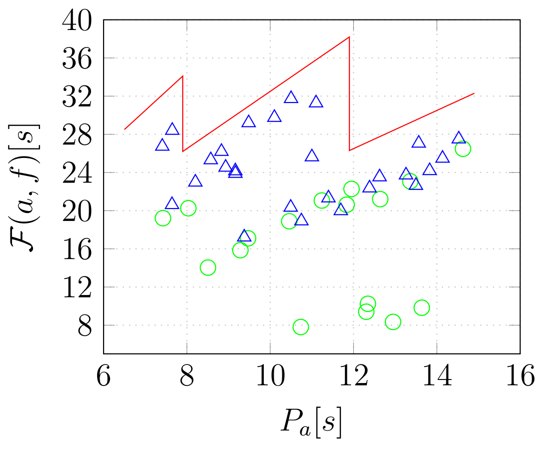
For our experiments, we use a setup consisting of 3 Rasperry-Pi 4B devices, with 8 GB RAM and a quad-core Cortex-A72 with a maximum frequency of 1.5 GHz. We chose a YOLOv3-tiny neural network implementation as our test application, simplifying the object detection structure described in [82]. The neural network runs on Tensorflow-CPU as a backend and uses the Keras Application Programming Interface (API). We split the neural network between the layers into five different tasks, with extra tasks for pre- and postprocessing adding up to 7 tasks in total. By splitting the neural network and introducing it into our framework, a pipeline for the application arises, which improves the overall throughput of the neural network by adding more computational power. We use a star-topology network with a switch and Transmission Control Protocol (TCP) Ethernet connections to communicate between the devices. Our analysis can also be applied to more complex architectures as long as the latencies for the message routings can be determined, which we assume is given for our approach. The Raspberry-Pis uses a service-oriented communication middleware based on the SOME/IP standard with a publish/subscribe pattern [36]. Our framework allows us to simulate ECU failures by shutting down the framework instances on a Raspberry Pi. Failures are detected via heartbeats and timeouts.

### 3.4.2 Experiments

For our experiments we obtain the application failover time $\mathcal{F}(a, f)$ by measuring the output delay $\Delta O(a, f)$ during a failover, while the tolerated output time interval $\Delta O_{tol}(a)$ can be calculated. Figure 3.2 presents the failover time of multiple experiments with a randomly varying application period $P_a$ of a single task failure scenario (Figure 3.2a) and a multi-task failure scenario (Figure 3.2b). Experiments with a random failure time are marked with a green circle. Furthermore, we provoked worst-case scenarios by shutting down the ECU at the worst-case point in time, where results are marked with blue triangles. Some factors that cause the failover time to fluctuate are the varying recovery time and application latencies. A negative failover time implies that the failover occurred within the tolerated output interval $\Delta O_{tol}(a)$, so no negative impact on the application timing behavior is observable. We used our worst-case analysis from Section 3.3 to obtain a strict upper bound plotted as the red curve.



**(a)** Failover scenario with failure of $t_2$.



**(b)** Failover scenario with failures of $t_2$ and $t_7$.

**Figure 3.2:** Experimental results with $\mathcal{L}_{wc}(a) = 16.5s$, $\mathcal{L}_{wc}(a, f) = 19s$, $\tau_r(t, f) = 7.5s$. Measurements with a random failure time are marked with green circles, while the worst-case failover measurements are marked with blue triangles. The red curve representing our worst-case analysis is a strict bound that is not exceeded by any measurement. The results are as close as 6.0% below our analytically derived exact bound.

Most importantly, we can observe that none of the measurements exceed the upper bound.

Our randomly generated worst-case results are as close as 6.0% below our analytically derived exact bound. Note that by conducting more experiments, the measurements would come even closer. Furthermore, it can be observed that the blue worst-case measurements also result in higher failover times than the random measurements marked in green. Comparing Figure 3.2a with Figure 3.2b, we can examine that a failure, where multiple tasks are affected, overall leads to a higher failover time as intuitively expected. Another interesting examination is that the upper bound builds a sawtooth-like curve with a decreasing slope with every step. With an increasing period $P_a$, the worst-case failover time increases linearly with a slope determined by the total number of lost iterations as described in Equation 3.5. However, as the system gains more time to recover with an increasing $P_a$, it will at one point lose one iteration less, causing a step and a decreased slope. Consequently, a slight period offset close to a step might significantly impact the worst-case failover time.

Overall, the results show that our prediction of the worst-case failover time has adequately estimated an upper bound for the failover time experienced by the system.

## 3.5 Conclusion

In this chapter we have introduced a formal analysis to derive the worst-case failover time for fail-operational distributed automotive applications. Our upper bound allows us to conduct an automated worst-case analysis to evaluate mappings at run-time. This way, unfeasible mappings can be identified and excluded such that a safe operation of safety-critical software within the bounds of the fault-tolerant timing interval can be guaranteed. We conducted experiments with a distributed fail-operational neural network on our hardware setup using an agent-based software platform, where we provoked worst-case failure scenarios to measure the failover time. Our randomly generated worst-case results are as close as 6.0% below our analytically derived exact bound.

Future work could extend our analysis for applications with parallel paths in the application graph. Furthermore, an agent-based mapping mechanism could conduct automated mapping decisions at run-time using the worst-case failover analysis. Summarized, our presented worst-case failover timing analysis allows us to perform an automated analysis at run-time to verify that the system operates within the bounds of the failover timing constraint such that dynamic and safe behavior of autonomous systems can be ensured.

# 4 Checkpointing Period Optimization of Distributed Fail-Operational Automotive Applications [5]

## Contents

## 4.1 Introduction

In previous chapters, we assumed that most applications are stateless without worrying about the loss of information in the case of an ECU failure. However, a significant challenge is that many applications have a state that might get lost during a failure such that a recovery might be impossible. Thus, in a system with passive redundancy, periodic checkpointing with rollback recovery can send the state to another ECU, where the application can be restarted after a failure. Here, the challenge is to find a suitable checkpointing period such that application-specific constraints on the state data age are met and the network and processing overhead caused by sending the checkpoint is minimized. However, as it might take the system some time to recover from the failure,

[5]Major parts of this chapter have been published in [31].

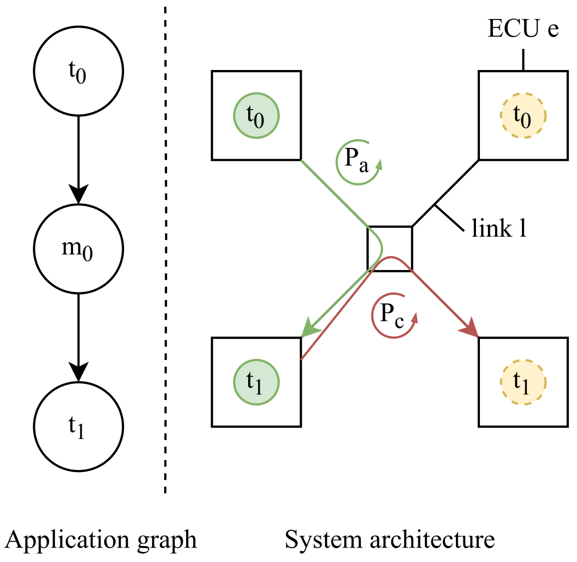


**Figure 4.1:** Representation of our system model with an example application and system architecture. The green circles indicate the active bindings of tasks $t_0$ and $t_1$, while the yellow circles indicate the passive task bindings. The message $m_0$ is sent periodically with the period $P_a$. A checkpoint is sent from the active task instance of $t_1$ to its passive task instance with the checkpointing period $P_c$.

the state data also ages during the downtime such that the worst-case state data age can not only be determined by the checkpointing period.

As the implications of the state data age are application-specific, we use a SLAM algorithm, an application commonly used in autonomous systems such as robots or self-driving cars, as a real-world example to determine the effects on the quality of the application. Using the SLAM algorithm, a moving object creates a global map of its current environment and uses this map to navigate or deduce its position and orientation. We refer to automotive terminology in the following, although the work also applies to other safety-critical autonomous systems. The approach presented in this chapter can be used to determine the maximum checkpointing period at which a safe operation within the bounds of the maximum allowed state data age is still possible. Therefore, we make the following contributions:

- We analyze related work in the field of checkpointing approaches in Section 4.2.
- We present our checkpointing approach in Section 4.3. Here, we formally introduce the checkpointing period and the achievable overhead reduction. By identifying components that influence a task's data age, we derive an upper bound for the worst-case data age. Using a state data age constraint, we derive the maximum possible checkpointing period that minimizes our network and processing overhead.
- In a case study, we show the dependency between state data age and accuracy of a SLAM algorithm as a real-world example in Section 4.4. Furthermore, we

show the applicability of our formal analysis by conducting failover experiments on our demonstrator setup with an agent-based software platform using the fail-operational distributed SLAM algorithm. The worst-case results of our randomized experiments are consistent with our analytically derived exact bound.

## 4.2 Related Work

Related work in the fields of predictable timing analysis has been covered in Section 2.4.1, and related work in the field of fail-operational systems, graceful degradation, and dynamic mapping has been covered in Section 2.3.1. The following provides a literature overview covering task migration and checkpointing approaches.

The authors in [40] propose a predictable task migration mechanism by implementing a migration timing analysis and a feasibility test for real-time applications. Here, the goal is to enable dynamic resource management to adapt the mapping of tasks at run-time. Our work does not migrate the tasks to optimize the mapping but periodically sends a checkpoint containing state information.

The study presented in [83] introduces a lightweight architecture to minimize excessive redundancy and tackle the problem of establishing a verifiable, fail-safe safety implementation for trajectory planning. In contrast to their example, which involves low-level prediction models applied to a real-world scenario, our work showcases the integration of an Extended Kalman Filter (EKF)-SLAM algorithm within our platform.

In [56], the authors explore the concept of dynamical reconfiguration and propose a hardware extension integrated into the architecture to ensure that the system retains its state during communication with peripherals. However, their approach does not encompass restarting entire backup tasks on other ECUs. In contrast, our work operates under the assumption that applications can withstand variations in the age of their state data, allowing for the adjustment and optimization of the checkpointing period.

In [84], a solution is presented to minimize checkpoint overhead by adapting the checkpoint interval based on failure probability rather than employing fixed periodic checkpoints. Simultaneously, in [85], authors propose reducing message losses during server failures by optimizing checkpointing, rollback, and overall time overheads associated with fixed checkpoint intervals. However, neither of these methods includes checkpoint interval optimization based on worst-case recovery time analysis. In contrast, our approach ensures that the maximum permissible data age of checkpointed data is never exceeded and optimizes the checkpoint process accordingly. Violating this data age constraint could result in unsafe system behavior.

In the work presented by [86], an approach is introduced for synthesizing fault-tolerant hard real-time systems tailored to safety-critical applications. This approach leverages checkpointing with rollback recovery and active replication as fundamental techniques. Nevertheless, it does not consider altering the checkpoint intervals, an aspect where our method could be complementary.

In summary, although the literature has considered alternating the checkpointing period, none has used a worst-case analysis of the failure and recovery time to obtain an optimal checkpointing period. Thus, existing approaches are unsuitable to ensure a fail-operational behavior within safe bounds when optimizing the checkpointing period.

## 4.3 Checkpoint Optimization

Since distributed systems are susceptible to failure, techniques to add reliability and high availability were developed. Checkpoints are a technique to ensure the fail-operational behavior of safety-critical applications in autonomous vehicles. By sending checkpoints over the network in distributed systems, states of the process can be saved during the failure-free execution. These checkpoints can be used after a failure and reloaded on another ECU to continue the execution of safety-critical tasks. Restarting the computation of tasks from an older saved state is called rollback recovery. A system recovers correctly when its internal state is consistent with its observable behavior before the failure [87]. In the following, we use the notation $P_c(t)$ to describe the checkpointing period of a task $t$ that periodically transfers its state from an active task instance to a passive task instance residing on another ECU. Figure 4.1 depicts the application and system model with an exemplary application consisting of a task chain with two tasks. We make the same assumptions for the remaining system model as introduced in Section 3.2.

The proper checkpoint period is a trade-off between network and processing overload and fail-safe recovery. Generating a checkpoint after every computational step ensures a higher probability of recovering from a failure properly. On the other hand, this would not only cause an overload of the network by forwarding the checkpoints to the passive tasks scheduled on the associated ECUs but also occupy processing resources. Therefore, increasing the checkpointing period is desirable to minimize both network and processing overhead. To obtain the maximum achievable checkpointing period $P_{c,max}(t)$, we are first introducing a formal definition of the checkpointing period and the possible overhead reduction in Subsection 4.3.1. Afterwards, we present the data age in Subsection 4.3.2. By analyzing components that influence the data age, we derive an analytical bound for the worst-case data age in Subsection 4.3.3. Based on this upper bound, we present the formula to derive the maximum checkpointing period $P_{c,max}(t)$ in Subsection 4.3.4.

### 4.3.1 Checkpointing Period

We define the default checkpointing period as $P_{c,d} = P_a$, in which case a checkpoint is taken after every computation and sent to a passive backup task. Furthermore, we define that the checkpointing period $P_c$ can only be a multiple $n \in \mathbb{N}$ of the default checkpointing period $P_c = n \cdot P_{c,d}$, which corresponds to a checkpoint taken after each $n$-th computation. Increasing the checkpointing period as much as possible would minimize the computational and networking effort spent taking and sending the checkpoint to the passive backup task. Increasing the checkpointing period by the factor $n$ generally results in an overhead reduction of $\frac{n-1}{n}$ as the checkpoint is only sent every $n$-th application period $P_a$. As an example, when increasing the default checkpointing period from $P_c =$

$P_{c,d}$ to $P_c = 5 \cdot P_{c,d}$, a reduction of 80% of the networking and processing overhead caused by checkpointing can be achieved.

## 4.3.2 Data Age

However, the checkpointing period can not be increased indefinitely; otherwise, no checkpoint would be required. If a failure occurs in the system and a passive backup task starts computing with the checkpoint data, the state data has already aged. We define the data age $d(t) \in \mathbb{N}$ as the number of computational steps that the data is behind when computing the subsequent output. Under normal operating conditions, a task would compute the following result using its internal state data, updated after every execution such that $d(t) = 1$. Using a checkpoint instead, the data is, in the worst case, behind by $n$ computational steps, resulting in a data age of $d(t) = n$.

