Enhancing Fault Tolerance in ARINC 653-compliant Partitioned Systems: A Focus on Dynamic Reconfiguration

***Abstract*—Avionic systems are critical for aircraft and spacecraft, incorporating key subcomponents such as Collision Avoidance Systems (CAS) and Terrain Awareness and Warning Systems (TAWS). The software within these systems must be resilient against failures, often using hypervisors to separate subcomponents into partitions that share CPU resources. However, these systems are subject to dynamic constraints, failures, or attacks, which can render precompiled configurations inflexible and insufficient. To address these challenges, there is a growing need for adaptive systems that can dynamically respond to changing operational and security conditions. One promising approach is dynamic reconfiguration, where alternative subcomponents take over from failing ones. For example, if camera vision fails in the dark, a RADAR system can compensate. However, ARINC 653-compliant hypervisors offer limited support for reconfiguration at run-time due to their fixed inter-partition communication structure. In this work, we introduce a "mitigator", along with routing and health monitoring components, to handle run-time partition failures within an ARINC 653-compliant environment. The mitigator dynamically reconfigures the system using redundancies and a run-time scoring scheme to replace failed partitions and optimize the use of remaining ones, ensuring continued functionality and performance. Our evaluation measures the system's adaptation speed and output coverage, revealing a trade-off between fault tolerance and processing overhead.**

***Index Terms*—Fault Tolerance, FDIR, Dynamic Reconfiguration, Safety-Critical Systems, Avionics, ARINC 653**

I. Introduction

Avionic systems are one of the most critical components of any aircraft or spacecraft. Hosting many critical subcomponents, such as a Collision Avoidance System (CAS) and a Terrain Awareness and Warning System (TAWS), the software used in such systems must be robust and resilient to failures. A common approach to meet this requirement is to use a virtualization technique between the different subcomponents, specifically a hypervisor. Hypervisors separate the subcomponents in time and space into so-called partitions, which receive a time share of the Central Processing Unit (CPU) according to a given static schedule. Depending on the exact domain and application scenario, avionic systems must deal with dynamic constraints and requirements from the environment. Furthermore, some of their subcomponents may be subject to failures or attacks. Therefore, avionic systems must be able to reconfigure themselves to proactively prevent such failures while maintaining overall system performance—ideally avoiding the need for recovery altogether. Instead of reacting only after failures occur, modern systems should evaluate potential configurations based on current system health, environmental constraints, and performance potential. In architectures offering multiple degrees of freedom—such as modular partitioning—this enables more intelligent and adaptive decision-making. Current state-of-the-art approaches for reconfiguring avionics systems, such as those in [7] and [3], often rely on static configurations that are either precompiled or selected based on predefined conditions. In such setups, static performance scores may be assigned to configurations, with reconfiguration permitted when a higher-scoring setup becomes feasible. However, system-wide reconfiguration based solely on these scores introduces the risk of instability due to configuration oscillation, especially when failures only become apparent during active use [15]. For instance, if a faulty partition is removed, its issue becomes hidden. The system may then revert to a previous configuration that reintroduces the same fault—resulting in a reconfiguration loop.

While static setups are straightforward to implement, they often fall short in real-world scenarios where conditions can shift rapidly, faults may be transient, and optimal performance demands contextual flexibility. As a result, there is a growing need for systems that can dynamically adapt and recover from run-time changes. A promising solution is to enhance the avionic system with the ability to score and reconfigure itself at run-time. The system will then be able to use alternative subcomponents to achieve the same functional goals. For example, while camera vision can enhance CAS with detailed visual context, it may fail in dark or low-visibility conditions. In such cases, a more reliable Radio Detection and Ranging (RADAR)-based component can take over, ensuring continuous functionality.

