# An Approach to Cost-Efficient Fault Tolerance in Inherently Redundant Fail-Operational Systems

Tobias Dörr*, Timo Sandmann*, Patrick Friederich†, Arnd Leitner‡, and Jürgen Becker*
*Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
Email: {tobias.doerr, sandmann, becker}@kit.edu
†Vector Informatik GmbH, Stuttgart, Germany
‡Schaeffler Automotive Buehl GmbH & Co. KG, Bühl, Germany

***Abstract*—Embedded systems in safety-critical environments are often subject to strict reliability requirements. This holds particularly true for modern fail-operational systems. In order to deliver a guaranteed minimum functionality at all times, these systems are often based on expensive fault tolerance mechanisms. In this work, we consider fail-operational systems with inherent redundancy. This property describes the presence of multiple hardware components, each of which is underutilized to a certain degree and thus able to serve as a fallback for one of the other components. We propose an off-chip fault tolerance mechanism for a pair of inherently redundant execution units that requires no further replication of these expensive resources. The key component of this concept is a lightweight proxy unit that handles faults of one execution unit by dynamically migrating the safety-critical portion of its functionality to its redundant counterpart. We present a prototypical implementation of this concept and evaluate the fault handling time of the resulting system experimentally. The results show that for an exemplary, processor-based control system with 256 bits of internal state, a cycle time of four milliseconds, and 64 bits of payload data that are read from or written to attached devices per cycle, the presented implementation is able to detect the failure of a unit, activate a fallback functionality on the complementary unit, and restore the internal state variables within five milliseconds.**

***Index Terms*—Fail-operational systems, embedded processors, simplex architecture, fault tolerance, dynamic redundancy, emergency operation, FPGA prototype, CAN bus, safety.**

## I. Introduction

Since the advent of the first commercially available microprocessor, the computational power that such integrated circuits are able to deliver has increased by several orders of magnitude. This has led to a world in which embedded systems are ubiquitous and fulfill an ever-increasing number of functions in the most diverse fields of human life. In safety-critical environments, however, an embedded system must also meet certain dependability requirements. A common example of such an environment is the automotive domain, in which the increasing popularity of drive-by-wire systems [1] and trends such as autonomous driving [2] require the electrical/electronic (E/E) architecture of modern vehicles to exhibit both a high performance and fail-operational behavior.

*Fail-operational behavior* refers to the property that even if a considered system is affected by errors, it remains able to deliver a certain minimum level of functionality [2], [3]. We refer to a system that is free of catastrophic consequences on users or the environment as *safe* [4] and define a *fail-operational system* as a system that can be safe only if it exhibits fail-operational behavior. It is important to note that for fail-operational systems, it is not possible to identify a safe state that can simply be entered in response to a fault. Therefore, they are generally more challenging to design than conventional, more manageable fail-safe systems.

To achieve fail-operational behavior, suitable fault tolerance techniques must be applied. A fault tolerance technique handles faults of the system in such a way that a specified functionality is maintained [5]. Such a technique is usually based on hardware redundancy, software redundancy, time redundancy or information redundancy [6]. A popular hardware redundancy technique that is able to mask single faults is the triple modular redundancy (TMR) scheme [7]. A drawback of this approach, however, is its significant hardware overhead.

A cost-efficient alternative to TMR are dynamic hardware redundancy techniques. They aim to provide fault tolerance by detecting faults and reconfiguring the system in an appropriate manner—usually by logically removing faulty components from the system and activating suitable spares [7].

In this work, we consider fail-operational systems that comprise a certain degree of inherent redundancy and propose a system-level concept that utilizes this redundancy to provide tolerance against a certain kind of permanent, intermittent, and transient faults. The key aspect of this concept is the introduction of a proxy unit that imposes a dynamic redundancy scheme upon the existing hardware redundancy.

After a review of previous and related work (Section II), we describe the considered fault model and define the problem that this work intends to solve (Section III). Following this, the system-level concept (Section IV), our prototypical implementation (Section V), and the results of its experimental evaluation (Section VI) are presented. As part of the evaluation, we discuss the application of the presented approach to a particular scenario from the automotive domain.

## II. Related Work

The use of dynamic hardware redundancy techniques to tolerate permanent, intermittent, and transient faults has already been studied intensively. The scheme that is presented in [8], for instance, protects a message-based multiprocessor system from physical faults of individual processors or their communication links. The author separates the system's functional

978-1-7281-9535-3/20/$31.00 ©2020 IEEE
DOI 10.1109/DSD51259.2020.00103

layer, which can be viewed as a set of processes and their interactions, from the underlying hardware. A certain degree of redundancy is introduced into the latter. When a physical fault occurs, the affected component is logically removed from the system and its functionality is distributed among the remaining hardware. Therefore, this approach allows the system to maintain its full functionality even after being affected by a certain number of permanent faults in its hardware.

The authors of [9] present a fault tolerance approach that focuses on the uniform treatment of transient and permanent faults in a multiprocessor environment. It is based on a reliability-aware task scheduler that makes use of spare resources to ensure the successful execution of the system's real-time tasks. Another fault tolerance approach for real-time tasks is described in [10]. This approach, however, considers transient faults of homogenous cores in a multicore processor. It makes use of spare resources to execute critical tasks redundantly and employs checkpointing with a rollback mechanism for non-critical tasks without real-time requirements.