## 4.3.3 Worst-Case Data Age

The state data age $d(t)$ can increase with an increasing checkpointing period due to failure and recovery time effects. After the failure of an active task, the passive backup task can not immediately start the following computation. The failure detection takes time, and the task might have to perform recovery steps until it can continue computing as usual. Our system setup uses heartbeats combined with timeouts to detect failures. Here, we assume that the worst-case failure detection time $\tau_d$ is equal for every ECU failure. For the remaining recovery, the tasks might have to re-publish their data and re-subscribe to preceding tasks in the application tree using the service-oriented middleware. An upper bound for the detection or subscription time can be found using a composable system, which is well-described in related work [17, 74]. In the following, we refer to $\tau_r(t, f)$ as the worst-case recovery time that it takes from the occurrence of a failure $f$ until a task $t$ is ready to receive and process the following input after recovery, which includes the worst-case failure detection time and publish and subscription times.

During the downtime of a task represented by the recovery time, it might miss one or multiple periodic inputs. In the worst case, the active task instance just finished processing but cannot send out the checkpoint in time, which causes the data at the passive task instance to be behind by an additional computational step. Afterward, the passive task instance might take some time to recover until it can receive and process the following input. During this time, it might miss additional inputs such that the data ages even further. In the worst case, at least one additional computational step will always be missed when the active task instance has just finished the execution and is about to send the next checkpoint. In chapter 3, we presented an analysis to derive the worst-case failover time for distributed applications. We can re-use the upper bound $N_t(a, f)$ for the missed computational steps for a scenario where multiple tasks of an application $a$ are affected at once by the failure $f$ of an ECU, which is calculated as

$$N_t(a, f) = \lfloor \frac{\tau_r(t, f) + \mathcal{L}_{wc}(t_l) - \mathcal{L}_{bc}(p(t_f))}{P_a} \rfloor + 1. \tag{4.1}$$

This bound considers the worst-case recovery time $\tau(r, f)$, but also the worst-case latencies $\mathcal{L}_{wc}(t_l)$ and best-case latencies $\mathcal{L}_{bc}(p(t_f))$ of latencies of the affected tasks. The formula also directly reflects that losing at least one iteration is unavoidable in the worst case. By repurposing this analytical bound, we can calculate the worst-case data age of a task after a failover as follows

$$d_{wc}(t) = \frac{P_c}{P_{c,d}} + N_t(a, f) = n + N_t(a, f). \tag{4.2}$$

We are using this formula to evaluate the effect of the checkpointing period and the application period on the data age of a SLAM application in Subsection 4.4.3. We can observe that the $n$ linearly influences the worst-case data age, while $N_t(a, f)$ is an offset for a given system architecture and task mappings.

### 4.3.4 Maximum Checkpointing Period

The question that arises and which plays a significant role in finding an optimal checkpointing period is which worst-case data age $d_{wc}(t)$ can be tolerated by a task $t$. In the following, we refer to $d_{max}(t)$ as the maximum data age that a task $t$ can tolerate. This maximum data age is highly application-dependent and has to be found individually for each application and task, as it directly influences the functionality of the underlying algorithms. We will derive such an exemplary data age constraint $d_{max}(t)$ in our case study using application-specific metrics in Subsection 4.4.2.

Using the number of computational steps $N_t(a, f)$ that will be missed in the worst-case due to the failure and recovery effects together with a data age constraint $d_{max}(t)$, we can obtain the maximum achievable checkpointing period $P_{c,max}(t)$, which is the goal of this work, as

$$P_{c,max}(t) = (d_{max}(t) - N_t(a, f)) \cdot P_{c,d}. \tag{4.3}$$

This period minimizes the network and processing overhead caused by the checkpoint messages as it exhausts the available data age limit $d_{max}(t)$ under consideration of the failure and recovery time effects. It can be observed that an increase in the checkpointing period is only possible if the tasks can tolerate the missed computational steps due to the effects of failure and recovery time. In case $d_{max}(t) < N_t(a, f)$, the system setup would already violate the data age constraint with the default checkpointing period. Therefore, fast failure detection and recovery are critical when optimizing the checkpointing period. We are using Equation 4.3 to calculate the maximum achievable checkpointing period on our demonstrator setup in Subsection 4.4.3 for the data age constraint $d_{max}(t)$ obtained in Subsection 4.4.2.

## 4.4 Case Study: SLAM Application

To evaluate the applicability of our checkpoint optimization approach introduced in Section 4.3, we present a case study using a SLAM application as a representative real-world application from the domain of the autonomous system. Determining a suitable upper bound for the data age is always application-dependent. It should also consider the impact on the safety of the application, e.g., by assuming application-specific metrics. We first introduce the SLAM application in Subsection 4.4.1 and present the effect of a successful and a faulty recovery. Afterward, we examine the dependency between state data age and accuracy by defining our metrics in Subsection 4.4.2. Using a quality analysis, we define an exemplary upper bound for the data age $d_{max}(t)$ that should not be exceeded. Furthermore, we perform failover experiments and calculate the corresponding worst-case data age $d_{wc}(t)$ as an upper bound in Subsection 4.4.3. Using $d_{max}(t)$ as a constraint, we analytically derive the maximum achievable checkpointing period $P_{c,max}(t)$ on our demonstrator setup.

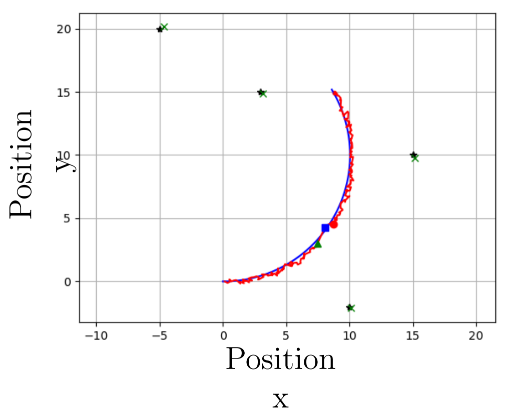
### 4.4.1 SLAM Application

SLAM is a fundamental robotics and autonomous systems. Its primary goal is to enable a robot or device to explore an unknown environment while constructing a map of that environment and determining its position within that map, all in real-time.

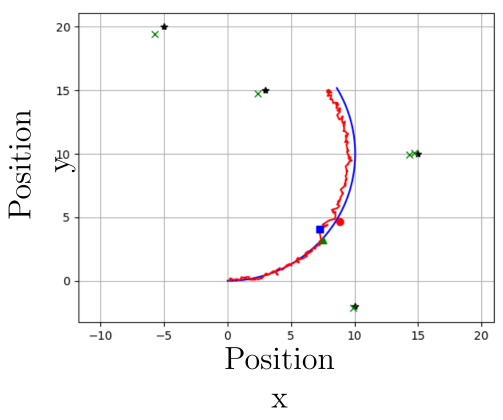
A SLAM algorithm typically comprises two fundamental steps: a prediction step, driven by a motion model, and an update step, guided by an observation model. These steps work in tandem to continually refine the car's estimate of its location and the map of its environment. This learning process involves iteratively reconciling the car's state with its sensor measurements, resulting in an increasingly accurate representation of the environment. One of the most common learning methods employed in SLAM is the Kalman Filter. The Kalman Filter assumes a uni-modal distribution, which models the state and measurements based on a single, most likely estimate. In our work, we utilize a simulation of an EKF SLAM application, which is available in the repository [88].

Figure 4.2a presents simulation results that compare the desired trajectory (depicted in blue) with the trajectory calculated by the EKF-SLAM approach (shown in red). The simulation progresses in discrete time steps denoted by $\tau_{step}$. Additionally, it demonstrates a successful recovery process, where the green triangle signifies the last checkpoint recorded before a failure occurs. In this representation, the red data point marks the moment of the failure, and the blue square signifies the first calculation after the recovery using the old checkpoint data. Notably, the initial data point after recovery estimates a position further behind the actual position. However, the red curve rapidly converges to regular operation as subsequent operations unfold. This behavior highlights the algorithm's capability to recover and re-establish accurate positioning after a temporary deviation caused by a failure event.

Figure 4.2b illustrates an example of a faulty recovery scenario in which a landmark is re-detected after the recovery process. The algorithm uses a threshold to distinguish points in the environment. During a recovery event following a failure, previously de-



(a) With an ideal recovery, no large impact on the deviation of the estimated trajectory to the real trajectory can be observed.



(b) A faulty re-detection of an already recognized landmark after recovery leads to a high deviation of the estimated trajectory to the real trajectory.

**Figure 4.2:** Ideal and non-ideal recovery after a failure in a simulation run with the EKF-SLAM algorithm. The blue curve is the real trajectory starting at (0,0), while the red curve is the trajectory estimated by the EKF-SLAM algorithm. Black stars represent accurate obstacle positions, while green crosses represent the estimated object position. The green triangle corresponds to the last checkpoint taken before the failure, the red circle to the failure time, and the blue square to the first computation after recovery using the checkpoint as the last recorded position. The first computation after recovery has a more significant deviation, so the algorithm jumps a bit backward.

tected landmarks may end up lying beyond this threshold radius, resulting in a false re-detection as a new landmark. This false re-detection can lead to a persistent deviation from the actual trajectory. In the example given, the algorithm incorrectly believes that there are two obstacles at position (15, 30) instead of one. This situation arises when the recovered position differs significantly from the actual position, causing a temporal gap between the car's position derived from the landmarks and the previously saved position.

### 4.4.2 Quality Analysis

We introduce two metrics to assess how data age affects the algorithm's quality. The first metric, labeled $e_i$, measures the Euclidean distance between computed points on the two curves at computational step $i$. We can evaluate the algorithm's performance before and after a simulated failure by averaging these errors. To establish these metrics, we adopt the following definitions:

$$\bar{e}^{(bf)} = \frac{1}{n}\sum_{i=1}^{n} e_i^{(bf)}, \tag{4.4}$$

$$\bar{e}^{(af)} = \frac{1}{n}\sum_{i=1}^{n} e_i^{(af)}, \tag{4.5}$$

$$e_{wc}^{(af)} = \max_{\forall i \in [1 \dots n]} e_i^{(af)}, \tag{4.6}$$

with $\bar{e}^{(bf)}$ being the average error before the failure, $\bar{e}^{(af)}$ being the average error after the failure and $e_{wc}^{(af)}$ being the worst-case error after the failure. We define $\Delta\bar{e}$ as the difference of the average of all calculated points before $\bar{e}^{(bf)}$ and after $\bar{e}^{(af)}$ a simulated failure event has occurred::

$$\Delta\bar{e} = \bar{e}^{(af)} - \bar{e}^{(bf)}. \tag{4.7}$$

We define $\Delta e_{wc}$ as the worst-case deviation $e_{wc}^{(af)}$ compared to the average $\bar{e}^{(bf)}$ as follows

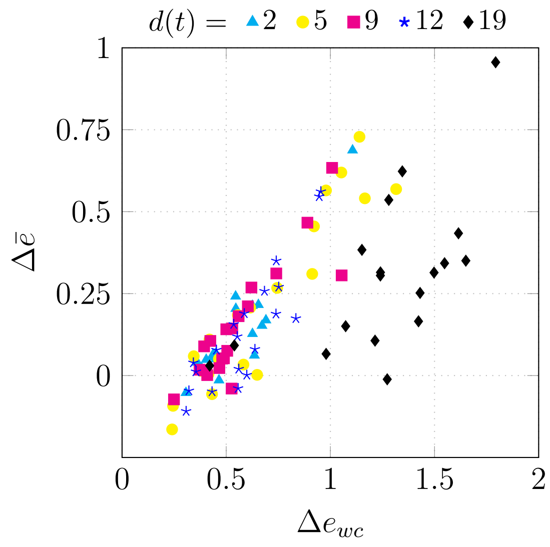
$$\Delta e_{wc} = e_{wc}^{(af)} - \bar{e}^{(bf)}. \tag{4.8}$$

The data presented in Figure 4.3 is derived from a simulation with a fixed discrete time step of $\tau_{step} = 2.0s$ and stationary landmark positions. To assess the influence of data age, experiments were conducted with data ages set to 2, 5, 9, 12, or 19. Smaller values indicate that the checkpoint was taken more recently before a failure occurred, allowing the use of more recent data for position calculation at the recovery point. Some data points with a data age of 19 deviate significantly and are not shown in the plot.

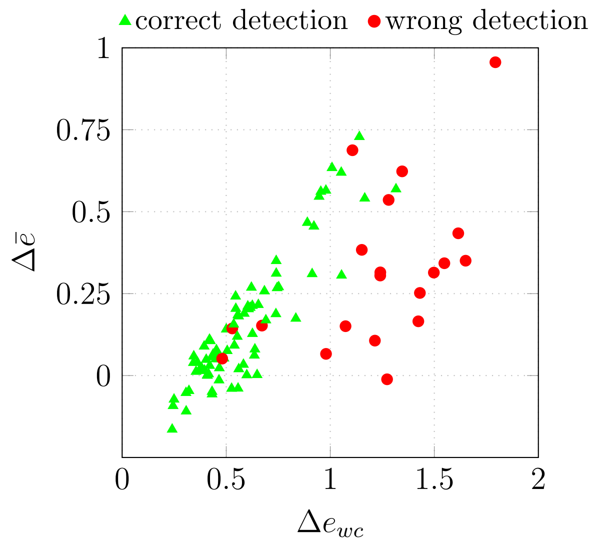
Subfigure 4.3b distinguishes between simulation runs where successful recovery was achieved and runs where a landmark was erroneously detected. Older checkpoints lead to larger average errors and notably more significant worst-case errors. Taking into account the impact of data age on both the average deviation $\Delta\bar{e}$, the worst-case deviation $\Delta e_{wc}$, and the number of incorrect detections, we establish a data age limit of $d_{max}(t) = 12$ as a representative upper bound with acceptable deviations. In the following subsection, we utilize the EKF-SLAM application presented here to conduct failover experiments, demonstrating the applicability of the analysis from Section 4.3. Additionally, we employ the upper bound $d_{max}(t) = 12$ as an example to determine the maximum achievable checkpointing period.

## 4.4.3 Failover Experiments

We use 3 Raspberry-Pi 4B devices for our failover experiments, with 8 GB RAM and a quad-core Cortex-A72 with a maximum frequency of 1.5 GHz. We use a star-topology network with a switch and Ethernet links to communicate between the devices. Our framework on each Raspberry Pi uses a service-oriented communication middleware with a publish/subscribe pattern. It simulates ECU failures by shutting down the framework instances. Failures are detected via heartbeats and timeouts. Passive task instances



**(a)** Simulation results for the data ages 2, 5, 9, 13 and 19 showing the average deviation $\Delta\bar{e}$ and the worst-case deviation $\Delta e_{wc}$.



**(b)** Red marks correspond to a faulty landmark detection, while green marks indicate that all landmarks have been detected correctly.