In this work, we adopt a static schedule in which all partitions are executed in every major frame cycle. This approach ensures that the system continuously monitors the outputs of all partitions, including those previously marked as faulty. Such constant observability is crucial for detecting recovery and enabling dynamic reconfiguration decisions at run-time. In contrast, one might consider dynamically reconfiguring the major frame to selectively execute only active or healthy partitions, thereby reducing overall execution time and improving efficiency. While ARINC 653 allows this through switching between pre-computed schedules using SET_MODULE_SCHEDULE, partitions excluded from a schedule can no longer be monitored, which limits the ability to detect recovery. Furthermore, even with all partitions running continuously, ARINC 653 still imposes

limitations on inter-partition communication. Communication channels—such as sampling and queuing ports—must be statically defined, with each port connected to only a single source.

To solve this reconfiguration issue, we introduce an additional component responsible for routing messages according to the determined configuration, which was partially addressed in [Reference omitted for blind review]. While this component introduces latencies, it enables the system's communication channels to be dynamically reconfigurable. In doing so, we explicitly constrain ourselves to achieving dynamic reconfiguration using only mechanisms that remain fully compliant with the ARINC 653 standard. For our reconfiguration strategy, we propose a partition-level dynamic reconfiguration approach. Rather than relying on static, system-wide configurations, our method continuously evaluates each partition individually at run-time—even those not currently in active use. As a result, the system can achieve localized fault isolation, reduce configuration oscillation, and enhance overall responsiveness. To implement this dynamic reconfiguration strategy, we focus on a key component called the "mitigator". The mitigator is responsible for addressing run-time failures in partitions managed by the hypervisor. When a partition fails, the mitigator identifies suitable replacement partitions to maintain system functionality and reliability based on our run-time scoring scheme. Its goal is to dynamically reconfigure the system not only to recover from failures but also to proactively maintain optimal performance. In cases of redundancy (i.e., multiple partitions providing the same functionality), it ensures that flight data—such as altitude advisories, intruder positions, and own-ship position—is provided by the best-performing partitions, potentially preventing future failures.

To evaluate our system, we previously analyzed the latencies introduced by the router [Reference omitted for blind review], and now augment this with the analysis of time required for fault detection and mitigation. Finally, we evaluated the mitigator as a standalone partition by measuring the flight data average channel coverage and mitigation time. Overall, we were able to demonstrate that higher fault levels negatively impact coverage and increase mitigation time. Increasing redundancy, either through channel mappings or by adding more backup partitions, can help mitigate these effects but introduces additional processing costs. The mitigator performed best in scenarios with low fault levels and moderate redundancy, where the average channel coverage remained high and mitigation times were reasonable.

The rest of the paper is organized as follows: Section II reviews the state of the art. In section III, we give an overview of our system. In Section IV, we introduce our mitigation approach. This approach is evaluated in Section V and further discussed in Section VI.

II. State of the art

Fault Detection, Isolation, and Recovery (FDIR) is a crucial principle for every avionic software stack. Employing FDIR usually includes simple mechanics as Triple Modular Redundancy (TMR) [9], N-Version programming [5] or providing spares. All these techniques are good and common practices, however, they can't provide a 100% safety, especially when it comes to more complex software. Those techniques cannot provide guarantees that other software components in the same system have an influence on the execution of the software. A mixed-criticality system is therefore not feasible and different components were often put into their own capsulated system consisting of specialized hardware and software.

In recent years, Integrated Modular Avionics (IMA) [10] has become the principle of choice for enabling mixed-criticality applications. A common approach to mitigate failures in IMA based architectures is the principle of reconfiguration. Here, a lost or failed component is mitigated by reconfiguring the system with the remaining resources to restore desired functionality. Reconfiguration often focuses on hardware-level adaptation, such as reconfiguring FPGAs [13] at run-time to maintain system availability, as demonstrated by Romero et al. [12], who explored the use of partial dynamic reconfiguration to adapt FPGA-based computing modules on the fly. In contrast, other approaches—like the one proposed in this paper—focus on software-level reconfiguration. Among these, Bieber et al. [3] proposed a design that leverages redundancy in reconfiguring IMA platforms through the use of pre-calculated, certifiable directed acyclic configuration graphs. The platform supports reconfiguration by switching between these validated configurations at run-time, however, it lacks the ability to dynamically generate or adapt new configurations in response to unforeseen situations.