It is important to note that a fail-operational system does not necessarily require its full functionality to be maintained after being affected by a fault. Instead, a degraded functionality might be sufficient to preserve its safety. The system-level simplex architecture [11], for instance, makes use of degradation to achieve fail-operational behavior in a control system context. It is based on the general idea of "using simplicity to control complexity" [12] and assumes that two controllers are available to drive a plant: One of them provides a high performance, while the other one is very simple and more reliable. A decision logic module monitors the plant and, at any time, selects one of the controllers as the active one. Whenever the high-performance controller fails to keep the system safe, the simple controller is activated. This dynamic redundancy scheme along with a suitable hardware deployment of the controllers ensures that faults of the high-performance controller cannot impair the safety of the overall system.

Conceptually very similar but defined in the context of automotive electronics is the fault tolerance approach from [13]. It comprises a self-checking main controller and a so-called limp-home controller. Failures of the former are handled by activating a degraded functionality on the latter.

The approach described in [3] is similar to those presented in [11] and [13] in the sense that a failure of a complex processor is handled by migrating its safety-critical functionality to a fallback processor. To reduce the resource consumption of the overall system, however, this fallback processor is not present in the fault-free case but introduced on demand by partially reconfiguring a field-programmable gate array (FPGA).

The formal model described in [14] regards the functionality of a system as a set of so-called features and its underlying hardware as a set of homogeneous, interconnected execution units. The execution units are assumed to contain a certain amount of spare resources. To a certain degree, the failure of such units can therefore be tolerated without degradation. If the available resources are no longer sufficient, individual features can be replaced by their degraded versions.

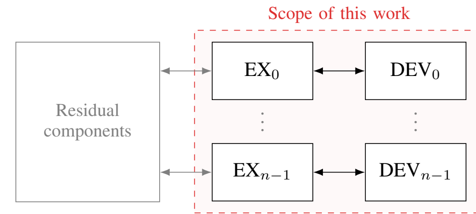


Fig. 1. Inherently redundant system with $n$ execution units

## III. Motivation and Problem Statement

This work is focused on fail-operational systems that comprise a certain degree of inherent redundancy in their execution units. We define an *execution unit* as a processor-based integrated circuit (IC) with all of its internal components (such as flash memory or an input/output controller) and the off-chip peripherals that are immediately attached to it (such as a CAN transceiver). Furthermore, we refer to *inherent redundancy* as the property that a system comprises $n > 1$ execution units, each of which with a certain amount of spare resources that cannot be optimized away and are able to execute additional tasks in case another unit fails. We distinguish two manifestations of these resources:

1) The resources are entirely unused during normal operation of the system, but external factors prevent them from being removed. An execution unit could, for instance, be a commercial off-the-shelf (COTS) product whose configuration cannot be influenced by the developer of the system under consideration.
2) The resources are utilized during normal operation but fulfill only tasks that can safely be suspended at any time. An example of such a task is one whose sole purpose it is to fulfill a non-critical convenience application, such as an infotainment system.

For the purposes of this work, we refer to both of them as *spare resources* and assume that they are or can be made available at any time, especially after the occurrence of a fault.

To fulfill its intended functionality, an execution unit makes use of its input/output controllers and transceivers to interact with a set of so-called *devices*. Some of these devices, such as sensors or actuators, are assumed to be physically constrained to certain locations of the system. Without loss of generality, we assume further that there is a limit to the physical distance that is permitted between an execution unit and its devices. An execution unit that is connected to a device via a conventional Serial Peripheral Interface (SPI) bus, for instance, needs to be within a comparably short distance of the device. Otherwise, issues such as the clock skew that is induced by the propagation delays [15] can render the bus unusable.

A system with $n$ redundant execution units and their respective device sets is shown in Fig. 1. For brevity, the visualization refers to the $i$-th execution unit as $EX_i$ and to its set of devices as $DEV_i$. The communication channels shown in gray represent all connections between an execution unit and the overall system that are excluded from the following considerations, for

example because they are entirely unaffected by the behavior of an execution unit. Note that the device sets are included in the scope of this work only due to the fact that their interaction with the execution units is subject to the physical constraints described above. The dependability of the devices per se is again beyond the scope of this work. In the following, we consider only systems with $n = 2$ execution units.

### A. Definition of the Fault Model

The fault model that serves as the basis for the presented approach can be described as follows: From the system-level perspective, a fault is the complete failure of an execution unit. Such a complete failure can be caused by any number of random and systematic faults within this execution unit. Random faults affect only the hardware portion of the unit and can be of permanent, intermittent, or transient nature. Examples of such random hardware faults are

- the complete and permanent loss of supply voltage,
- a permanent defect of the digital logic within a processing core that causes all its computations to be incorrect, or
- intermittent single-event upsets that repeatedly cause the control flow of a processing core to entirely deviate from its intended execution path.

Systematic faults can affect both the hardware and the software portion of an execution unit. Examples of such faults are

- a hardware issue that in combination with a specific input pattern leads to a deadlock within the digital logic or
- a software bug that in combination with a specific input pattern causes the control flow of a processing core to entirely deviate from its intended execution path.

A single-event upset that causes an incorrect calculation result but not a complete failure of an execution unit does not lead to a system-level fault that this fault model captures. Note that malicious attacks on an execution unit are also not considered as part of this work. The fault model assumes that at any point in time, at most one of the two execution units is faulty, i.e., suffers from a complete failure according to the definition above. Components that are explicitly introduced into the system as part of the proposed concept (such as the lightweight proxy unit) are assumed to be free of systematic faults but susceptible to random hardware faults.

### B. Definition of the Considered Problem

Based on the previous definitions and assumptions, we formulate the problem that this work is looking to solve as follows: Given a fail-operational system with two inherently redundant execution units, modify the system in such a way that it exhibits fail-operational behavior with respect to the complete failure of a single execution unit, does not require the replication of an execution unit, and is able to handle a certain number of random hardware faults within the explicitly introduced components. The second requirement is based on the observation that increasingly complex execution units are being employed in modern embedded systems. This makes their replication more and more expensive. For cost-sensitive applications, it is therefore interesting to research approaches that make use of the existing redundancy.