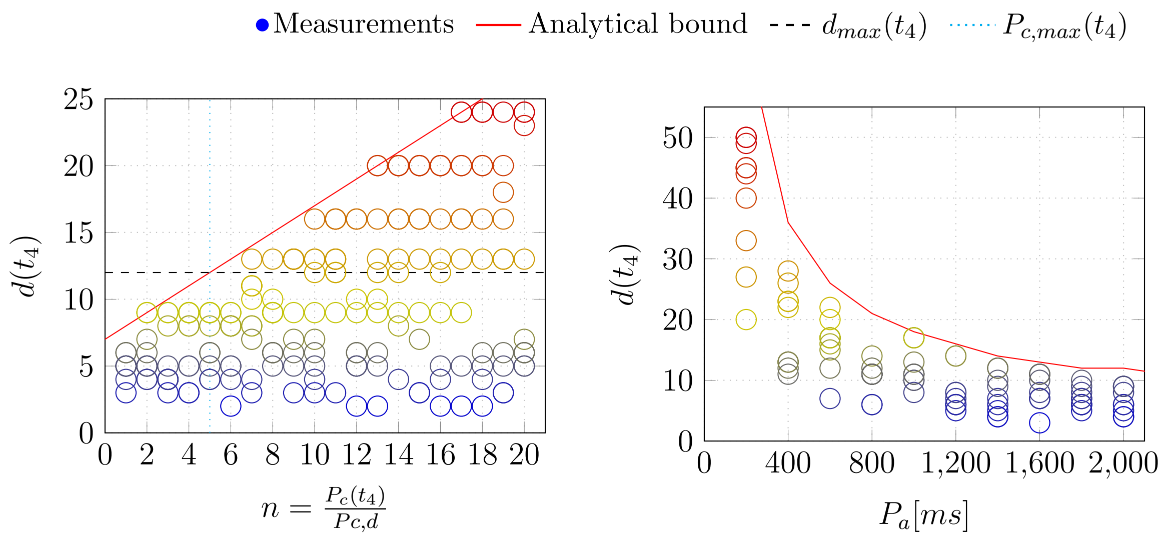
**Figure 4.3:** The simulated failover results for the EKF-SLAM application are obtained using a discrete time step $\tau_{step} = 2.0s$ with varying data ages. In most cases, the data points corresponding to data ages 2, 5, 9, and 12 tend to cluster closely together. However, the data points associated with a data age of 19 stand out distinctly, having a direct negative impact on both the average deviation $\Delta\bar{e}$ and the worst-case deviation $\Delta e_{wc}$. Additionally, the probability of incorrectly detecting landmarks experiences a significant increase in this scenario.

subscribe to the checkpoints of active task instances. If an active task instance is affected by a failure, the passive task instance is restarted using the last successfully transmitted checkpoint. We divided the complete application into four tasks:

- $t_1$: Simulation task
- $t_2$: Pre-processing task
- $t_3$: Observation task
- $t_4$: EKF-SLAM algorithm task

For the failover measurements presented in Figure 4.4, the tasks $t_1$, $t_2$, and $t_3$ are active on $e_1$ and $t_4$ are active on $e_2$, while all passive tasks are mapped to $e_3$. We shut down $e_2$ at random time points during the experiments and measured the data age $d(t_4)$ after the recovery process . This allowed us to assess the effectiveness of the failover mechanism and its impact on the age of the data associated with task $t_4$.

Figure 4.4a presents the data age obtained by our experiments with an application period of $P_a = 2s$ and a varying checkpoint period $P_c(t_4)$. We carried out ten measurements for each variable setup of $P_c(t_4)$. The red curve corresponds to the upper



**(a)** Results with an application period $P_a = 2s$ and a varying checkpointing period $P_c(t_4)$. For a given data age constraint, the maximum checkpointing period can be obtained graphically by finding the intersection point with the red curve. The black dashed line marks the data age constraint $d_{max}(t_4)$ obtained through the quality analysis in Subsection 4.4.2. The cyan dotted line marks the corresponding maximum checkpointing period $P_{c,max}(t_4)$ obtained through Equation 4.3.

**(b)** Results for a fixed checkpointing period of $P_c(t_4) = 5 \cdot P_{c,d}$ and a varying application period $P_a$.

**Figure 4.4:** Results of the random failover experiments presenting the data age $d(t_4)$ at the backup task instance of $t_4$ after successful recovery with $\tau_r(t, f) = 11.82s$, $\mathcal{L}_{bc}(p(t_4)) = 0.0548s$ and $\mathcal{L}_{wc}(t_4) = 0.238s$. None of the measurements exceed the upper bound obtained through Equation 4.2, here marked as the red curve. The worst-case results of our experiments are consistent with our analytically derived exact bound, as some measurements lie directly on the curve.

bound $d_{wc}(t_4)$ obtained through Equation 4.2 for a given checkpointing period $P_c(t_4)$. For the experiments depicted in Figure 4.4b, a fixed checkpoint period of $P_c = 5 \cdot P_{c,d}$ was selected, while the application period $P_a$ was varied.

Most importantly, it can be observed that none of the measurements exceed the upper bound obtained through Equation 4.2. Some worst-case measurements lie directly on the bound, so the experiments are consistent with our analytically derived exact bound. In Figure 4.4a, the worst-case data age increases linearly with the checkpointing period $P_c(t_4)$, while the worst-case number of lost iterations $N_t(a, f)$ adds a constant offset. In Figure 4.4b, the worst-case data age decreases exponentially with an increasing application period $P_a$ and an offset of the checkpointing period $P_c(t_4) = 5 \cdot \cdot P_{c,d}$. Once $P_a$ doubles, about half of the iterations will be lost as the worst-case recovery time remains the same. Note that we define the data age in iterations and not in absolute time, which could be obtained by multiplying the data age $d(t_4)$ with the application period $P_a$.

Following the quality analysis from Subsection 4.4.2, we use an exemplary data age constraint $d_{max}(t_4) = 12$, which is depicted as a black dashed line in Figure 4.4a. The resulting maximum achievable checkpointing period of $P_{c,max} = 5 \cdot P_{c,d}$ is marked with a green dotted line. By increasing the default checkpointing period from $P_c(t_4) = 1 \cdot P_{c,d}$ to $P_{c,max}(t_4) = 5 \cdot P_{c,d}$, the networking and processing overhead caused by checkpointing can already be reduced by 80% in this example.

In summary, our case study and experiments have confirmed the applicability of our approach. We have shown the application-specific dependency between state data age and the accuracy of an EKF-SLAM application. We analytically derived the maximum achievable checkpointing period using a data age constraint obtained by the quality analysis. By performing additional failover experiments, we have confirmed that the worst-case results of our randomized experiments are consistent with our analytically derived exact bound. To apply our approach to other applications, a safe upper bound for the data age has to be found by using application-specific metrics and safety considerations. This maximum data age constraint can be used for our application-independent analysis to find a suitable maximum achievable checkpointing period. Overall, when designing the checkpointing period, a trade-off between the impact on algorithmic behavior (in our example on the average deviation $\Delta\bar{e}$ and the worst-case deviation $\Delta e_{wc}$) and the overhead reduction has to be made. However, our approach might not apply to applications where the loss of any state can not be tolerated. An active redundant approach might be required in these scenarios, such that the state is continuously computed in parallel on the redundant component.

## 4.5 Conclusion

In this chapter, we have presented an analysis to calculate the maximum achievable checkpointing period for distributed fail-operational automotive applications. We analyzed the effects of the recovery time on the data age. We determined a worst-case number of computational steps that will be missed during the downtime to obtain the maximum possible checkpointing period. Furthermore, we presented a detailed case study of a SLAM application as a representative real-world application from the automotive domain. Here, we analyzed the dependency between state data age and accuracy and derived an exemplary state data age constraint. In addition, we conducted failover experiments on our demonstrator setup using an agent-based software platform to show the applicability of our worst-case analysis. The worst-case results of our randomized experiments are consistent with our analytically derived exact bound. In summary, our approach can be used to reduce network and processing overhead caused by sending checkpoints to achieve a cost-efficient and safe behavior of stateful distributed fail-operational applications.

# 5 Reliability Analysis of Gracefully Degrading Automotive Systems [6]

## Contents

## 5.1 Introduction

The main challenge for our graceful degradation approach, presented in Chapter 2, is achieving a predictable behavior. In previous chapters, we have addressed issues related to the predictability of timing behavior and the loss of state data. In this chapter, we focus on the reliability and the impact of graceful degradation on non-critical applications. The presented scheduling approach in Section 2.4 supports a gracefully degrading behavior such that resources of non-critical applications might be taken from re-starting critical applications. As in our gracefully degrading system, resources are shifted dynamically, and non-critical applications not directly affected by an ECU failure might get shut down to free resources for restarting critical tasks. Thus, with a graceful

[6]Major parts of this chapter have been published in [32].

degradation approach, the reliability of critical applications is increased at the cost of a decrease in the reliability of non-critical applications. Here, it is crucial to quantify and understand the effect of a graceful degradation approach on critical and non-critical applications to increase predictability.

Furthermore, it is important to compare graceful degradation to active redundancy based on reliability and resource consumption such that a reasonable trade-off can be made. There has not been any work in literature that analyzes the trade-off between additional resource consumption and impact on the reliability of graceful degradation or compares it to approaches such as active redundancy. Therefore, we make the following contributions:

- We present related work in the fields of reliability analysis in Section 5.2 and conclude that there is a lack of work on reliability analysis of graceful degradation approaches.
- After giving a brief overview of reliability analysis in Section 5.3, we introduce our approach to formally analyze the impact of graceful degradation on the reliability of critical and non-critical applications in Section 5.4. This approach considers that scheduling slots of non-critical applications can be reserved by critical applications, effectively reducing reliability.
- We perform experiments in our in-house developed simulation framework, simulating the agent-based approach on a virtual architecture in Section 5.5. Here, we present, for the first time, an in-depth trade-off analysis of a graceful degradation approach where we analyze the resource consumption and the impact of graceful degradation on the reliability of critical and non-critical applications. In our experiments, we evaluate our three allocation and reservation strategies and compare them to active redundancy. Additionally, we present experimental results of our *Predecessor Heuristic*, which aims to reduce the individual applications' exposure to failure sources.
- We conclude in Section 5.6 that graceful degradation can be a powerful methodology that reduces resource consumption compared to active redundancy while providing the same reliability to critical applications as an active redundancy approach. However, this is bought with reduced reliability of non-critical applications.

## 5.2 Related Work

While relevant work in the field of fail-operational systems and graceful degradation approaches has been covered in Section 2.3.1, we provide an overview of the literature in the field of reliability analysis in the following.

Numerous approaches to designing reliable embedded systems have been introduced in research. An overview of these methods is provided by the authors in [89].

Conventional reliability analysis approaches typically prioritize the design and implementation of the system initially, followed by conducting reliability analysis to ensure compliance with particular requirements. If the reliability criteria are not met, system redesign may be necessary, as described in reference [90]. Reliability can serve as a metric within an optimization problem, enabling the discovery of Pareto-optimized solutions that simultaneously address various design objectives, including aspects like area or performance, during the process of design space exploration. Solutions aimed at maximizing reliability during design are presented by the authors in [91], [92], and [93].

Reliability can be used as a metric in an optimization problem to find a Pareto-optimized solution and multiple other design objectives, such as area or performance, when performing a design space exploration. The authors in [91], [92], and [93] propose solutions to maximize reliability at design time. In [94], the author incorporates reliability as an optimization objective during system-level design. In this context, hardware redundancy is employed as a means to handle faults, although it notably leads to a substantial cost increase.

In [90], the authors introduce an automated method for system synthesis with a focus on reliability considerations. They present a multi-objective synthesis approach that takes into account various parameters, including area, costs, and reliability, enabling the generation of reliable embedded systems. This approach utilizes data flow and resource graphs to model different architectural implementations. Additionally, related research, as presented in [95] and [96] suggests a symbolic reliability analysis of self-healing networks with self-reconfiguration and routing. This proposed analytical solution takes performance and memory constraints into consideration and demonstrates the maximum achievable reliability metrics for a given system, specifically the MTTF.

While these approaches present a design time synthesis, our agent-based system finds feasible application mappings at run-time. Using reliability as an optimization objective directly in the mapping process in future work could be interesting. However, this would require finding optimized solutions at design time and then finding a feasible mapping at run-time as presented in hybrid mapping approaches such as in [74]. Instead of optimizing reliability, our work focuses on the reliability and resource consumption analysis of a graceful degradation approach.

In [66], the authors utilize reliability as a metric to optimize an embedded system that features multiple degradation modes. With multiple degradation modes, a constraint is imposed, requiring higher modes to possess greater reliability than lower ones. In contrast, our work offers an analysis of a system where mappings are determined dynamically during runtime, and individual applications are evaluated separately, rather than grouping them into degradation modes. Additionally, the authors in [66] do not provide an analysis of the resource consumption associated with their approach, making it challenging to draw conclusions about potential resource savings compared to an active redundancy approach. Nevertheless, this aspect is crucial when designing a system to strike the right balance between resource conservation and its impact on reliability.

Our base system differs from existing work, using an agent-based approach that finds application mappings at run-time. Here, applications can independently allocate and reserve resources. Our work examines the impact of graceful degradation on the reliability of a highly distributed system consisting of critical and non-critical applications. This has not been considered and analyzed in the literature yet.

## 5.3 Introduction to Reliability Analysis

In this section, we are giving a short introduction to reliability analysis. Afterward, we present our approach to formally analyze the impact of graceful degradation on the reliability of critical and non-critical applications. For a specified duration, a system's reliability, denoted as $R(\tau)$ represents the likelihood that the system can maintain continuous operation without encountering failures [97, 98].

The system's reliability is defined within a time interval spanning from zero to t, where t can represent any predetermined duration in the future, extending to infinity. The system's reliability is the complement of its failure rate, denoted as $\lambda(\tau)$:

$$\lambda(\tau) = 1 - R(\tau) \tag{5.1}$$

Reliability is frequently represented as a distribution function. Electronic components and systems typically follow an exponential distribution function for their failure rate, denoted as $\lambda(\tau)$. The reliability of an exponential system can be expressed as follows::

$$R(\tau) = e^{-\int_0^\tau \lambda(\theta) d\theta} \tag{5.2}$$

The failure rate $\lambda(\tau)$ of a sample of independent components can be represented with the bathtub curve [89]. In most cases, it is conventional to assume that the system is free from faults at the initial time, indicating that $\lambda(0) = 1$. This further reduces the reliability equation to:

$$R(\tau) \ = e^{-\lambda\tau} \tag{5.3}$$

For a given time $\tau$, the only factor we need to consider is the failure rate, which we denote as $\lambda$. In our system, each component (ECU) has a constant failure rate which we assume is identical for all components. In our experiments, we set the component failure rate to $\lambda = 0.01$.