The work of Engel et al. [7] developed a decentralized reconfiguration approach to improve fault tolerance and maintain critical functions even under component failures. Their system is based on a predefined set of mappings between applications and modules. Each module is responsible for executing the necessary applications for health monitoring and reconfiguration. In the event of a module failure, the remaining modules use a Byzantine agreement protocol to collectively decide on a transition to the next configuration. However, despite this level of coordination, they still rely on static setups defined at design time, without the flexibility to generate new configurations dynamically at run-time. Dusch et al. [6] explore the migration of applications away from failed cores, but also propose adding a dynamic scheduling layer to the otherwise static XNG hypervisor [11] to handle deadline overruns in safety-critical applications. While this enhances flexibility, it risks violating temporal isolation guarantees [15]. In contrast, the authors of [15] propose dynamic reconfiguration through three distinct "static" scheduling strategies.

The first approach is *switching between pre-computed static schedules for reconfiguration*, with each schedule defining a different set of partitions and applications. When a partition is excluded from a schedule, its resources are freed, but it can no longer be actively monitored. This approach benefits from determinism and design-time validation of all schedules. However, it faces scalability issues, synchronization

challenges, and platform-imposed limits on the number of supported schedules [15].

The second is *switching applications inside of partitions*, which uses generic partitions, each containing all applications but executing only one at a time. Upon reconfiguration, the active application in a partition is switched dynamically without altering the global schedule. This allows fast, flexible changes. However, it increases complexity—requiring all apps to be compiled together in one partition image—and poses spatial isolation and software assurance challenges [15].

The third approach, which we adopt, uses a *static schedule with dynamic communication routing*. Here, all partitions execute in every cycle, allowing continuous health monitoring. Reconfigurability is achieved by rerouting messages between partitions dynamically based on real-time health data. To support this, we introduce an additional routing layer, since ARINC 653 requires statically defined communication channels that cannot be changed at run-time. At the heart of this setup is the *mitigator*, which evaluates partition performance at runtime and triggers the routing layer to favor the best-performing partitions. This enables adaptive, fine-grained control over data flow—achieving ARINC 653-compliant reconfigurability.

III. System Overview

Our system consists of interconnected sub-components that collaborate to implement a reconfigurable partitioned architecture managed by an ARINC 653-compliant hypervisor. Fig. 1 illustrates how the system components communicate and interact within scheduled time windows managed by the hypervisor. Each partition communicates via a central router that receives all outputs and has access to all input channels. Health events and output data from each partition are forwarded to a dedicated health monitor. The health monitor analyzes this data and generates a health report for each partition. These reports are forwarded to the mitigator, which uses a run-time scoring scheme to evaluate each partition and generate a reconfiguration decision (see Sec. IV-D), which is then translated into a new routing table and passed back to the router. The router then uses this updated table to dynamically adjust data flow, routing messages only from the best-performing partitions to their respective consumers, as illustrated in Fig. 2. Next, we describe each system component in detail.

A. Hypervisor

With modern IMA systems, we consolidate software of different criticality on the same avionic computer. For maintaining isolation guarantees which were previously provided by running software on different machines, hypervisors are utilized as a segregation and management layer between computer and software. These hypervisors prevent software from interfering with each other by protecting their memory and enforcing static time windows [8].

For our purposes, we use the XtratuM Next Generation (XNG) hypervisor from fentISS in combination with the ARINC 653-compliant LithOS for demonstration as well as our own Linux-based hypervisor for fast development. The XNG hypervisor is known for its use in aircraft [6] and offers us a solution which can be quickly compiled for our Xilinx Zynq-7000 development boards [14]. On the other hand, the Linux based hypervisor, together with our ARINC 653 abstraction library, gives us the means to focus on the functional development of our mitigator partition while allowing us to use the same partition code for both hypervisors in development and deployment [Reference omitted for blind review].