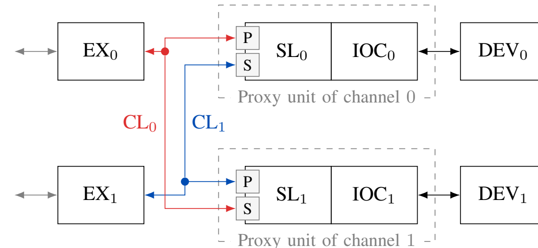


Fig. 2. Introduction of proxy units

The limitation that only one execution unit can become faulty at a time is made with a certain kind of scenarios in mind: In applications in which cost-efficient handling of hazardous situations is more valuable than maintaining operational capability over a long period of time, avoiding physical harm for a short time interval (such as seconds or minutes) and taking other actions might be favorable over tolerating a fault and restoring the initial or comparable fault tolerance characteristics. This kind of fail-operational behavior is similar to the *emergency operation* from ISO 26262 [5]. Therefore, the underlying assumption of this work is that the simultaneous failure of both execution units is sufficiently unlikely.

## IV. System-Level Concept

The concept that we propose is based on the idea that whenever one execution unit fails, the safety-relevant portion of its functionality can be provided by the spare resources of the other one. From the perspective of a set of devices, this is comparable to the system-level simplex architecture: Two execution units (controllers) are available to control the device set (plant). If one execution unit becomes faulty, the other one ensures that the guaranteed minimum functionality is maintained. To reuse the inherent redundancy of the execution units, however, we suggest to superimpose two mirrored instances of this approach. We refer to one such instance (DEV$_i$, EX$_i$ and EX$_{1-i}$) as a *channel* and to the presented concept in its entirety as the *mirrored simplex architecture*.

The key component of the mirrored simplex architecture is a lightweight but sufficiently reliable proxy unit that is inserted into each channel. This results in the system visualized in Fig. 2. A channel is numbered according to the device set that it contains. A proxy unit consists of the so-called *simplex logic* (SL) and the *I/O components* (IOC) that are necessary to connect to the device set. IOC$_i$ logically replaces the I/O components (such as I/O controllers and transceivers) that were previously part of EX$_i$. Therefore, the maximum distance between a set of devices and its proxy unit is limited in the same way as it was previously limited between the set of devices and its execution unit. Via its *primary port* (P) and its *secondary port* (S), the simplex logic is connected to both execution units through *communication links* (CL) whose length limitations are negligible in the considered context.

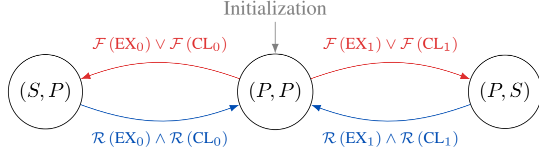


Fig. 3. Transitions between the system modes $(m_0, m_1)$

Therefore, the introduction of the proxy unit lifts the restriction that an execution unit needs to be close to its devices.

The purpose of $\mathrm{SL}_i$ is to orchestrate and enable the dynamic behavior of channel $i$. To achieve this, it activates and deactivates functions on the execution units, detects faults and the recovery of $\mathrm{EX}_i$, and serves as a buffer for $\mathrm{DEV}_i$-related state variables of the execution units. Furthermore, it forwards application-specific payload messages between the active execution unit and $\mathrm{IOC}_i$, thereby enabling an execution unit to control even distant devices that it would not be able to control without this eponymous proxy functionality. Since a proxy unit is in full control of the respective channel, its failure can easily lead to a violation of the overall system's safety requirements. It is therefore necessary to ensure that a proxy unit itself is sufficiently reliable. As part of this concept, we propose to implement it using a TMR scheme to ensure that it is able to mask any kind of single random hardware fault. Note that depending on the exact safety requirements, the application of $N$-modular redundancy schemes with $N > 3$ is also conceivable. In any case, it is of vital importance to minimize the complexity of the SL and IOC blocks to keep the costs of the overall approach at an acceptable level.

### A. *Dynamics of the Architecture*

At any point in time, an $\mathrm{SL}_i$ block is in one of two possible *modes*. They are given by $M = \{P, S\}$ and can be described as follows: $P$ refers to the normal mode in which the simplex logic assumes the execution unit at its primary port to be free of faults and therefore selects it as the component to control $\mathrm{DEV}_i$. $S$ describes the case in which the simplex logic recognizes that the execution unit at its primary port is affected by faults and therefore selects the execution unit at its secondary port to control $\mathrm{DEV}_i$ in a degraded manner. The functionality that this unit delivers in this case will be referred to as the *fallback functionality*. A switch to this functionality corresponds to the activation of an emergency operation.

We use the variable $m_i \in M$ to refer to the mode that the simplex logic of channel $i$ is currently in. Fig. 3 shows the possible mode combinations $(m_0, m_1)$ of the overall system. In the figure, $\mathcal{F}(\cdot)$ refers to the detection of faults within a component and $\mathcal{R}(\cdot)$ to the recognition that a component is not or no longer faulty. Self-transitions are hidden for clarity. In $(P, P)$, the initial mode, every device set is controlled by the execution unit at the primary port of its proxy unit. Detected faults within an execution unit or a communication link lead to the degraded modes $(S, P)$ and $(P, S)$, respectively. In the first case, for instance, $\mathrm{EX}_1$ continues to control $\mathrm{DEV}_1$ while using its spare resources to deliver a fallback functionality for $\mathrm{DEV}_0$. If these resources are made available by suspending non-critical tasks, this corresponds to a degradation of the functionality that $\mathrm{EX}_1$ delivers for $\mathrm{DEV}_1$. It should be noted that the mere selection of an execution unit does not mean that it is actually able to deliver the required functionality. If, e.g., $\mathrm{EX}_0$ is affected by a fault that the simplex logic cannot detect, $\mathrm{DEV}_0$ might be uncontrolled even if $m_0 = P$. Therefore, the capabilities and a sufficient diagnostic coverage of the employed fault detection mechanism are of major importance to achieve fail-operational behavior.