Equation 5.3 represents the reliability distribution function for an individual ECU in our system. We use this reliability function to derive the MTTF which serves as a metric for assessing the reliability of both critical and non-critical applications. The MTTF signifies the average duration a component or system typically operates from its installation until a failure takes place. The MTTF can be calculated as:

$$MTTF(\tau) = e^{-\int_0^\infty \lambda(\theta) d\theta} = \frac{1}{\lambda(\tau)}. \tag{5.4}$$

## 5.4 Formal Reliability Analysis of Gracefully Degrading Systems

In the following, we assume a mapping process has been performed as introduced in Section 2.4, including the allocation and reservation. The mapping of the tasks and the information on which specific slots have been allocated and reserved for each task are required to determine the structure function $\varphi$ for each application. While the derivation of the structure function $\varphi$ is more straightforward for critical applications, it is more complex for non-critical applications as the degradation process also influences the reliability. We assume that an application is still operational as long as there is at least one task instance of each task available and can communicate.

### 5.4.1 Derivation of Structure Functions

To describe the behavior of our applications, we use boolean functions represented by a structure function $\varphi$, which is encoded in a Binary Decision Diagram (Binary Decision Diagram (BDD)) [90]. A BDD is a rooted, directed, and acyclic graph that consists of multiple decision nodes and two terminal nodes that determine the outcome of the boolean function [99]. The maximal size of a BDD encoding a formula on $n$ variables is $2^n/n$ [100]. As described below, the complexity of automatically generating our structure functions is linear in the number of tasks.

In a system with $n$ components, the state of each component $i$ can be encoded in a boolean variable $x_i$, which evaluates to 1 if the component is operational and 0 if the component is defective. A structure function $\varphi(x) = \varphi(x_1, x_2, ..., x_n)$, which represents the system's state, evaluates to 1 if the system is operational and 0 if not.

In the following, we denote $z : \mathcal{V} - > \{0, 1\}$ as a function, which translates the task instances into a binary variable with 1 indicating a proper operation of the task instance. We define that $\varphi(z) = \varphi(z(t_{0,a}), z(t_{0,b}), ...z(t_{n,a}), z(t_{n,b}))$ represents the structure function of an application $a$. To derive the reliability, we translate these boolean variables dependent on the state of the task instances into boolean variables only dependent on the state of the ECUs. Therefore, we denote $y : E - > \{0, 1\}$ as a function that translates ECUs into a binary variable with 1 indicating a proper operation of the ECU. Furthermore, we define $u : \mathcal{V} \to E$ as a function which translates a boolean variable $z(t)$ dependent on the state of a task instance $t$ to a boolean variable $y(e)$ which only depends on the state of ECU $e$, such that $u(z(t)) = y(e)$, where $\alpha(t) = e$ for active task instances and $\beta(t) = e$ for passive task instances. As an example, if a task instance $t_{0,a}$ is bound to an ECU $\alpha(t_{0,a}) = e_1$, the function $z(t_{0,a})$ only evaluates to 1 as long as the function $y(e_1)$ is indicating a proper operation. The resulting structure function $\varphi(u(z)) = \varphi(y) = \varphi(y(e_0), ...y(e_n))$ is then only dependent on a set of independent variables indicating the status of the ECUs.

### 5.4.2 Derivation of Reliability

In our experiments, we use the JRELIABILITY framework to evaluate the MTTF [99]. Here, a Shannon-decomposition-based algorithm is applied to the structure function to calculate the reliability [90]. The complexity of this algorithm is linear in the size of the BDD [100].

A BDD allows us to calculate the probability of the root event of the tree using Shannon's decomposition if the probabilities of the leaves are given as follows:

$$p_f(g) = p_f(x = 1) \cdot p_f(g_{x=1}) + p_f(x = 0) \cdot p_f(g_{x=0}), \tag{5.5}$$

with $p_f$ being the probability that is calculated, $g$ being a function and $x$ a variable occurring in $g$ [100].

Using the probability function $p_f(\varphi, \tau)$ as the probability $p$ being calculated and the structure function $\varphi$ as the function $f$ being evaluated, the likelihood of a working system can be calculated as [101]:

$$p_f(\varphi, \tau) = p_f(y(e) = 1, \tau) \cdot p_f(\varphi_{y(e)=1}, \tau) + p_f(y(e) = 0, \tau) \cdot p_f(\varphi_{y(e)=0}, \tau). \tag{5.6}$$

With a homogeneous set of ECUs and the assumption that the same reliability function $R(\varphi)$ is applied to each ECU, this can be further simplified to:

$$p_f(\varphi, \tau) = R(\tau) \cdot p_f(\varphi_{y(e)=1}, \tau) + (1 - R(\tau)) \cdot p_f(\varphi_{y(e)=0}, \tau), \tag{5.7}$$

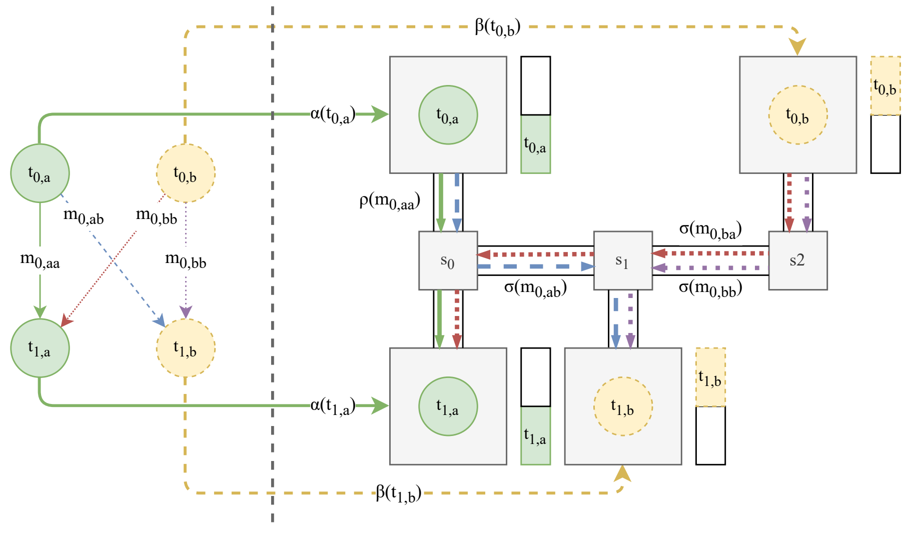
resulting in the desired reliability function $R(\varphi, \tau) = p_f(\varphi, \tau)$ describing the probability of a proper working application. Finally, using equations 5.3 and 5.4, the MTTF can be derived from $R(\varphi, \tau)$.

### 5.4.3 Formal Reliability Analysis of Critical Applications

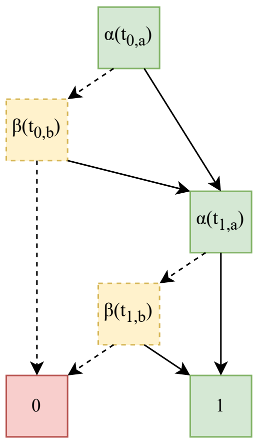
Critical applications in our system model are required to have double redundancy which means every active task instance has a passive counterpart. As the active task instance and passive tasks instances are mapped onto different ECUs, a critical application would always remain operational in the event of a single ECU failure. However, if two or more ECUs had a critical failure and both task instances of a task are affected by it, then the critical application fails.

For example, a critical application consisting of two active and two passive task instances is mapped onto four ECUs in Subfigure 5.1a. One slot has been allocated or reserved for the active and passive task instances. If both the active and passive task instances $t_{0,a}$ and $t_{0,b}$ or $t_{1,a}$ and $t_{1,b}$ are affected by ECU failures then the application would be considered as failed, resulting in the structure function:

$$\varphi(z) = (z(t_{0,a}) \vee z(t_{0,b})) \wedge (z(t_{0,a}) \vee z(t_{1,b})). \tag{5.8}$$



**(a)** Mapping example.



**(b)** BDD.

**Figure 5.1:** An example mapping of a critical application onto four ECUs and the corresponding BDD. The slot allocation and reservation are indicated right of the ECUs. The BDD represents the boolean structure function $\varphi(y) = (y(\alpha(t_{0,a})) \vee y(\beta(t_{0,b}))) \wedge (y(\alpha(t_{0,a})) \vee y(\beta(t_{1,b})))$. If both task instances of any task were affected by ECU failures, then the application would fail.

The resulting structure function $\varphi(y)$ dependent on the operational status of the ECUs is:

$$\varphi(y) = (y(\alpha(t_{0,a})) \vee y(\beta(t_{0,b}))) \wedge (y(\alpha(t_{0,a})) \vee y(\beta(t_{1,b}))) \tag{5.9}$$

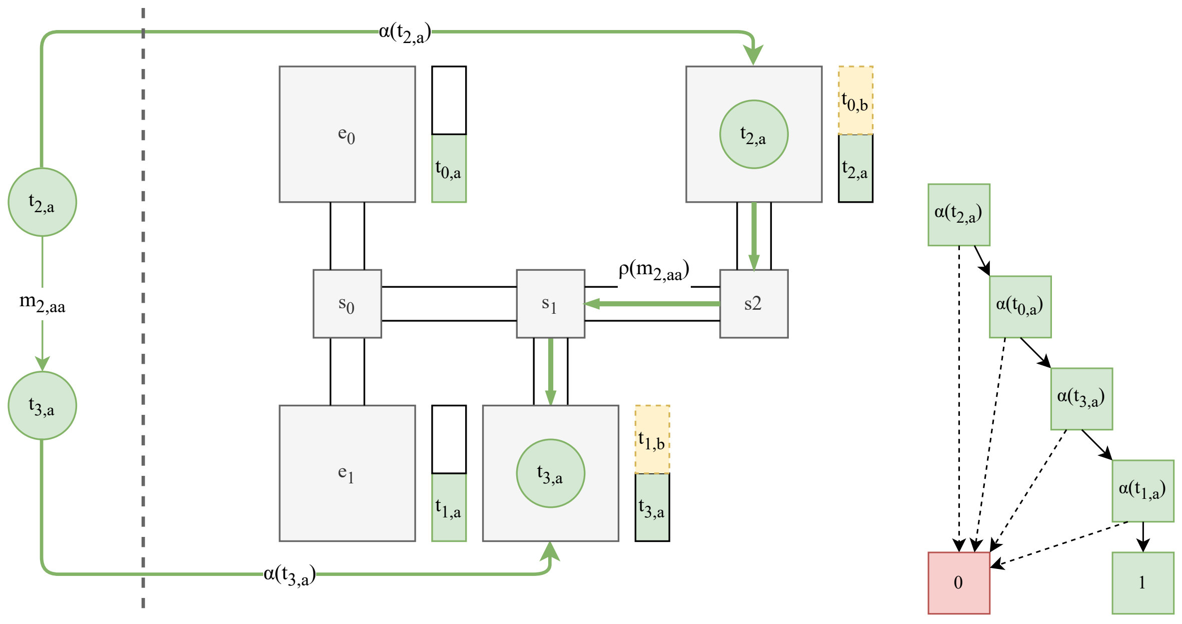
Subfigure 5.1b shows the corresponding BDD, with all possible paths leading either to a failure (0) or to success (1). Each decision node in the BDD has two outgoing edges that correspond to the variable being 0 (dashed arrow) or 1 (normal arrow). Each variable assignment that results in 1 means that the application is still operational, while the paths leading to 0 represent a failed application.

Generalizing the structure function $\varphi(z)$ for critical applications, if for all tasks $t \in \mathcal{V}$ either the active task instance $t_a$ or the the passive task instance $t_b$ is working, then the critical application is operational:

$$\varphi(z) = \bigwedge_{t \in \mathcal{V}} z(t_a) \vee z(t_b). \tag{5.10}$$

### 5.4.4 Formal Reliability Analysis of Non-Critical Applications

As we assume that no redundancy is used, a non-critical application is not operational anymore if any of the tasks is affected by a failure. However, not only a direct ECU failure can lead to an application failure but also degradation effects. Suppose a slot allocated by a non-critical task is reserved by a critical application's passive task instance of a critical application. In that case, a degradation occurs if that passive task instance is activated.



**(a)** Mapping example. **(b)** BDD.

**Figure 5.2:** An example mapping of a non-critical application onto the system and the corresponding BDD. The slot allocation and reservation are indicated right of the ECUs. The passive task instances of the critical example from the previous example also reserve the slots allocated by the task instances. The BDD represents the boolean structure function $y(\alpha(t_{2,a})) \wedge y(\alpha(t_{0,a})) \wedge y(\alpha(t_{3,a})) \wedge y(\alpha(t_{1,a}))$. The non-critical application would immediately fail if any task instances were affected by an ECU failure. Furthermore, a degradation effect has to be considered as the slots of the task instances are reserved by passive task instances of the critical application, If any of the active task instances of the critical applications were affected by a failure, the corresponding passive task instance would get activated, and the task instance of the non-critical application would be degraded.

Activation of this passive task instance would only happen if the corresponding active task instance were affected by an ECU failure. This means that the loss of an ECU to which none of the tasks of a non-critical application are mapped can indirectly lead to a failure of that application through degradation.

As an example, Subfigure 5.2a presents the mapping of a non-critical application consisting of the task instances $t_{2,a}$ and $t_{3,a}$ onto the system. In this example, we assume the critical application from the previous example in Subfigure 5.1a is still mapped onto ECUs. The two task instances allocated a slot each. However, both of these slots are also reserved by the critical application's passive task instances $t_{0,b}$ and $t_{1,b}$. Here, if either the ECU of the task instance $t_{0,a}$ or the ECU of the task instance $t_{1,a}$ of the critical application failed, then the corresponding passive task instance $t_{0,b}$ or $t_{1,b}$ would be activated. This would then lead to a degradation of either $t_{2,a}$ or $t_{3,a}$ and, therefore, to an indirect shutdown of the non-critical application. The structure function $\varphi(z)$ is:

$$\varphi(z) = z(t_{2,a}) \wedge z(t_{0,a}) \wedge z(t_{3,a}) \wedge z(t_{1,a}) \tag{5.11}$$

The resulting structure function $\varphi(y)$ is is also represented as a BDD in Subfigure 5.2b:

$$\varphi(y) = y(\alpha(t_{2,a})) \wedge y(\alpha(t_{0,a})) \wedge y(\alpha(t_{3,a})) \wedge y(\alpha(t_{1,a})) = y(e_2) \wedge y(e_1) \wedge y(e_3) \wedge y(e_0) \tag{5.12}$$

Overall, this means that the failure of any ECU in this example would lead to a loss of the non-critical application. In exchange, the critical application can remain operational if only one ECU is affected by a failure. Therefore, through passive redundancy and graceful degradation, the reliability of the critical application is increased at the cost of reducing the reliability of the non-critical application.