B. Application Partitions

We call all partitions other than the health-monitor, router, and mitigator *application partitions*. This is to distinguish them from partitions that only implement the reconfiguration concepts. Examples of application partitions include driver partitions with direct access to the memory region of devices as well as processing partitions which process some incoming data while generating new data.

C. Reconfiguration Partitions

We refer to the partitions that implement our reconfiguration concepts, the health monitor, router, and mitigator, *reconfiguration partitions*, to distinguish them from system level partitions, since they require no additional privileges beyond those provided to regular application partitions to perform their tasks. However, since they control the communication between application partitions, they fulfill system-level partition roles and should be considered of similar criticality.

1) Health Monitor: The health monitor partition has two main responsibilities: to obtain observations from the application partitions, and to estimate their health based (not only) on these observations. We use a cooperative model in which application partitions are responsible for communicating values or measurements relevant to their health, including a regular heartbeat communication. There are several *health estimators*, which provide independent health estimates of each application partitions based on different inputs. The outputs of the health estimators are estimates in the $[0, 1]$ interval. We aggregate the outputs of the different estimators per partition by reporting the lowest overall health estimate to the mitigator, as well as which estimator is ultimately responsible (see Sec. IV-A).

Health estimators include presence and plausibility checking of the heartbeat messages, range checking of measurements, and a model-free anomaly detector (based on deviations from previously observed data patterns, not a predefined model). We also monitor the execution times of partitions, as well as their reporting behavior in terms of message rate, and allow for partitions to report their own health issues for a comprehensive estimate.

Our communication-focused dynamic reconfiguration approach allows us to keep all application partitions running, even if they are not contributing to the system at any given moment. This in turn allows for valid health estimates of all partitions, even passivated ones (see Sec. IV-F). As a consequence, there is a reduced potential of oscillation between

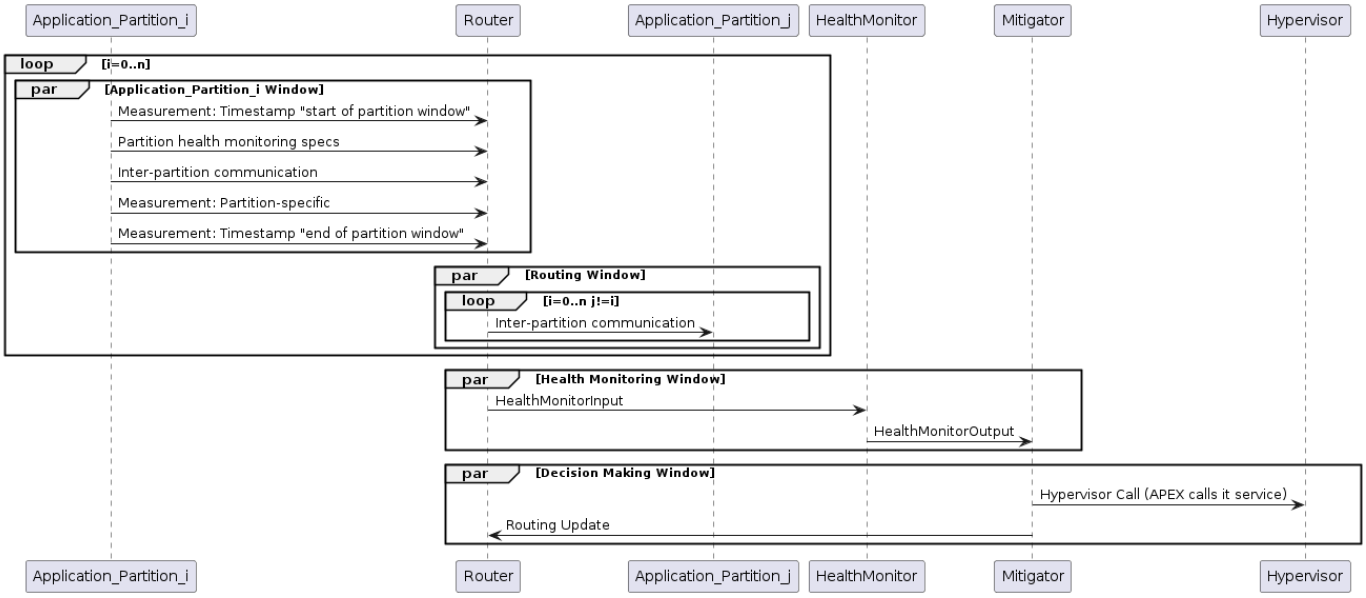

Fig. 1. Sequence diagram showing partition execution, health monitoring, mitigation, and routing within scheduled time windows.