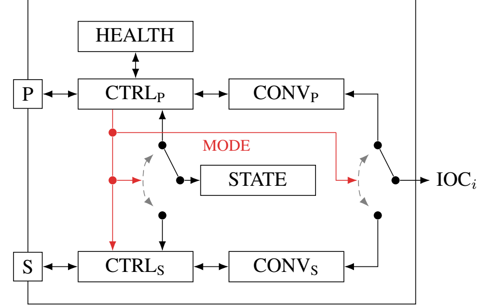


Fig. 4. Functional architecture of the simplex logic ($\mathrm{SL}_i$)

A transition back to $(P, P)$ can restore the initial fault tolerance characteristics after intermittent and transient random or the effects of systematic faults, but it is entirely optional. The underlying worst-case assumption remains that faults are permanent and the system is required to deliver only an emergency operation for a limited amount of time. In such a case, the respective degraded mode is retained.

In order to handle intermittent, transient, or the effects of systematic faults, a temporary switch to one of the degraded modes is not strictly necessary and can create a time overhead. Nevertheless, this architecture treats all kinds of faults in a uniform manner: As soon as a fault is detected, a switch to one of the degraded system modes is performed. This means that the respective simplex logic will immediately isolate the faulty execution unit from its device set by no longer forwarding its messages and request the degraded functionality from the complementary execution unit. The health of the initial execution unit is re-examined only after this switch has occurred. The reason for this procedure is that this health examination takes a certain amount of time. Checking for a certain kind of fault before switching to a degraded mode would create an unnecessary delay in the handling of permanent faults and might take longer than the set of devices is allowed to be in an uncontrolled state.

### B. *Interaction with the Simplex Logic (SL)*

The functional architecture of the simplex logic is visualized in Fig. 4. Two main paths can be identified from the figure: One path leads from the primary port (P) to the STATE block and the IOC output, respectively. The other one leads from the secondary port (S) to these destinations. At any point

TABLE I
CONTROL MESSAGES ON THE CLS

| Message | Sender | Receiver |
|---|---|---|
| RESET | $SL_i$ | $EX_i$ |
| ACTIVATE | $SL_i$ | $EX_i$, $EX_{\Phi(i)}$ |
| DEACTIVATE | $SL_i$ | $EX_{\Phi(i)}$ |
| WDG_CHALLENGE | $SL_i$ | $EX_i$ |
| WDG_RESPONSE | $EX_i$ | $SL_i$ |
| STATE_REQUEST | $EX_i$ | $SL_i$, $SL_{\Phi(i)}$ |
| STATE_RESPONSE | $SL_i$ | $EX_i$, $EX_{\Phi(i)}$ |
| STATE_WRITE | $EX_i$ | $SL_i$, $SL_{\Phi(i)}$ |

in time, $m_i$ decides which of these paths is the active one. For $m_i = P$, it is the former, and for $m_i = S$, the latter. Whenever $m_i = P$, $CTRL_P$ is in control of channel $i$. For this purpose, it periodically reads from and writes to $CL_i$ to

1) forward messages between $EX_i$ and $IOC_i$,
2) receive internal state variables of $EX_i$, and
3) detect faults of $EX_i$ and $CL_i$.

$CONV_P$ is an adapter for the forwarded messages. The received state variables are written into the STATE block, whose purpose it is to prevent the loss of the internal state of $EX_i$ when it or its CL becomes faulty. The fault detection is performed by the HEALTH block. This block triggers watchdog challenges that $EX_i$ has to respond to in a certain manner and within a specified deadline. The execution unit receives a token as part of the challenge and, to determine the correct response, has to perform a pre-defined sequence of computational steps on it. These steps should be simple but involve all the components (such as the arithmetic logic units of all involved processing cores) that are relevant for the detection of a fault according to the fault model.

The detection of a fault activates the secondary path. $CTRL_S$ will then request the EX at the secondary port, $EX_{1-i}$, to launch the fallback functionality of the channel. For brevity, we define $\Phi(i) := 1 - i$ and refer to this EX as $EX_{\Phi(i)}$. After this request, $EX_{\Phi(i)}$ will read the most recently saved state from the STATE block and start to deliver the fallback functionality of channel $i$. Note that this fallback functionality continues to update the STATE block to allow for a possible switch back. As soon as $EX_{\Phi(i)}$ controls the channel, $CTRL_P$ resets $EX_i$ and uses the HEALTH block to check if this eliminates the fault that caused the mode transition. In this case, it can switch back to $m_i = P$ by requesting $EX_i$ to resume the full functionality. $EX_i$ will then once again read the most recently saved state from STATE and do so. At the same time, $CTRL_S$ notifies $EX_{\Phi(i)}$ that it is no longer required to deliver the fallback functionality. After this, $(m_0, m_1) = (P, P)$ and both execution units communicate with the SL block of their corresponding channel only.