The generalized structure function $\varphi(z)$ for non-critical applications is:

$$\varphi(z) = \bigwedge_{t \in \mathcal{V}} (z(t_a) \wedge \bigwedge_{t^r \in T^r_{t_a}} z(t^r_a)). \tag{5.13}$$

The first part of the equation corresponds to the direct failure of the ECU $\alpha(t_a)$ to which the task $t_a$ is mapped. The second part of the equation includes the indirect shutdown through degradation. Here, we define that $t^r \in T^r_{t_a}$ consists of all tasks $t^r$ of critical applications where a passive task instance has reserved a slot allocated by the task $t_a$. The failure of any ECU $\alpha(t^r_a)$ would lead to an activation of the corresponding passive task instance $t^r_b$ and a shutdown of the task $t_a$.

Using the reliability function 5.3 and the structure functions 5.10 and 5.13, the reliability function and MTTF of each critical and non-critical application can be finally obtained. We are using these functions and the MTTF to evaluate the impact of graceful degradation on non-critical applications in Section 5.5.

## 5.5 Evaluation

In the following, we analyze the impact of graceful degradation on the reliability of critical and non-critical applications using our in-house developed simulation framework. We first introduce our experimental setup in Subsection 5.5.1. Afterward, we present the experimental results of our graceful degradation approach and compare it to active redundancy in Subsection 5.5.2. Then, we analyze the effect of our allocation and reservation strategies, which we initially introduced in Subsection 2.4.4.4, on the reliability in Subsection 5.5.3. In Subsection 5.5.4, we present and analyze results obtained with the *Predecessor Heuristic*, which aims to reduce the application's failure exposure. Last, we summarize all findings from these experiments and draw a conclusion about the importance and impact of graceful degradation in Subsection 5.5.5.

### 5.5.1 Experimental Setup

Our simulation framework has been developed to simulate automotive hardware architectures and the execution and communication of the system software according to our system model. On top of the simulation framework, we implemented the agents, resource managers, and strategies as described in Chapter 2.4. For the simulation framework, we chose a process-based Discrete-Event Simulation architecture based on the SimPy framework [73]. The hardware architecture and system software are described in a specification file using the XML schema from the OpenDSE framework [72]. The simulation framework supports any type of hardware architecture consisting of ECUs, switches, and links. To allow a dynamic behavior where tasks and agents are moving between ECUs at run-time, we use a communication middleware based on the SOME/IP standard [36]. The middleware consists of a service discovery that dynamically finds services at run-time. Communication participants are either modeled as clients or services. Furthermore, the middleware supports remote procedure calls and includes a publish/subscribe scheme. The framework also offers the possibility to simulate ECU failures by shutting down ECUs. ECU failures are detected via heartbeats that are periodically sent between all ECUs. Once a watchdog does not receive the heartbeat within a specific timeout interval, it reports the corresponding ECU failure. We use shortest path routing based on Dijkstra's algorithm for routing the messages [38]. Our experiments use applications synthetically generated by the OpenDSE framework using the TGFF algorithm [72, 80].

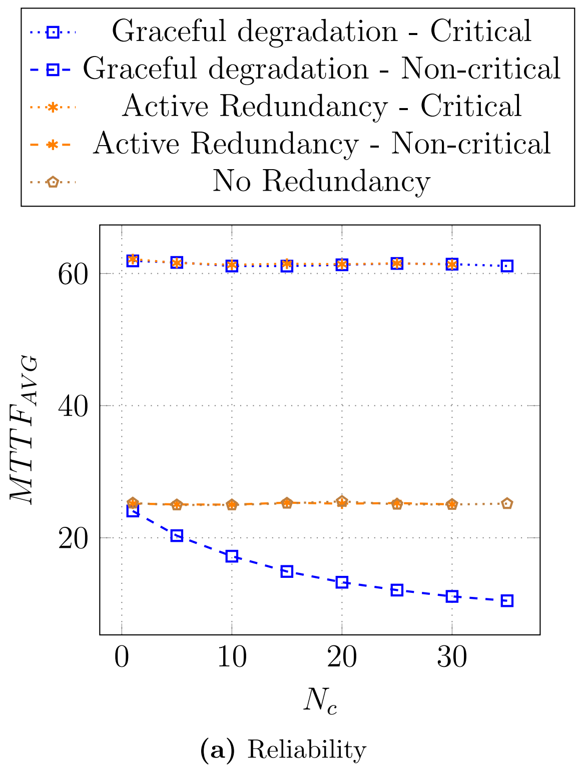
After each application has been mapped successfully, our framework generates the structure functions according to equations 5.10 and 5.13. Afterward, we use the JRELIABILITY framework [99] to evaluate the structure functions and to obtain the MTTF as our evaluation metric. In the following, we calculate and present the average MTTF of the critical and non-critical applications as

$$MTTF_{AVG} = \frac{\sum_{i=1}^{n} MTTF_i}{n}, \tag{5.14}$$

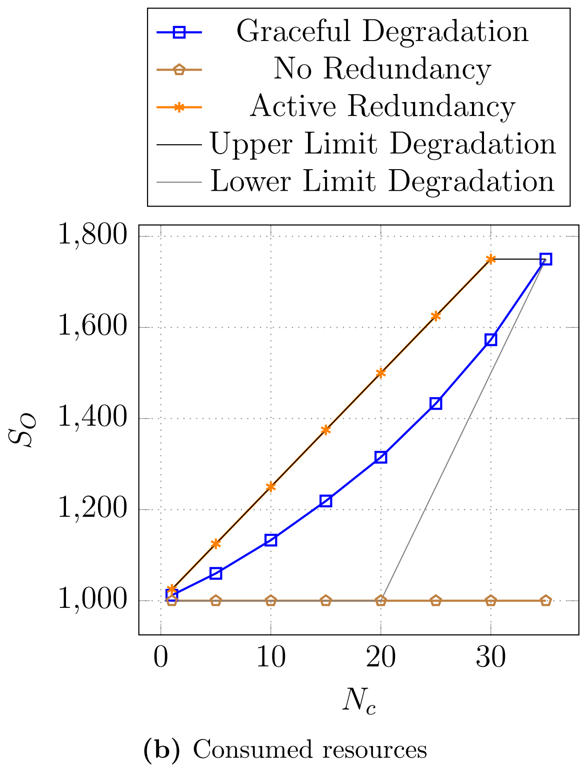
where $n$ is the number of critical or non-critical applications. All experiments use a hardware architecture consisting of ten ECUs. Each ECU has a capacity of $S_e = 175$ slots that can be allocated and reserved by critical and non-critical tasks, resulting in $S_t = 1750$ slots overall supplied by the architecture. In each setup, the overall number of applications $N_a$ mapped onto the architecture was set to 40 applications consisting of five tasks each. In the experiments, the amount of critical $N_c$ and non-critical $N_{nc}$ applications is varied to evaluate scenarios where resources are sufficiently available and limited.

### 5.5.2 Graceful Degradation vs. State of the Art

Subfigure 5.3a presents the average $MTTF_{AVG}$ separated for non-critical and critical applications for a scenario with no redundancy and a scenario with our solution where redundancy and graceful degradation is applied. In this example, we used the *Random* strategy for allocating and reserving slots as the default solution. Other strategies are



**(a)** Reliability



**(b)** Consumed resources

**Figure 5.3:** Experimental results presenting the $MTTF_{AVG}$ and number of consumed slots $S_O$ of our graceful degradation approach (blue plot lines with square marks), an active redundancy approach (orange plot lines with asterisk marks) and no redundancy (brown plot lines with pentagram marks) scenario over an increasing number of critical application $N_c$. Our graceful degradation approach significantly reduces resource consumption compared to an active redundancy approach while guaranteeing the same reliability to critical applications. In this example, it is possible to fit five more critical applications onto the same hardware platform with our graceful degradation approach compared to active redundancy.

evaluated in the following experiments. In the case of no redundancy (brown dotted plot line with pentagram marks), the average $MTTF_{AVG}$ is constant over all experiments for both critical and non-critical applications. With an active redundancy approach (orange plot lines with asterisk marks) it can be observed that the $MTTF_{AVG}$ more than doubles for critical applications compared to no redundancy while the value remains constant for non-critical applications. However, it was impossible to evaluate a scenario with $N_c = 35$ critical applications here as the resource limit was reached with $N_c = 30$ critical applications. Using our approach with passive redundancy and graceful degradation (blue plot lines with square marks) the same $MTTF_{AVG}$ for critical applications as with an active redundancy approach could be reached. As the approach is more resource-saving than active redundancy it is also possible to fit $N_c = 35$ critical applications on the same hardware platform. However, in the case of non-critical applications a steady decline of the $MTTF_{AVG}$ with an increasing amount of critical applications can be observed. With more critical applications on the same hardware architecture resources become more limited leading to an increased chance that slots allocated by non-critical tasks are also reserved for critical tasks. This means there are more possible scenarios where in a failure scenario non-critical applications are shut down to save a critical application resulting in reduced reliability for non-critical applications compared to the cases of no redundancy and active redundancy.

This becomes more obvious when looking at Subfigure 5.3b which presents the resource consumption as the total number of occupied slots $S_O$ of all applications. Next to the three scenarios, graceful degradation, active redundancy, and no redundancy, the plot also shows the analytically derived lower and upper limits for the resource consumption of our graceful degradation solution. The resource consumption of all applications without any redundancy remains constant at $S_O = 1000$ slots. The resource consumption for active redundancy follows the upper limit until the maximum available resource of $S_O = 1750$ slots on the platform is hit at $N_c = 30$ critical applications. The resource consumption of our graceful degradation solution increases steadily with an increasing amount of critical applications reaching the maximum amount of $S_O = 1750$ possible slots, which is the maximum capacity of the hardware architecture, at $N_c = 35$. If the number of critical applications were increased further not all applications could fit onto the platform. It can be observed that our solution is located approximately in the middle between the upper and the lower limit.

To further compare our graceful degradation approach with active redundancy we introduce two metrics. We define

$$MTTF_{reduction,nc} = -\frac{MTTF_{AVG,active,nc} - MTTF_{AVG,deg,nc}}{MTTF_{AVG,active,nc}}, \tag{5.15}$$

as the percental MTTF reduction of non-critical applications of our degradation approach compared to the active redundancy approach with $MTTF_{AVG,active,nc}$ representing the average MTTF of non-critical applications for the active redundancy approach and $MTTF_{AVG,deg,nc}$ representing the average MTTF of non-critical applications for our graceful degradation approach. For the resource consumption we define $S_{OH,deg}$ as the slot overhead introduced by the degradation approach as $S_{OH,deg} = S_{O,deg} - S_{O,no}$, with $S_{O,deg}$ being the total number of consumed slots of the degradation approach and $S_{O,no}$ being the total number of consumed slots when using no redundancy. Furthermore, we define $S_{OH,active} = S_{O,active} - S_{O,no}$ as the slot overhead introduced by the active redundancy approach, with $S_{O,active}$ being the total number of consumed slots of the active redundancy approach. Now we can define

$$R_{savings} = \frac{S_{OH,active} - S_{OH,deg}}{S_{OH,active}}, \tag{5.16}$$

as the percental resource savings of our degradation approach over the active redundancy approach.

Figure 5.4 presents our two metrics over the data points obtained through our experiments. It can be observed that the percental resource savings $R_{savings}$ of our degradation approach declines with an increasing number of critical applications. The potential for percental cost savings decreases as with more critical applications and a more constrained resource situation fewer slots for allocation from non-critical applications become available. This leads to a higher allocation of non-occupied slots and adds more resource overhead. The $MTTF_{reduction,nc}$ sinks over time until it reaches a minimum of $-55.6\%$ at $N_c = 30$. With an increasing number of critical applications more slots that are allocated by non-critical applications are also reserved for critical applications leading to the reduction.

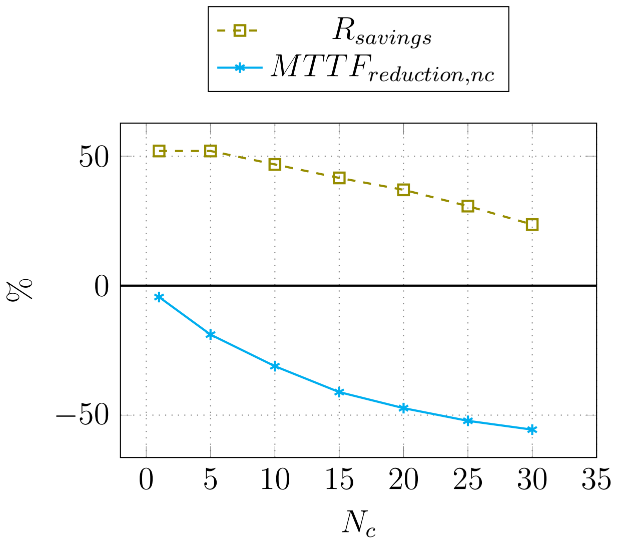


**Figure 5.4:** Experimental results presenting the resource savings $R_{savings}$ (dashed olive plot line with square marks) and the MTTF reduction $MTTF_{reduction,nc}$ of the non-critical applications(solid cyan plot line with asterisk marks) of our graceful degradation approach compared to an active redundancy approach.

Overall the advantage of our graceful degradation approach becomes visible. Our approach consumes significantly fewer resources than an active redundancy approach while still being able to maintain the same MTTF for critical applications. In this example, it is also possible to map five more critical applications onto the same architecture before the resource capacity is reached. This advantage is bought by the decreased reliability of non-critical applications. Consequently, our graceful degradation methodology can significantly reduce resource consumption and costs if a decreased reliability of non-critical applications can be tolerated while guaranteeing the same reliability to critical applications as active redundancy approaches.