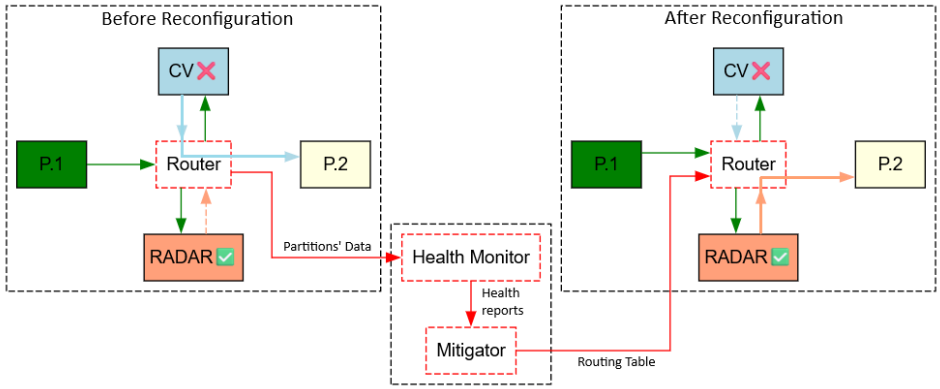

Fig. 2. Example run-time reconfiguration: Camera Vision (CV) output replaced by RADAR output.

configurations and a quicker recovery if partitions become healthy again.

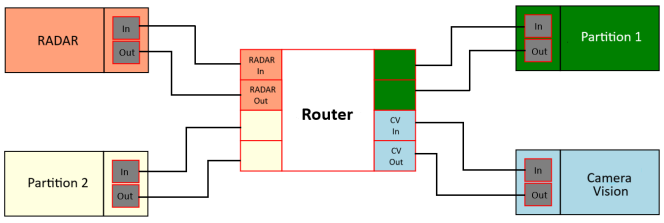

Fig. 3. Example modified inter-partition communication of a channel for use with dynamic reconfiguration.

2) Router Partition: The router serves as the central point from which to execute mitigation steps. It runs as its own ARINC 653 partition and has the purpose of forwarding flight data between the application partitions. For this, each channel between individual application partitions is replaced by a pair of channels of which one port each resides inside the router and one in the respective application partition, as shown in Fig. 3.

During system integration, a parameter data item is generated that details the ports made available to the router and the initial forwarding table that connects these ports. At initialization time, the router then uses this parameter data item to initialize itself. The router offers an interface exclusively to the mitigator by which the mitigator can instruct the router to update its forwarding table to match a new system configuration. Since the router does not store any run-time-dynamic state besides the currently forwarded message and the current forwarding table, it can be seamlessly reconfigured without the risk of partial reconfiguration. The transmission time available during the partition window of the router is distributed fairly among the active channels using a round-

robin scheduler and static sampling frequencies configured as part of the same configuration data item. The time required for transmitting all active channels, along with the time needed for reconfiguration, is included in the time allocated to the router partition in the static hypervisor configuration. Some inter-partition communications may be excluded from management by the reconfiguration mechanism. In such cases, messages can be forwarded directly between the excluded partitions, bypassing the router.

3) Mitigator: The main purpose of the mitigator is to find replacements for failed application partitions to maintain system functionality. In addition, in case of redundancies, it ensures that flight data output channels are serviced by the most optimal partitions. The functionality of the mitigator is explained in detail in the following section.

IV. Mitigator

A. Mitigator Inputs

The mitigator inputs contain two essential components.