### C. Signals on the Communication Link (CL)

From the perspective of an application, the CL serves as the communication path that it needs to read from and write to its devices. However, a CL also transmits the eight

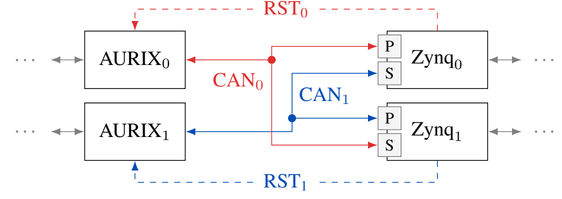


Fig. 5. Prototypical implementation of the proposed concept

architecture-specific control signals shown in Table I. Using the RESET message, the SL block resets the EX at its primary port after switching to the fallback functionality. ACTIVATE refers to the message that $SL_i$ sends to $EX_i$ or $EX_{\Phi(i)}$ to request the delivery of the respective functionality for $DEV_i$. DEACTIVATE is sent to $EX_{\Phi(i)}$ to terminate its fallback functionality. WDG_CHALLENGE and WDG_RESPONSE transmit the watchdog interactions. STATE_REQUEST requests the most recently saved state from an SL block. After receiving such a request, the SL block uses the STATE_RESPONSE message to return this state. State variables are written to an SL block using the STATE_WRITE message. Note that an EX needs to be able to distinguish if a message originates from a primary or a secondary port to determine its exact meaning.

## V. PROTOTYPICAL IMPLEMENTATION

For a practical implementation of the concept, the execution units of the original system need to be modified in such a way that they are connected to the CLs and conform to the protocol that the mirrored simplex architecture imposes upon them. Furthermore, the proxy units have to be designed and manufactured using a suitable technology. Due to its application-specific nature and the requirement to be both cost-efficient and reliable, we propose to implement a proxy unit as an application-specific integrated circuit (ASIC) in a TMR arrangement with attached I/O transceivers. This way, it can be optimized for reliability while keeping the production costs small—especially for large quantities.

As part of this work, we performed a prototypical implementation of the concept. It is based on the following example scenario from the automotive domain: A front-wheel drive vehicle with electric wheel hub motors comprises an electronic control unit (ECU) for each of the two motors. Each of the ECUs receives an individual setpoint such as the target rotation speed, reads in measurement values from the sensors of its respective motor, and transmits pulse-width modulation (PWM) signals to the power electronics of its respective motor. Note that this setup is symmetrical in the sense that both ECUs perform the same functionality with respect to the motor they control. To avoid faults of an ECU leading to catastrophic consequences, the mirrored simplex architecture shall be applied. In this scenario, the ECUs correspond to the execution units, while the protected devices are the power electronics and sensors on either side of the vehicle. The architecture of our implementation is visualized in Fig. 5. We employed an AURIX microcontroller from Infineon as

execution unit and the FPGA portion of a Zynq-7000 device from Xilinx as the platform to realize the TMR arrangement of proxy units.[1] A reset wire and its corresponding CAN bus represent a CL. Note that the Zynq itself is a system on chip (SoC) that comprises not only the FPGA. Its hard-wired processor will be used during the experimental evaluation.

We used the above-mentioned example scenario as a starting point and extended the generic framework with the following application-specific functionality: Each proxy unit's IOC block consists of a PWM controller and a set of general-purpose inputs. Both the full and the degraded AURIX applications run with a period of $T_{\text{cycle}}$ to repeatedly

1) read 32 bits of payload data from their IOC block,
2) write 32 bits of payload data to their IOC block, and
3) write $N_{\text{state}}$ variables (32 bit) to their STATE block.

We assume that the computational effort of the degraded functionality is considerably smaller than that of the full one and, therefore, that an execution unit is able to deliver both functionalities at the same time. For the implementation, this means that if a proxy unit requests the degraded functionality from its complementary AURIX, this execution unit continues to deliver the full functionality for its primary device set. The employed AURIX is a multicore platform that comprises three processing cores. On such a platform, it is conceivable to execute the two functionalities truly in parallel. In this implementation, however, they share the same core.

Except for the `RESET` message, all control messages from Table I as well as the application-specific payload messages are exchanged using one of the CAN buses. The message IDs have been chosen so that transmissions between $\text{SL}_i$ and $\text{EX}_i$ take precedence over transmissions between $\text{SL}_i$ and $\text{EX}_{\Phi(i)}$.

The watchdog challenges that a processor has to respond to are issued by the HEALTH block with a configurable period of $T_{\text{wdg}} \leq T_{\text{cycle}}$. Such a challenge consists of a randomly generated 8-bit token that the processor needs to modify in a predefined way. After issuing a challenge, the HEALTH block expects to receive the modified token from the processor within a specified deadline. If this deadline is missed or the HEALTH block receives an incorrect response, a fault of the processor or the CAN bus that it connects to is assumed.

As soon as $\text{SL}_i$ requests a degraded functionality from the execution unit at its secondary port, the load on $\text{CL}_{\Phi(i)}$ increases significantly. For the implementation, we assume that within one $T_{\text{cycle}}$-interval, $\text{CAN}_{\Phi(i)}$ is able to transmit

1) all the messages between $\text{EX}_{\Phi(i)}$ and $\text{SL}_{\Phi(i)}$ that are nominally scheduled for this interval,
2) the messages that $\text{EX}_{\Phi(i)}$ and $\text{SL}_i$ exchange solely for the purpose of the mode transition, and
3) the first sequence of payload and state messages that are exchanged between $\text{EX}_{\Phi(i)}$ and $\text{SL}_i$.

According to [16], the maximum transmission time of a CAN message with an 11-bit identifier and $s$ data bytes

[1]The execution and proxy units comprise not only these ICs but also their directly attached components (such as I/O transceivers). For brevity, these components are regarded as logically belonging to the ICs.