## 5.5.3 Allocation and Reservation Strategies

In this subsection we are evaluating and comparing the MTTF and resource costs of the three allocation and reservation strategies *Random*, *Free-Last* and *Free-First*, which were first introduced in Subsectionwhich we initially introduced in Section 2.4.4.4. Subfigure 5.5a presents the MTTF for all three strategies separated into critical and non-critical

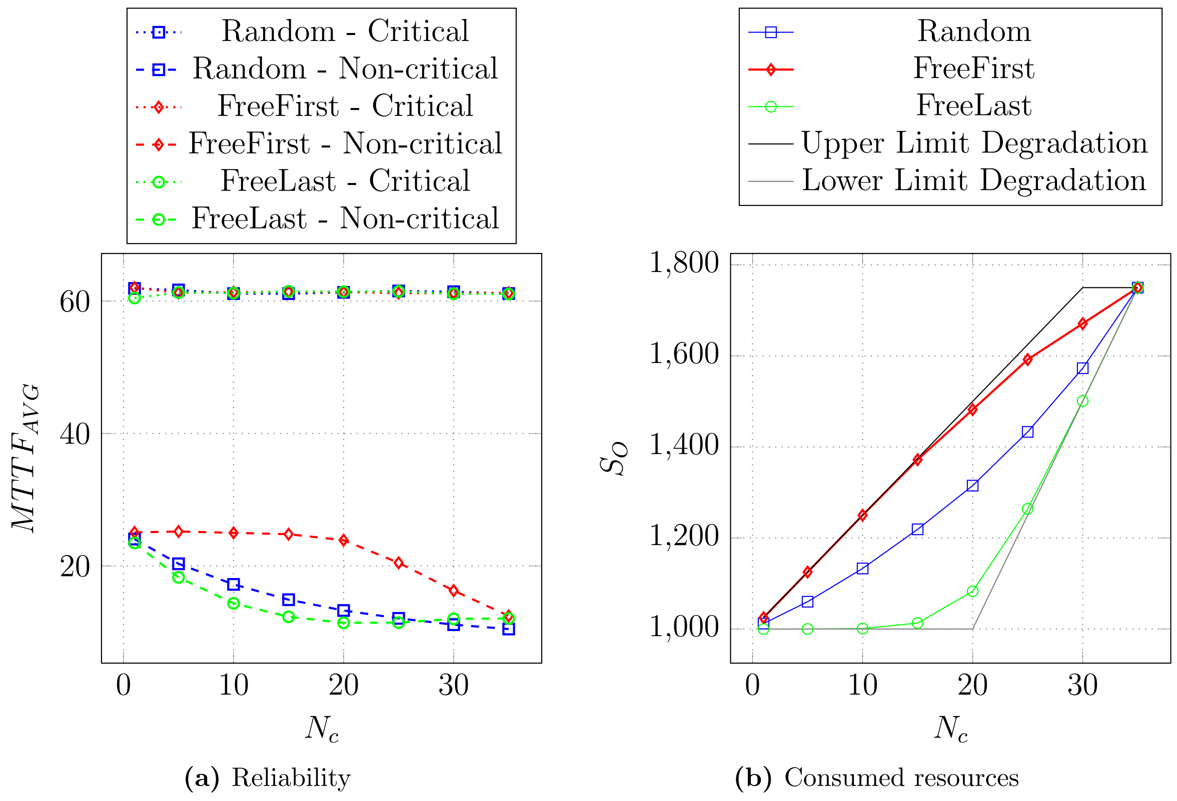


**Figure 5.5:** Experimental results presenting the $MTTF_{AVG}$ and number of consumed slots $S_O$ of our three allocation and reservation strategies *Random* (blue plot lines with square marks), *FreeFirst* (red plot lines with diamond marks) and *FreeLast* (green plot lines with circle marks) over an increasing number of critical application $N_c$. The *FreeFirst* strategy maximizes the use of available resources in order to reduce the degradation effect on non-critical applications but also increases the resource consumption. By contrast, the *FreeLast* strategy uses the resources most efficiently reducing the consumption of hardware resources at the cost of a reduced reliability of non-critical applications.

applications. The *Random* strategy has already been evaluated in the previous experiment in Subsection 5.5.2.

It can be observed that the allocation and reservation strategy does not change the average MTTF of critical applications (dotted plot lines). However, the plot lines for the MTTF of the non-critical application differ from each other. The *Free-First* strategy (red dashed plot line with diamond shapes) keeps a significantly higher MTTF with an increasing number of critical applications and starts dropping later in the plot compared to the other two strategies. By contrast, the MTTF of *Free-Last* strategy (green dashed plot line with circle shapes) drops even earlier than with the *Random* strategy (blue dashed plot line with square shapes). The *Free-First* strategy allocates and reserves first any slots that have not been occupied and, therefore, reduce the number of overlapping slots and by that the degradation impact on non-critical applications. However, as soon as resources become more limited, finding free slots becomes more difficult resulting in a reduced MTTF for non-critical applications. Conversely, the *Free-Last* strategy tries to achieve a maximum overlap of slots by choosing occupied slots first. It only allocates or reserves free slots if not otherwise possible. This also results in a maximum possible degradation effect on non-critical applications and on average a lower MTTF.

Subfigure 5.5b presents the resource costs of the three strategies as the total number of slots occupied by all applications on the platform. The plot line of the *Free-First* strategy (red plot line with diamond shape) remains at the upper limit while the plot line of the *Free-Last* strategy remains at the lower limit (green plot line with circle shape).

These results confirm that *Free-First* strategy maximizes the use of the available resources to reduce the degradation effect, but might increase the resource costs. Therefore, for a given amount of resources, the strategy avoids unnecessary degradation by using up the resources as far as possible. By contrast, the *Free-Last* strategy uses the resources most efficiently reducing the consumption of hardware resources at the cost of reduced reliability of non-critical applications. Overall these experiments show that *Free-First* strategy minimizes the degradation effect while *Free-Last* strategy maximizes it.

### 5.5.4 Exposure Reduction

In previous experiments, the ECU of a task instance was chosen randomly by the agent and the task was mapped to the next ECU with available resources. This led to scenarios where both critical and non-critical applications were highly distributed over the whole hardware platform. This would mean that more ECU failures could lead to the failure of an application. However, it would be preferable regarding reliability to reduce the exposure to different failure sources. Therefore, we introduce the *Predecessor Heuristic* where agents do not choose ECUs randomly but prefer ECUs to which preceding task instances are already mapped to concentrating the task instances on a smaller number of ECUs.

Figure 5.6 presents the results for the three allocation and reservation strategies *Random*, *FreeFirst*, and *FreeLast* as well the results for the case with no redundancy. Comparing the scenario without redundancy (brown plot line with pentagon marks) to Subfigure 5.3a, the MTTF increased from around 25 to almost 100. Here, the tasks of one application are almost always mapped to only on ECU, while previously applications were scattered around 4.1 ECUs on average. The MTTF is even exceeding the MTTF of critical applications from Subfigure 5.3a. Although the critical applications had a double redundancy, they were distributed around 6.5 ECUs on average which increased the exposure to failure significantly.

In Figure 5.6 the plot lines for the critical applications of all three strategies (dotted plot lines) are similar. The MTTF starts around 152 until it hits a low around an MTTF of 115 at $N_c = 35$. Initially, agents of critical applications manage to distribute the task instances mostly on two ECUs (at least two ECUs have to be involved to ensure double redundancy). However, as resources become more limited, the chance to map all task

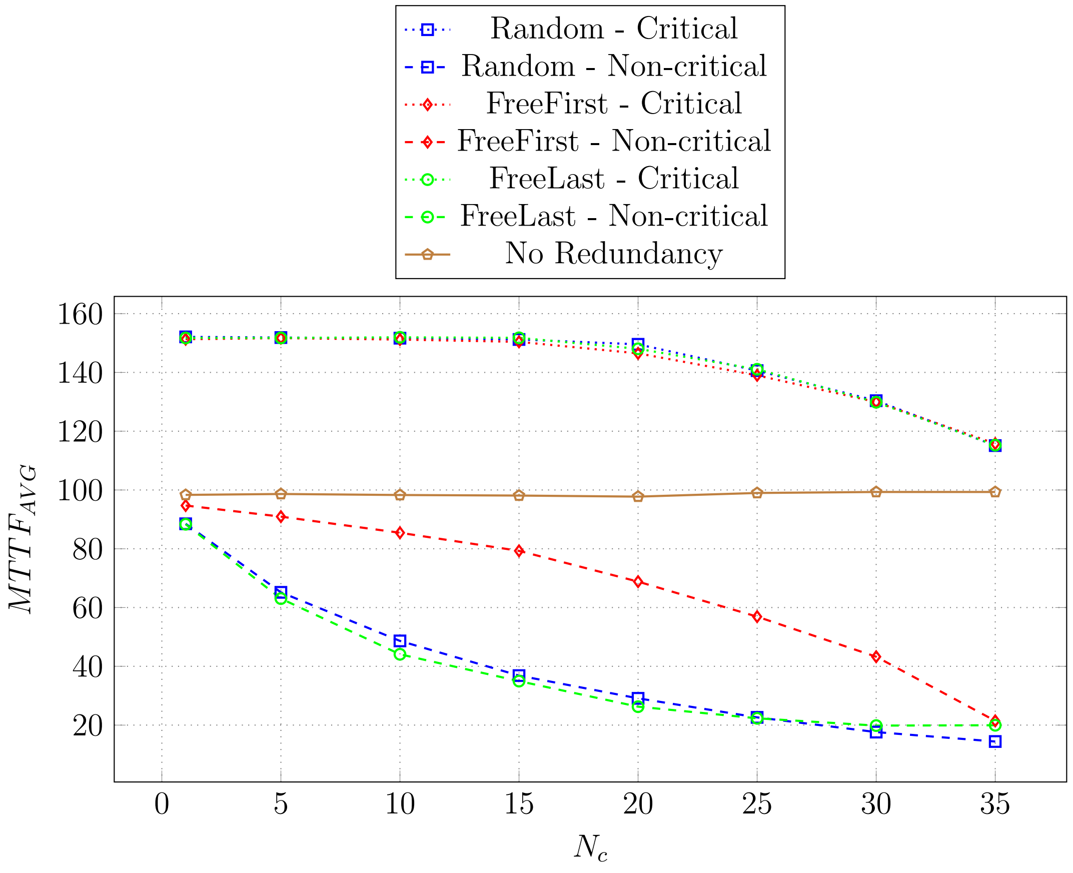


**Figure 5.6:** Experimental results presenting the $MTTF_{AVG}$ of our three allocation and reservation strategies *Random* (blue plot lines with square marks), *FreeFirst* (red plot lines with diamond marks) and *FreeLast* (green plot lines with circle marks) using our *Predeccesor Heuristic* over an increasing number of critical application $N_c$. Compared to previous experiments, the $MTTF_{AVG}$ of both critical and non-critical applications increased significantly as the exposure of applications to different failure sources was reduced. The $MTTF_{AVG}$ decreases with an increasing number of critical applications $N_c$ as resources become more limited and the chance to map all task instances on the same two ECUs decreases, distributing the critical application more on the hardware platform.

instances on the same two ECUs decreases, distributing the critical application more on the hardware platform. This effect is not visible in Subfigure 5.3a as the ECUs were already chosen randomly.

The general course of the plot lines of non-critical applications of all three strategies (dashed plot lines) is similar as in Subfigure 5.5a but there are a few differences. Overall the MTTF starts at slightly below 100 until it hits a low around an MTTF of 20 at $N_c = 35$. Here, the degradation has a much more significant impact on the MTTF than in the previous example. The reason is that the MTTF starts with a high value where applications are mostly mapped to only one ECU and, thus, only the failure of one ECU can cause an application failure. When resources become more limited, more slots for non-critical tasks are reserved from many different critical applications, which themselves are also getting more distributed on the platform. Overall this increases the exposure of non-critical applications to about 5 ECUs on average. This experiment has shown that while redundancy can significantly increase reliability, reducing the exposure of applications to different failure sources is a leverage that should not be underestimated.

### 5.5.5 Summary

Overall our experiments have shown that graceful degradation significantly reduces resource consumption while maintaining the same reliability as an active redundancy approach for critical applications. Furthermore, it is possible to fit more applications onto the same hardware platform as resources are used more efficiently. The decreased reliability of non-critical applications buys this advantage. Additionally, we evaluated the three allocation and reservation strategies *Random*, *FreeFirst*, and *FreeLast*. The *FreeFirst* strategy maximizes the use of available resources and reduces the degradation effect on non-critical applications but also increases resource consumption. By contrast, the *FreeLast* strategy minimizes resource consumption at the cost of reduced reliability of non-critical applications. Last we evaluated our *Predecessor Heuristic* where tasks preferred ECUs to which already other tasks of the same application were mapped. Here, the reliability of both critical and non-critical applications increased significantly as the exposure of applications to different failure sources was reduced. In summary, graceful degradation can be a powerful methodology that uses resources more efficiently than common redundancy approaches and can enormously increase the number of applications that can be mapped onto the same system architecture while providing the same fail-operational capabilities.

## 5.6 Conclusion

In this chapter we presented a reliability analysis of our gracefully degrading automotive systems which we introduced in Chapter 2. With a graceful degradation approach, the reliability of critical applications is increased at the cost of a decrease in the reliability of non-critical applications. To quantify and understand the effect that a graceful degradation approach has on both critical and non-critical applications we gave a short overview of state-of-the-art reliability analysis. We introduced our approach to formally analyze the impact of graceful degradation on the reliability of critical and non-critical applications. We performed multiple experiments on our in-house developed simulation platform to evaluate our graceful degradation approach. Compared to active redundancy, graceful degradation can significantly reduce resource consumption while maintaining the same reliability for critical applications. However, this advantage is bought with a reduced reliability of non-critical applications. In resource-constrained scenarios, the graceful degradation approach can map more applications on the same hardware platform than an active redundancy approach. Our experiments also showed that the *FreeFirst* strategy maximizes the use of available resources and reduces the degradation effect on non-critical applications but also increases resource consumption. By contrast, the *FreeLast* strategy minimizes resource consumption at the cost of reduced reliability of non-critical applications. The experimental results with the *Predecessor Heuristic* showed that the reliability of critical and non-critical applications could be increased significantly as the exposure of applications to different failure sources was reduced. Summarized, our results confirm that graceful degradation can be a powerful methodology that significantly reduces resource consumption compared to active redundancy while providing the same fail-operational capabilities to critical applications.

# 6 Conclusions and Future Work

To enable autonomous driving a fail-operational behavior of safety-critical applications has to be ensured. This is a significant challenge for automotive manufacturers as a fail-operational behavior requires redundancy, which would increase hardware costs enormously. Furthermore, they must enable frequent over-the-air software updates and offer configuration possibilities for customers, leading to unique and customized software solutions. State-of-the-art static redundancy approaches can not fulfill the requirements for a cost-sensitive and adaptable solution. Therefore, new techniques are required to achieve a dynamic, safe, and cost-efficient behavior.