1) *Mitigator Configuration*: This component contains partition mitigation specifications, including a list of partitions to be mitigated along with the plausible mitigations applicable for each partition in the list.
2) *Health Monitor Output*: This component includes a list of health monitor reports of the run-time health estimates of individual partitions as reported by the system's health monitor. Health is estimated in terms of a partition's probability to produce correct outputs. The health monitor aggregates several sources of health estimates and only reports the lowest estimate to the mitigator. Health estimate sources include a partition's adherence to the schedule, presence of heartbeat messages, range checks and anomaly detection on measurements made by the partition, as well as a self-reporting facility. In addition to the lowest estimate, the respective source is reported to the mitigator as well, for more differentiated mitigation strategies.

B. Partition Health States

Below are the different health states that any partition in the system can have:

- *Healthy partition*: A partition is considered healthy if it has a health estimate x such that $0.0 < x \leq 1.0$.
- *Unhealthy partition*: A partition is considered unhealthy if it has a health estimate $x = 0.0$.
- *Unhealthy partition in startup*: Occasionally, a partition may be reported as unhealthy during the startup phase. In such cases, the mitigator applies the same mitigation logic as for a healthy partition. The health monitor reports not only a low health estimate, but also the reason (startup in this case), enabling this special treatment.

C. Run-time Blueprint

The mitigator utilizes a certain data format to access static information about the system during run-time, which we call the Run-time Blueprint. The blueprint provides a comprehensive overview of the system's structure and inter-partition dependencies, detailing essential elements such as:

1) Partition Identification: Each partition within the system is assigned a unique identifier, allowing for referencing and identification during run-time operations.

2) Flight Data Output Channels: The blueprint specifies which output channels are served by each partition and the quality of those outputs. For example, the RADAR partition provides data to the CAS partition with a quality score of 30 (Line 10 in Listing. 1) while Camera_Vision also provides to the same partition but with a lower quality of 25. The quality numbers attached to each output indicate how desirable this output is in comparison to other outputs, with a higher number representing a higher desirability. These numbers are assigned during the system design phase and should be proportional to the expected accuracy and completeness of the provided output data. For example, there could be two partitions providing the same output but with different quality levels due to variations in hardware or software designs. A partition that is not always available but produces more accurate and complete data and a partition that is always available but does not produce data of the same high quality.

3) Input Requirements: The blueprint also specifies each partition's required inputs—for example, RADAR uses "RADAR_input_data".

4) Potential Replacements' Combinations: In the Run-time Blueprint, a list is provided for each partition, outlining all possible combinations of replacement partitions for that partition. These combinations aim to yield the same output as the original partition. To be more precise, if a partition produces outputs X and Y, possible replacements might include another partition that can produce both outputs X and Y, or a combination of two partitions, one producing output X and the other producing output Y. At run-time, the mitigator selects the best replacement combination to recover in-case of a partition's failure to provide for its designated output channel(s). This selection process is detailed in the next section.

```
1  pub const CHANNELS: &[Channel] = &[
2      Channel { name: "RADAR_input_data" },
3      Channel { name: "Camera_input_data" },
4      Channel { name: "Detected_Obstacles" },
5  ];
6  pub const SW_COMPS: &[SWComponent] = &[
7      SWComponent {
8          partition_id: 1, name: "RADAR",
9          required_inputs: &[&CHANNELS[0]],
10         provided_outputs: &[(&CHANNELS[2], 30)],
11         possible_replacements_combinations: &[&[2]],
12     },
13     SWComponent {
14         partition_id: 2, name: "Camera_Vision",
15         required_inputs: &[&CHANNELS[1]],
16         provided_outputs: &[(&CHANNELS[2], 25)],
17         possible_replacements_combinations: &[&[1]],
18     },
19     SWComponent {
20         partition_id: 3, name: "CAS",
21         required_inputs: &[&CHANNELS[2]],
22     ...
23 ];
```

Listing 1. Example Run-time Blueprint for a simple RADAR and camera-based collision avoidance system.