TABLE II
LOAD FACTOR VALUES FOR DIFFERENT CONFIGURATIONS

| | | Number of 32-bit state variables ($N_{\text{state}}$) | | | | |
|---|---|---|---|---|---|---|
| $T_{\text{cycle}}$ | $T_{\text{wdg}}$ | 1 | 2 | 4 | 8 | 16 |
| 1 ms | 1 ms | 0.92 | 1.20 | 1.77 | 2.91 | 5.19 |
| 2 ms | 1 ms | 0.52 | 0.67 | 0.95 | 1.52 | 2.66 |
| 2 ms | 2 ms | 0.46 | 0.60 | 0.89 | 1.46 | 2.60 |
| 4 ms | 1 ms | 0.33 | 0.40 | 0.54 | 0.83 | 1.40 |
| 4 ms | 2 ms | 0.26 | 0.33 | 0.48 | 0.76 | 1.33 |
| 4 ms | 4 ms | 0.23 | 0.30 | 0.44 | 0.73 | 1.30 |

is given by $C_m(s) := (55 + 10s)\,\tau_{\text{bit}}$, where $\tau_{\text{bit}}$ describes the transmission time of one bit. Since the implementation uses 11-bit identifiers, this formula can be used to calculate the portion of the considered $T_{\text{cycle}}$-interval with messages on $\text{CAN}_{\Phi(i)}$. With $k_{\text{wdg}} := \lceil T_{\text{cycle}}/T_{\text{wdg}} \rceil$ watchdog challenges during the interval and under the assumption that no messages need to be retransmitted, the utilized time is given by

$$\begin{aligned}\tau_{\text{sum}} &= \underbrace{\tau_{\text{wdg}} + \tau_{\text{state}} + \tau_{\text{payload}}}_{(1)} + \underbrace{\tau_{\text{activation}} + \tau_{\text{state}}}_{(2)} \\ &\quad + \underbrace{\tau_{\text{payload}} + \tau_{\text{state}}}_{(3)} \\ &= 2C_m(1) \cdot k_{\text{wdg}} + 3C_m(4) \cdot N_{\text{state}} + 4C_m(4) \\ &\quad + C_m(1) + C_m(0) \\ &= (130k_{\text{wdg}} + 285N_{\text{state}} + 500)\,\tau_{\text{bit}}.\end{aligned}$$

This corresponds to a relative utilization or a *load factor* of

$$\eta = \frac{\tau_{\text{sum}}}{T_{\text{cycle}}} = \frac{(130k_{\text{wdg}} + 285N_{\text{state}} + 500)\,\tau_{\text{bit}}}{T_{\text{cycle}}}.$$

Table II shows the load factor for a CAN frequency of 1 MHz and various combinations of $T_{\text{cycle}}$, $T_{\text{wdg}}$, and $N_{\text{state}}$. Scenarios with a load factor of $\eta > 1$ are grayed out as they violate the requirement defined above and are considered infeasible.

## VI. EVALUATION AND RESULTS

To evaluate the prototypical implementation, we created a setup based on two TriBoards and two ZedBoards. Each of the TriBoards was mounted with an AURIX of type TC277, while the ZedBoards featured a Zynq device of type XC7Z020. Synthesizing the RTL description of a single proxy unit for the FPGA portion of an XC7Z020 leads to the resource utilization values shown in Table III. As shown in the table, the CAN controllers for the primary and secondary ports are the most resource-intensive components of this design. The actual simplex logic consumes less than half of the total number of required LUTs, registers, and BRAMs. Note that the "interconnections" category comprises the logic that serves as an adapter between the custom modules and the IP cores that implement the primary and secondary CAN ports.

In the evaluation setup, an FPGA comprises a TMR arrangement of proxy units and a fault injection module. The resource utilization of one such FPGA is shown in Table IV.

TABLE III
Resource Utilization of the Proxy Unit

| Module | Resource type | | |
|---|---|---|---|
| | LUTs | Registers | BRAMs |
| $\text{CAN}_i$ controller | 659 | 536 | 2 |
| $\text{CAN}_{\lambda(i)}$ controller | 661 | 536 | 2 |
| $\text{CTRL}_\text{P}$, $\text{CONV}_\text{P}$, HEALTH | 333 | 348 | 0 |
| $\text{CTRL}_\text{S}$, $\text{CONV}_\text{S}$ | 124 | 246 | 0 |
| STATE | 69 | 3 | 1 |
| PWM controller | 28 | 22 | 0 |
| Interconnections | 108 | 121 | 0 |
| Total | 1,982 | 1,812 | 5 |

TABLE IV
Resource Utilization in the Evaluation Setup

| Module | Resource type | | |
|---|---|---|---|
| | LUTs | Registers | BRAMs |
| TMR arrangement | 6,037 | 5,538 | 15 |
| Fault injection module | 904 | 1,875 | 0 |
| Total | 6,941 | 7,413 | 15 |

It corresponds to 13.05 % of the LUTs, 6.97 % of the registers, and 10.71 % of the BRAM tiles available on the XC7Z020.

When the concept was described in Section IV, we referred to the modes that an SL block can be in as $P$ and $S$. It is important to understand, however, that when a channel experiences a transition from $P$ to $S$ or vice versa, there is a certain amount of time in which neither the full nor the degraded functionality of the channel is delivered. We refer to such an interval as a *mode transition*. The duration of these transitions is crucial when deciding whether a system meets its fail-operational requirements. This is especially true for a transition from $P$ to $S$, most importantly because random or the effects of systematic faults appear unexpectedly and require immediate initiation of such a transition. Transitions into the opposite direction, i.e., after the recovery of an execution unit, can be performed in a more controlled manner. We therefore focused on the $P \rightarrow S$ transition of the implementation and performed an experimental evaluation of its duration.