To cope with the complexity of software development, the industry is already moving towards more integrated E/E architectures with decoupled hardware and software. Software platforms open new possibilities for more cost-sensitive and dynamic fail-operational approaches. Dynamic graceful degradation is a promising approach where resources are re-distributed after a failure at run-time from non-critical to critical applications to ensure the fail-operational behavior of critical applications. The main challenges of achieving such a dynamic fail-operational approach are to ensure predictable behavior and to find bounds within which a safe and dynamic operation is possible.

## 6.1 Conclusions

In this thesis, we presented an approach to dynamically achieve a gracefully degrading system behavior using an agent-based approach in automotive systems. We introduced methodologies to achieve a more predictable system design and analyze non-functional properties, focusing on timing behavior, state recovery, and reliability analysis.

We contributed a graceful degradation approach based on a resource allocation and reservation system that is, by design, composable. This design of our graceful degradation approach allows a predictable outcome of the degradation process for any ECU failure in the system. Furthermore, it enables an isolated timing analysis of applications and allows a quick failover process.

We presented an agent-based backtracking approach to set up our gracefully degrading system and find suitable resource allocations and reservations at run-time. This approach and the decentralized control of resources enable a dynamic and adaptable behavior and fail-operational management by eliminating any single point of failure. Our first experiments showed that using this approach to reconfigure the system, the number of sustained ECU failures increases significantly until a critical application is affected by a loss directly.

Utilizing the composable design of our graceful degradation approach, we then presented a timing analysis that allows us to verify quickly at run-time if a binding of a critical application (including passive task instances) can meet its timing constraint before and after any possible failover. Furthermore, we embedded this timing analysis in our agent-based approach to find feasible bindings that respect the timing constraint. This further increases the predictability of our approach as this ensures that a valid solution is available for any failure scenario to which a quick failover can be performed. Our experimental results confirmed that graceful degradation strongly increases the success rate of finding a valid mapping for critical applications compared to an active redundancy approach in resource-constrained scenarios. Furthermore, resource utilization could be significantly reduced.

Afterward, we presented a formal analysis to derive the worst-case failover time, ensuring that a failover can be performed within the FTTI. Our analytical exact bound allows us to verify at run-time if a binding respects a failover timing constraint. We conducted experiments with a distributed fail- operational neural network, where we provoked worst-case failure scenarios to measure the failover time. Our randomly generated worst-case results are as close as 6.0% below our analytically derived exact bound. Overall, this approach increases timing predictability in failover scenarios.

To extend our work to stateful applications, we presented a checkpointing and rollback recovery approach. Instead of taking and sending a checkpoint with the application period, we presented an approach to analytically derive the maximal checkpointing period to reduce network and processing overhead. We used a SLAM application to analyze the dependency between state data age and the algorithm's accuracy. Further failover experiments showed that the worst-case results of our randomized experiments were consistent with our analytically derived exact bound.

Last, we evaluated our graceful degradation approach based on resource consumption and the impact on reliability. For this purpose, we presented a formal reliability analysis to derive the reliability of critical and non-critical applications. We compared three allocation and reservation strategies and an active redundancy approach for our evaluation. Our experimental results showed that graceful degradation significantly reduces resource consumption while still maintaining the same reliability as an active redundancy approach for critical applications. Furthermore, it is possible to fit more applications onto the same hardware platform in resource-constrained scenarios using graceful degradation compared to active redundancy. However, the advantages of graceful degradation are purchased with a reliability decrease of non-critical applications. Our *FreeFirst* allocation and reservation strategy minimized the negative impact on the reliability of non-critical applications but also resulted in minimal to no resource savings compared to active redundancy. Our *FreeLast* strategy maximized the graceful degradation effect with the highest reliability decrease for non-critical applications but also resulted in the highest resource savings. Furthermore, in experiments for our predictable timing analysis, the *FreeLast* strategy resulted in the highest success chance to find valid mappings, especially in more resource-constrained scenarios. On the other hand, the *FreeFirst* strategy resulted in the lowest success rate. The *FreeLast* strategy maximizes the graceful degradation effect while the *FreeFirst* strategy minimizes it with all accompanying advantages and disadvantages. Our experimental results confirmed that graceful degradation can be a powerful methodology that significantly reduces resource consumption compared to active redundancy while providing the same fail-operational capabilities to critical applications if a reliability decrease of non-critical applications is acceptable.

Overall, we contributed an approach that combines graceful degradation with a decentralized system control. Compared to conventional redundancy approaches, this approach is more resource-efficient. It allows a dynamic behavior while eliminating any single point of failure for the system control and offering the same fail-operational capabilities to critical applications.

## 6.2 Future Work

Our proposed approaches can be further improved and extended. In the following, we propose further research directions and future work.

**Benchmark** We need a new benchmark suite for autonomous automotive systems to ease the evaluation and comparison of synthesis approaches. This benchmark suite should contain applications from various automotive E/E domains such as infotainment, chassis, powertrain, comfort, or autonomous driving. The data should have a graphical representation of the applications and run times on a few typical hardware elements. Critical applications could be classified regarding their safety requirements, such as a required level of redundancy or reliability and requirements on the FTTI. Further information could include the size and type of state data. Standardized run times and requirements approved by the research community and automotive experts could help create a common communication basis and ease the comparison of synthesis approaches.

**Hybrid Mapping Approaches** While our proposed agent-based approach can find mappings at run-time, there are open questions about the scalability of the approach. Future work could address this issue by evaluating hybrid mapping approaches for our gracefully degrading system. Here, operation points could be pre-computed using meta-heuristic design space exploration approaches in the car or the cloud. Afterward, the resulting constraint graphs can be used to find a feasible mapping for an application at run-time. As time is less limited at design time than after a failover, application mappings can also be optimized, e.g., regarding their energy consumption. An interesting and open question is which time frame would be considered reasonable to perform an in-car reconfiguration after a failover.

**Real-Time Migration** In our current approach, cars must stop after a failover to safely perform a reconfiguration and re-establish redundancy for critical applications. Another interesting research question would contain the real-time migration of safety-critical tasks from one ECU to another. A reconfiguration could be performed without stopping the car if this can be achieved safely and within an application's operation conditions. In this scenario, a passenger would ideally not even notice the incident.

# Bibliography

[1] S. Saidi, S. Steinhorst, A. Hamann, D. Ziegenbein, and M. Wolf. Future Automotive Systems Design: Research Challenges and Opportunities: Special Session. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 1–7, 2018. doi:10.1109/CODESISSS.2018.8525873.

[2] Y. Yeh. Safety Critical Avionics for the 777 Primary Flight Controls System. In *20th DASC. 20th Digital Avionics Systems Conference (Cat. No. 01CH37219)*, volume 1, pages 1C2–1, 2001. doi:10.1109/DASC.2001.963311.

[3] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner. Software engineering for automotive systems: A roadmap. In *Future of Software Engineering (FOSE'07)*, pages 55–71, 2007.

[4] S. Fürst. Challenges in the design of automotive software. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 256–258, 2010. doi:10.1109/DATE.2010.5457201.

[5] M. Lunt. E/E-Architecture in a Connected World. URL: https://www.asam.net/index.php?eID=dumpFile&t=f&f=798&token=148b5052945a466cacfe8f31c44eb22509d5aad1.

[6] WikiChip. Tesla FSD Computer, 2019. URL: https://en.wikichip.org/wiki/tesla_(car_company)/fsd_chip.

[7] Bosch. Vehicle-Centralized, Zone-Oriented E/E Architecture with Vehicle Computers, 2022. URL: https://www.bosch-mobility-solutions.com/en/mobility-topics/ee-architecture/?gclid=Cj0KCQjwgO2XBhCaARIsANrW2X1FfP3bQFfM4NWaTRZBagYqA9gVQ91v9oIriWyb38Xm_ntCPErQgoQaAm09EALw_wcB.

[8] C. Buckl, A. Camek, G. Kainz, C. Simon, L. Mercep, H. Stähle, and A. Knoll. The software car: Building ICT architectures for future electric vehicles. In *2012 IEEE International Electric Vehicle Conference*, pages 1–8, 2012. doi:10.1109/IEVC.2012.6183198.

[9] M. Buechel, J. Frtunikj, K. Becker, S. Sommer, C. Buckl, M. Armbruster, A. Marek, A. Zirkler, C. Klein, and A. Knoll. An Automated Electric Vehicle Prototype Showing New Trends in Automotive Architectures. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pages 1274–1279, 2015. doi:10.1109/ITSC.2015.209.

[10] P. Weber, P. Weiss, D. Reinhardt, and S. Steinhorst. Energy-Optimized Elastic Application Distribution for Automotive Systems in Hybrid Cloud Architectures. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 455–462, 2020. doi:10.1109/DSD51259.2020.00078.

[11] R. Schneider, A. Kohn, K. Schmidt, S. Schoenberg, U. Dannebaum, J. Harnisch, and Q. Zhou. Efficient Virtualization for Functional Integration on Modern Microcontrollers in Safety-Relevant Domains. Technical report, SAE Technical Paper, 2014.

[12] AUTOSAR. AUTOSAR Adaptive Platform. URL: https://www.autosar.org/standards/adaptive-platform/.

[13] P. S. Börge Schmelz, Martin Böhner. Over-the-air updates – what advantages does the AUTOSAR Adaptive Platform offer? URL: https://www.elektrobit.com/trends/ota-updates-with-adaptive-autosar/.

[14] M. Lukasiewycz, S. Steinhorst, S. Andalam, F. Sagstetter, P. Waszecki, W. Chang, M. Kauer, P. Mundhenk, S. Shanker, S. A. Fahmy, et al. System architecture and software design for electric vehicles. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1–6, 2013. doi:10.1145/2463209.2488852.

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

[16] A. Burns and R. I. Davis. Mixed Criticality Systems - A Review. 2022. URL: https://eprints.whiterose.ac.uk/183619/.

[17] B. Akesson, A. Molnos, A. Hansson, J. A. Angelo, and K. Goossens. Composability and Predictability for Independent Application Development, Verification, and Execution. In *Multiprocessor System-on-Chip*, pages 25–56. Springer, 2011. doi:10.1007/978-1-4419-6460-1_2.

[18] R. Isermann, R. Schwarz, and S. Stolzl. Fault-Tolerant Drive-by-Wire Systems. *IEEE Control Systems Magazine*, 22(5):64–81, 2002. doi:10.1109/MCS.2002.1035218.

[19] M. Blanke, M. Staroswiecki, and N. E. Wu. Concepts and methods in fault-tolerant control. In *Proceedings of the 2001 American control conference.(Cat. No. 01CH37148)*, volume 4, pages 2606–2620, 2001. doi:10.1109/ACC.2001.946264.

[20] C. Temple and A. Vilela. Fehlertolerante Systeme im Fahrzeug-Von Fail Safe zu Fail Operational. *Elektronik automotive*, pages 1614–0125, 2014.

[21] A. Kohn, R. Schneider, A. Vilela, A. Roger, and U. Dannebaum. Architectural Concepts for Fail-Operational Automotive Systems. Technical report, SAE Technical Paper, 2016. doi:10.4271/2016-01-0131.

[22] Infineon Technologies AG. AURIX™ 32-bit Microcontrollers for Automotive and Industrial Applications. URL: https://www.infineon.com/dgdl/Infineon-TriCore_Family_BR-ProductBrochure-v01_00-EN.pdf?fileId=5546d4625d5945ed015dc81f47b436c7.

[23] R. E. Lyons and W. Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962. doi:10.1147/rd.62.0200.

[24] O. González, H. Shrikumar, J. A. Stankovic, and K. Ramamritham. Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *Proceedings Real-Time Systems Symposium*, pages 79–89, 1997. doi:10.1109/REAL.1997.641271.

[25] C. P. Shelton, P. Koopman, and W. Nace. A Framework for Scalable Analysis and Design of System-wide Graceful Degradation in Distributed Embedded Systems. In *Proceedings of the 8th Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, pages 156–163, 2003. doi:10.1109/WORDS.2003.1218078.

[26] J. C. Knight and K. J. Sullivan. On The Definition Of Survivability. Technical report, University of Virginia, Department of Computer Science, 2000. URL: https://www.cse.msu.edu/~cse870/Materials/FaultTolerant/john.dsn.pdf.

[27] Meyer. On Evaluating the Performability of Degradable Computing Systems. *IEEE Transactions on Computers*, C-29(8):720–731, 1980. doi:10.1109/TC.1980.1675654.

[28] P. Weiss, A. Weichslgartner, F. Reimann, and S. Steinhorst. Fail-Operational Automotive Software Design Using Agent-Based Graceful Degradation. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1169–1174, 2020. doi:10.23919/DATE48585.2020.9116322.

[29] P. Weiss and S. Steinhorst. Predictable Timing Behavior of Gracefully Degrading Automotive Systems. *Design Automation for Embedded Systems*, pages 1–36, 2023. doi:10.1007/s10617-023-09271-x.

[30] P. Weiss, S. Elsabbahy, A. Weichslgartner, and S. Steinhorst. Worst-Case Failover Timing Analysis of Distributed Fail-Operational Automotive Applications. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1294–1299, 2021. doi:10.23919/DATE51398.2021.9473950.

[31] P. Weiss, E. Daporta, A. Weichslgartner, and S. Steinhorst. Checkpointing Period Optimization of Distributed Fail-Operational Automotive Applications. In *2021 24th Euromicro Conference on Digital System Design (DSD)*, pages 389–395, 2021. doi:10.1109/DSD53832.2021.00066.

[32] P. Weiss, A. Younessi, and S. Steinhorst. Reliability Analysis of Gracefully Degrading Automotive Systems. *Preprint on arXiv*, 2023. doi:10.48550/arXiv.2305.07401.

[33] P. Weiss, S. Nagel, A. Weichslgartner, and S. Steinhorst. Adaptable Demonstrator Platform for the Simulation of Distributed Agent-Based Automotive Systems. In *2nd International Workshop on Autonomous Systems Design (ASD 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020. doi:10.4230/OASIcs.ASD.2020.3.

[34] A. Kohn, M. Käßmeyer, R. Schneider, A. Roger, C. Stellwag, and A. Herkersdorf. Fail-Operational in Safety-Related Automotive Multi-core Systems. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–4, 2015. doi:10.1109/SIES.2015.7185051.

[35] K. Becker and S. Voss. Analyzing Graceful Degradation for Mixed Critical Fault-Tolerant Real-Time Systems. In *18th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 110–118, 2015. doi:10.1109/ISORC.2015.10.

[36] *Scalable service-Oriented MiddlewarE over IP (SOME/IP)*, 2021. URL: http://some-ip.com/.

[37] International Organization for Standardization. *ISO 26262, Road vehicles - Functional Safety - Part 1-9*, 1st edition, 2011.

[38] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959. doi:https://doi.org/10.1007/BF01386390.