Fig. 6 illustrates the steps that constitute a mode transition of channel $i$ in the $P \rightarrow S$ direction. A fault occurring at $t = t_0$ must first be detected by the HEALTH block of the respective simplex logic ($\text{SL}_i$). As soon as this has happened, from $t = t_1$ onwards, $\text{SL}_i$ stops to react to messages from $\text{EX}_i$ and requests the fallback functionality from $\text{EX}_{\Phi(i)}$. Then, from $t = t_2$ to $t = t_3$, it transfers the most recently written set of state variables to $\text{EX}_{\Phi(i)}$. Based on the definitions from [5], we refer to the duration of the first interval as *fault detection time*, to the combined duration of the latter two intervals as *fault reaction time*, and to their sum as *fault handling time*.

Due to symmetry in the setup, our evaluation focused on the fault handling characteristics of only one channel. More specifically, we used the hard-wired processor of and the fault injection module implemented on the $\text{Zynq}_0$ device to temporarily interrupt the supply voltage of $\text{AURIX}_0$ and perform an automated measurement of the individual fault handling intervals. For every combination of $T_\text{cycle}$, $T_\text{period}$, and $N_\text{state}$ that is not grayed out in Table II, 300 independent measurements of this kind were performed. In all cases, a watchdog tolerance of 500 µs was used. For the sake of brevity, only the results for a cycle time of $T_\text{cycle} = 4\,\text{ms}$ will be considered in the following. These results are shown in Fig. 7, where square markers indicate the average values and vertical bars represent the measured range of values.

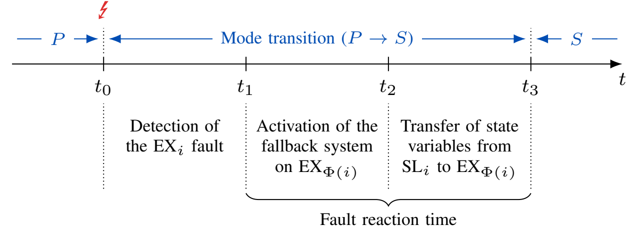


Fig. 6. Schematic visualization of the $P \rightarrow S$ mode transition intervals

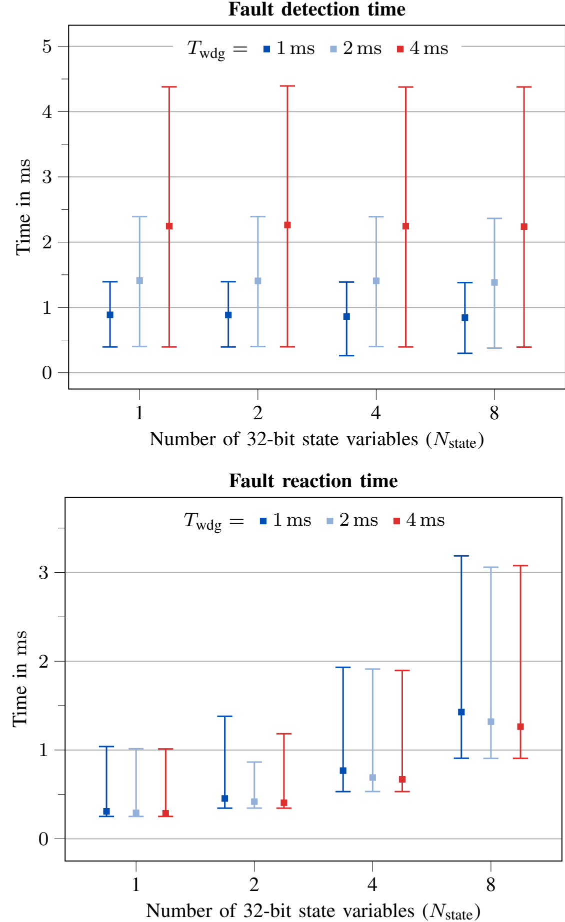


Fig. 7. Measured fault handling times for a system with $T_\text{cycle} = 4\,\text{ms}$

As shown in the first plot, the fault detection times did not exceed $1.5\,\mathrm{ms}$ for $T_{\text{wdg}} = 1\,\mathrm{ms}$, $2.5\,\mathrm{ms}$ for $T_{\text{wdg}} = 2\,\mathrm{ms}$, and $4.5\,\mathrm{ms}$ for $T_{\text{wdg}} = 4\,\mathrm{ms}$. Furthermore, the plot demonstrates that the size of the synchronized internal state of an application had no significant influence on the fault detection time.

The second plot shows that the average fault reaction time increased with the size of the synchronized internal state. This is not only due to the increased number of messages that are necessary to read the state from $\mathrm{SL}_i$ but also due to the fact that for higher $N_{\text{state}}$ values, more state messages will be exchanged between $\mathrm{EX}_{\Phi(i)}$ and $\mathrm{SL}_{\Phi(i)}$. Since they belong to the primary port of $\mathrm{SL}_{\Phi(i)}$, they have a higher priority than the messages between $\mathrm{EX}_{\Phi(i)}$ and $\mathrm{SL}_i$. For the same reason, an increase of $T_{\text{wdg}}$, i.e., issuing watchdog challenges less frequently, resulted in less messages with a higher priority and, therefore, decreased the average fault reaction time slightly.

In a realistic implementation of the wheel hub motor control, the payload data that is exchanged between an EX and its IOC amounts to approximately 32 bits per cycle and direction. The internal state variables that need to be repeatedly written to the SL block do not exceed 256 bits, which corresponds to $N_{\text{state}} = 8$. Such a system can operate with $T_{\text{cycle}} = 4\,\mathrm{ms}$ and $T_{\text{period}} = 1\,\mathrm{ms}$. During the evaluation, it showed a maximum fault detection time of $1.380\,\mathrm{ms}$ as well as a maximum fault reaction time of $3.187\,\mathrm{ms}$. This corresponds to a fault handling time of no more than $4.567\,\mathrm{ms}$. The length of the example scenario's fault tolerant time interval (FTTI), which is defined as the "minimum time-span from the occurrence of a fault in an item to a possible occurrence of a hazardous event, if the safety mechanisms are not activated" in [5], ranges between $50\,\mathrm{ms}$ and $100\,\mathrm{ms}$. Therefore, we consider the achieved fault handling time to be sufficiently small to achieve fail-operational behavior in this specific context.

## VII. Conclusion and Future Work

In this work, we focused on fail-operational systems with a certain degree of inherent redundancy and proposed the concept of the mirrored simplex architecture. It makes use of spare resources in a pair of redundant execution units and dynamically migrates essential functions of a completely failed unit to its redundant counterpart. Our prototypical implementation of the concept shows that it exhibits short fault handling times and—given that at most one unit fails at a time and that the employed challenge-response watchdog exhibits a sufficient diagnostic coverage with respect to the relevant safety requirements—eliminates the need to replicate the execution units in order to achieve fail-operational behavior. Therefore, the presented approach is particularly suited to meet fault tolerance requirements of fail-operational systems in cost-sensitive fields such as the automotive domain.

A possible starting point for further research activities is the question of how to scale this approach up to an arbitrary number of channels. A scalable concept with $n > 2$ channels could, most importantly, remove the current limitation that only one execution unit at a time may fail. Furthermore, adapting the concept to handle certain types of transient and intermittent faults more efficiently or replacing the CAN bus of the current implementation with different alternatives to optimize the achievable throughput and to provide rigorous real-time guarantees are further topics for future work.

## Acknowledgment

This work was funded by the German Federal Ministry of Education and Research (BMBF) under grant number 01IS16025 (ARAMiS II). The responsibility for the content of this publication lies with the authors.

## References

[1] S. Shreejith, S. A. Fahmy, and M. Lukasiewycz, "Reconfigurable Computing in Next-Generation Automotive Networks," *IEEE Embedded Systems Letters*, vol. 5, no. 1, pp. 12–15, Mar. 2013.

[2] R. Ernst, "Automated Driving: The Cyber-Physical Perspective," *Computer*, vol. 51, no. 9, pp. 76–79, Sep. 2018.

[3] T. Dörr, T. Sandmann, F. Schade, F. K. Bapp, and J. Becker, "Leveraging the Partial Reconfiguration Capability of FPGAs for Processor-Based Fail-Operational Systems," in *Applied Reconfigurable Computing (ARC 2019)*, C. Hochberger, B. Nelson, A. Koch, R. Woods, and P. Diniz, Eds. Cham: Springer International Publishing, 2019, pp. 96–111.

[4] 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*, vol. 1, no. 1, pp. 11–33, 2004.

[5] ISO 26262-1:2018, "Road vehicles — Functional safety — Part 1: Vocabulary," Geneva, Dec. 2018.

[6] N. R. Storey, *Safety-critical computer systems*. Addison-Wesley, 1996.

[7] V. B. Prasad, "Fault tolerant digital systems," *IEEE Potentials*, vol. 8, no. 1, pp. 17–21, Feb. 1989.

[8] S. M. Ellis, "Dynamic software reconfiguration for fault-tolerant real-time avionic systems," *Proceedings of the 1996 Avionics Conference and Exhibition*, vol. 21, no. 1, pp. 29–39, Jul. 1997.

[9] J. Huang, J. O. Blech, A. Raabe, C. Buckl, and A. Knoll, "Analysis and Optimization of Fault-Tolerant Task Scheduling on Multiprocessor Embedded Systems," in *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, Taipei, Taiwan, 2011, p. 247.

[10] M. H. Mottaghi and H. R. Zarandi, "DFTS: A dynamic fault-tolerant scheduling for real-time tasks in multicore processors," *Microprocessors and Microsystems*, vol. 38, no. 1, pp. 88–97, Feb. 2014.

[11] 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 (RTAS 2009)*. San Francisco, CA, USA: IEEE, Apr. 2009, pp. 99–107.

[12] L. Sha, "Using Simplicity to Control Complexity," *IEEE Software*, vol. 18, no. 4, pp. 20–28, Aug. 2001.

[13] M. Ghadhab, M. Kuntz, D. Kuvaiskii, and C. Fetzer, "A Controller Safety Concept Based on Software-Implemented Fault Tolerance for Fail-Operational Automotive Applications," in *Formal Techniques for Safety-Critical Systems (FTSCS 2015)*. Cham: Springer International Publishing, 2016, pp. 189–205.

[14] K. Becker and S. Voss, "A Formal Model and Analysis of Feature Degradation in Fault-Tolerant Systems," in *Formal Techniques for Safety-Critical Systems (FTSCS 2015)*, C. Artho and P. C. Ölveczky, Eds. Cham: Springer International Publishing, 2016, pp. 139–154.

[15] T. Kugelstadt, "Extending the SPI bus for long-distance communication," *Texas Instruments, Analog Applications Journal*, vol. Q4 2011, 2011.

[16] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, Apr. 2007.