[39] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004. doi:10.1109/TDSC.2004.2.

[40] B. Pourmohseni, F. Smirnov, S. Wildermann, and J. Teich. Real-Time Task Migration for Dynamic Resource Management in Many-Core Systems. In *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, volume 77 of *OpenAccess Series in Informatics (OASIcs)*, pages 5:1–5:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020.

[41] F. Smirnov, F. Reimann, J. Teich, Z. Han, and M. Glaß. Automatic Optimization of Redundant Message Routings in Automotive Networks. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, pages 90–99, 2018. doi:10.1145/3207719.3207725.

[42] T. Frese, T. Leonhardt, D. Hatebur, I. Côté, H.-J. Aryus, and M. Heisel. Fault Tolerance Time Interval: How to define and handle. *Neue Dimensionen der Mobilität: Technische und betriebswirtschaftliche Aspekte*, pages 559–567, 2020. doi:10.1007/978-3-658-29746-6_45.

[43] D. Penha, G. Weiss, and A. Stante. Pattern-Based Approach for Designing Fail-Operational Safety-Critical Embedded Systems. In *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, pages 52–59, 2015. doi:10.1109/EUC.2015.14.

[44] H. Seebach, F. Nafz, J. Holtmann, J. Meyer, M. Tichy, W. Reif, and W. Schäfer. Designing Self-healing in Automotive Systems. In *International Conference on Autonomic and Trusted Computing*, pages 47–61, 2010. doi:10.1007/978-3-642-16576-4_4.

[45] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini. Fault-Tolerant Platforms for Automotive Safety-Critical Applications. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 170–177, 2003. URL: https://doi-org.eaccess.ub.tum.de/10.1145/951710.951734, doi:10.1145/951710.951734.

[46] S. Sommer, A. Camek, K. Becker, C. Buckl, A. Zirkler, L. Fiege, M. Armbruster, G. Spiegelberg, and A. Knoll. Race: A centralized platform computer based architecture for automotive applications. In *2013 IEEE International Electric Vehicle Conference (IEVC)*, pages 1–6, 2013. doi:10.1109/IEVC.2013.6681152.

[47] K. Becker, J. Frtunikj, M. Felser, L. Fiege, C. Buckl, S. Rothbauer, L. Zhang, and C. Klein. RACE RTE: a runtime environment for robust fault-tolerant vehicle functions. In *CARS Workshop, 11th European Dependable Computing Conference-Dependability in Practice*, 2015.

[48] R. Mariani, T. Kuschel, and H. Shigehara. A Flexible Microcontroller Architecture for Fail-Safe and Fail-Operational Systems. In *2nd HiPEAC Workshop on Design for Reliability (DFR'10)*, 2010. URL: http://www.ece.ucy.ac.cy/labs/easoc/dfr10/Papers/mariani_kuschel_shigehara_dfr2010_final.pdf.

[49] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini. Fault-Tolerant Platforms for Automotive Safety-Critical Applications. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 170–177, 01 2003. doi:10.1145/951710.951734.

[50] T. Fruehling. Delphi Secured Microcontroller Architecture. *SAE Transactions*, pages 317–328, 2000. doi:10.4271/2000-01-1052.

[51] J. Braun and J. Mottok. Fail-Safe and Fail-Operational Systems Safeguarded with Coded Processing. In *Eurocon 2013*, pages 1878–1885, 2013. doi:10.1109/EUROCON.2013.6625234.

[52] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha. The System-Level Simplex Architecture for Improved Real-Time Embedded System

Safety. In *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 99–107, 2009. doi:10.1109/RTAS.2009.20.

[53] F. Bapp, T. Dörr, T. Sandmann, F. Schade, and J. Becker. Towards Fail-Operational Systems on Controller Level Using Heterogeneous Multicore SoC Architectures and Hardware Support. Technical report, SAE Technical Paper, 2018. doi:10.4271/2018-01-1072.

[54] L. Sha. Using Simplicity to Control Complexity. *IEEE Software*, 18(4):20–28, 2001. doi:10.1109/MS.2001.936213.

[55] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha. The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety. In *15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 99–107, 2009. doi:10.1109/RTAS.2009.20.

[56] F. Oszwald, P. Obergfell, M. Traub, and J. Becker. Reliable Fail-Operational Automotive E/E-Architectures by Dynamic Redundancy and Reconfiguration. In *2019 32nd IEEE International System-on-Chip Conference (SOCC)*, pages 203–208, 2019. doi:10.1109/SOCC46988.2019.1570547977.

[57] B. Randell, P. Lee, and P. C. Treleaven. Reliability Issues in Computing System Design. *ACM Comput. Surv.*, 10(2):123–165, 1978. doi:10.1145/356725.356729.

[58] J. Kim, G. Bhatia, R. Rajkumar, and M. Jochim. Safer: System-Level Architecture for Failure Evasion in Real-Time Applications. In *2012 IEEE 33rd Real-Time Systems Symposium*, pages 227–236, 2012. doi:10.1109/RTSS.2012.74.

[59] J. Kim, R. Rajkumar, and M. Jochim. Towards dependable autonomous driving vehicles: a system-level approach. *ACM SIGBED Review*, 10(1):29–32, 2013. doi:10.1145/2492385.2492390.

[60] A. Ruiz, G. Juez, P. Schleiss, and G. Weiss. A safe generic adaptation mechanism for smart cars. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 161–171, 2015. doi:10.1109/ISSRE.2015.7381810.

[61] T. Saridakis. Design patterns for graceful degradation. *Transactions on Pattern Languages of Programming I*, pages 67–93, 2009. doi:10.1007/978-3-642-10832-7_3.

[62] O. González, H. Shrikumar, J. A. Stankovic, and K. Ramamritham. Adaptive Fault Tolerance and Graceful Degradation under Dynamic Hard Real-Time Scheduling. In *Proceedings Real-Time Systems Symposium*, pages 79–89, 1997. doi:10.1109/REAL.1997.641271.

[63] K. G. Shin and C. L. Meissner. Adaptation and Graceful Degradation of Control System Performance by Task Reallocation and Period Adjustment. In *Proceedings*

*of 11th Euromicro Conference on Real-Time Systems (Euromicro RTS'99)*, pages 29–36, 1999. doi:10.1109/EMRTS.1999.777447.

[64] W. Nace and P. Koopman. A Graceful Degradation Framework for Distributed Embedded Systems. 2001. doi:10.1184/R1/6620663.v1.

[65] M. P. Herlihy and J. M. Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, 1991. doi:10.1109/71.80192.

[66] M. Glaß, M. Lukasiewycz, C. Haubelt, and J. Teich. Incorporating Graceful Degradation into Embedded System Design. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 320–323, 2009. doi:10.1109/DATE.2009.5090681.

[67] A. Weichslgartner, S. Wildermann, and J. Teich. Dynamic Decentralized Mapping of Tree-Structured Applications on NoC Architectures. In *Proceedings of the Fifth ACM/IEEE International Symposium*, pages 201–208, 2011. doi:10.1145/1999946.1999979.

[68] M. Faruque, R. Krist, and J. Henkel. ADAM: Run-Time Agent-Based Distributed Application Mapping for On-Chip Communication. In *Proceedings of the 45th Annual Design Automation Conference*, pages 760–765, 2008. doi:10.1145/1391469.1391664.

[69] E. L. de Souza Carvalho, N. L. V. Calazans, and F. G. Moraes. Dynamic Task Mapping for MPSoCs. *IEEE Des. Test*, 27(5):26–35, 2010. doi:10.1109/MDT.2010.106.

[70] B. Pourmohseni, S. Wildermann, M. Glaß, and J. Teich. Predictable Run-Time Mapping Reconfiguration for Real-Time Applications on Many-Core Systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 148–157, 2017. doi:10.1145/3139258.3139278.

[71] E. Rambo et al. The Information Processing Factory: A Paradigm for Life Cycle Management of Dependable Systems. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. IEEE, 2019.

[72] F. Reimann, M. Lukasiewycz, M. Glaß, and F. Smirnov. *OpenDSE - Open Design Space Exploration Framework*, 2021. URL: http://opendse.sourceforge.net/.

[73] T. SimPy. *SimPy Discrete Event Simulation Library for Python, Version 4.0.1*, 2021. URL: https://simpy.readthedocs.io.

[74] A. Weichslgartner, S. Wildermann, D. Gangadharan, M. Glaß, and J. Teich. A Design-Time/Run-Time Application Mapping Methodology for Predictable Execution Time in MPSoCs. *ACM Trans. Embed. Comput. Syst.*, 17(5), 2018. doi:10.1145/3274665.

[75] Z. Guo, K. Yang, S. Vaidhun, S. Arefin, S. K. Das, and H. Xiong. Uniprocessor mixed-criticality scheduling with graceful degradation by completion rate. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 373–383, 2018. doi:10.1109/RTSS.2018.00052.

[76] B. Pourmohseni, M. Glaß, J. Henkel, H. Khdr, M. Rapp, V. Richthammer, T. Schwarzer, F. Smirnov, J. Spieck, J. Teich, et al. Hybrid Application Mapping for Composable Many-Core Systems: Overview and Future Perspective. *Journal of Low Power Electronics and Applications*, 10(4), 2020. doi:10.3390/jlpea10040038.

[77] F. Oszwald, J. Becker, P. Obergfell, and M. Traub. Dynamic Reconfiguration for Real-Time Automotive Embedded Systems in Fail-Operational Context. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 206–209, 2018. doi:10.1109/IPDPSW.2018.00039.

[78] F. Krichen, B. Hamid, B. Zalila, and B. Coulette. Designing Dynamic Reconfiguration for Distributed Real Time Embedded Systems. In *2010 10th Annual International Conference on New Technologies of Distributed Systems (NOTERE)*, pages 249–254, 2010. doi:10.1109/NOTERE.2010.5536671.

[79] J. Heisswolf, R. König, M. Kupper, and J. Becker. Providing Multiple Hard Latency and Throughput Guarantees for Packet Switching Networks on Chip. *Comput. Electr. Eng.*, 39(8):2603–2622, 2013. doi:10.1016/j.compeleceng.2013.06.005.

[80] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: task graphs for free. In *Proceedings of the Sixth International Workshop on Hardware/Software Codesign. (CODES/-CASHE'98)*, pages 97–101, 1998.

[81] T. Schwarzer, S. Roloff, V. Richthammer, R. Khaldi, S. Wildermann, M. Glaß, and J. Teich. On the complexity of mapping feasibility in many-core architectures. In *2018 IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 176–183, 2018. doi:10.1109/MCSoC2018.2018.00038.

[82] J. Redmon and A. Farhadi. YOLOv3: An Incremental Improvement. *arXiv*, 2018.

[83] S. Vom Dorff, B. Böddeker, M. Kneissl, and M. Fränzle. A Fail-Safe Architecture for Automated Driving. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 828–833, 2020. doi:10.23919/DATE48585.2020.9116283.

[84] S. M. A. Akber, H. Chen, Y. Wang, and H. Jin. Minimizing Overheads of Checkpoints in Distributed Stream Processing Systems. In *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*, pages 1–4, 2018. doi:10.1109/CloudNet.2018.8549548.

[85] T. Aung, H. Y. Min, and A. H. Maw. Coordinate Checkpoint Mechanism on Real-Time Messaging System in Kafka Pipeline Architecture. In *2019 International Conference on Advanced Information Technologies (ICAIT)*, pages 37–42, 2019. doi:10.1109/AITC.2019.8921392.

[86] P. Pop, V. Izosimov, P. Eles, and Z. Peng. Design Optimization of Time- and Cost-Constrained Fault-Tolerant Embedded Systems With Checkpointing and Replication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(3):389–402, 2009. doi:10.1109/TVLSI.2008.2003166.

[87] A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2011.

[88] AtsushiSakai. EKF-SLAM. https://github.com/AtsushiSakai/PythonRobotics/blob/master/Localization/extended_kalman_filter/extended_kalman_filter_localization.ipynb.

[89] A. Birolini. *Reliability Engineering: Theory and Practice*. Engineering online library. Springer Berlin Heidelberg, 4 edition, 2010. doi:10.1007/978-3-642-14952-8.

[90] M. Glaß, M. Lukasiewycz, T. Streichert, C. Haubelt, and J. Teich. Reliability-Aware System Synthesis. In *2007 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2007. doi:10.1109/DATE.2007.364626.

[91] R. Karri and A. Orailoglu. Transformation-Based High-Level Synthesis of Fault-Tolerant ASICs. In *DAC*, pages 662–665, 1992. doi:10.1109/DAC.1992.227803.

[92] S. Tosun, N. Mansouri, E. Arvas, M. Kandemir, and Y. Xie. Reliability-Centric High-Level Synthesis. In *Design, Automation and Test in Europe*, pages 1258–1263, 2005. doi:10.1109/DATE.2005.258.

[93] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Reliability-Aware Co-Synthesis for Embedded Systems. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 49(1):87–99, 2007. doi:10.1007/s11265-007-0057-6.

[94] A. Jhumka, S. Klaus, and S. A. Huss. A Dependability-Driven System-Level Design Approach for Embedded Systems. In *Design, Automation and Test in Europe*, pages 372–377, 2005. doi:10.1109/DATE.2005.10.

[95] M. Glaß, M. Lukasiewycz, F. Reimann, C. Haubelt, and J. Teich. Symbolic Reliability Analysis of Self-Healing Networked Embedded Systems. In *International Conference on Computer Safety, Reliability, and Security*, pages 139–152, 2008. doi:10.1007/978-3-540-87698-4_14.

[96] D. Koch, T. Streichert, S. Dittrich, C. Strengert, C. D. Haubelt, and J. Teich. An Operating System Infrastructure for Fault-Tolerant Reconfigurable Networks. In *International Conference on Architecture of Computing Systems*, pages 202–216, 2006. doi:10.1007/11682127_15.

[97] N. Medvidovic and R. N. Taylor. Software architecture: foundations, theory, and practice. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 471–472, 2010. doi:10.1145/1810295.1810435.

[98] P. Koopman. *Better embedded system software*. Drumnadrochit Education, 2010.

[99] M. Glaß. *The JRELIABILITY Tutorials*, 2008. URL: http://jreliability.org/tutorial/tutorial.pdf.

[100] A. Rauzy. New algorithms for fault trees analysis. *Reliability Engineering & System Safety*, 40(3):203–211, 1993. doi:10.1016/0951-8320(93)90060-C.

[101] M. Glaß, M. Lukasiewycz, C. Haubelt, and J. Teich. Towards scalable system-level reliability analysis. In *Proceedings of the 47th Design Automation Conference*, pages 234–239, 2010. doi:10.1145/1837274.1837334.