D. Selection of the Best Replacements' Combination for a Failed Partition

The selection of the best combination of replacements in the event of a partition's failure is crucial for maintaining system reliability. The function responsible for this task takes as input a set of combinations, each representing a potential replacements' combination for the partition as specified by the Run-time Blueprint. A combination has a total number of partitions denoted by the symbol T_P.

A possible combination of replacements is considered 'valid' at run-time if it contains partitions that are either healthy or unhealthy but have their inputs provided by healthy partitions. Additionally, valid replacements' combinations cannot contain partitions that have been decommissioned mid flight as will be explained later in this section. The mitigator firstly eliminates invalid combinations and then evaluates the following criteria for each of the remaining valid combinations:

- *Healthy partitions ratio*: The number of healthy partitions in the combination is counted (H_C). Next, the ratio of healthy partitions to the total number of partitions in the combination is calculated. Combinations with a higher ratio of healthy partitions are favored.
- *Restart count of a partition* (rc): If an unhealthy partition's output is needed, the mitigator requests a restart via the hypervisor. A mitigation history tracks restarts per partition over several major frames and is used in run-time scoring—favoring combinations with fewer restarts. Since only unhealthy partitions are restarted, a high restart count reflects poor performance. Additionally, if a partition exceeds a restart threshold within a set time (e.g., the first hour of flight), it is decommissioned and excluded from providing outputs for the rest of the flight.
- *Health inclusive quality of a partition* (hq): This is a product of the run-time health estimate of the partition in a replacement combination and the quality of its output(s) that are shared with the original partition to be replaced.
- *Already active bonus* (ab): This added value provides control over the degree to which active components are favored over inactive ones. This promotes system stability and prevents unnecessary switching between components due to slight performance differences.

Based on these criteria, during run-time, the function computes a score for each combination following (1). The combination with the highest score is selected as the best replacement option.

$$\text{score} = \left(\frac{H_C}{T_P}\right) + \sum_{i=1}^{H_C} hq_i - \sum_{i=1}^{T_P} rc_i + \begin{cases} ab & \text{if partition is active} \\ 0 & \text{if partition is inactive} \end{cases} \tag{1}$$

E. Redundancy Check

The mitigator performs a redundancy check at run-time to determine if a partition is redundant, meaning its output(s) isn't required during the current major frame, as it is already being provided by a better performing partition(s). During the redundancy check, the mitigator uses (1) also to score each partition, not just its potential replacements.

Algorithm 1 illustrates the logic of the redundancy check, which is divided into two key parts:

- Firstly, in Lines 1 - 2, the mitigator checks for each partition if it is better than its replacements' combinations using (1). By treating the partition as its own combination, it undergoes comparison against its possible replacements' combinations to determine if it outperforms them. If the partition proves to be the best performing option among its possible replacements, it is considered non-redundant.
- Secondly, in Lines 3 - 10, if the partition is not the best compared to its replacements, the mitigator checks if there exists a replacements' combination with a higher score that consists solely of healthy partitions. If so, the original partition is considered redundant.

Algorithm 1: Redundancy Check Algorithm

```
   Input  : partition to be checked for redundancy
   Output: Redundancy status of partition
1  if partition is the best provider for its designated output channel(s) then
2      return Non-redundant
3  else
4      replacements combinations ← Get replacements' combinations of
         partition
5      foreach combination in replacements combinations do
6          if All partitions in combination are healthy and combination has a
             higher score than partition score then
7              return Redundant
8          end
9      end
10     return Non-redundant
11 end
```

F. Mitigations

Each partition specified in the mitigator configuration receives only one mitigation during the execution time slot of the mitigator within a major frame of the scheduling cycle. Possible mitigations include:

- *Activate*: If a partition is marked as "Active", this means that the system could potentially use the outputs of this partition during the current major frame as will be explained in Sec. IV-H.
- *Passivate*: A redundant partition is passivated regardless of its health state. This means that the system will not be using the outputs of this partition during the current major frame.
- *Restart*: If the outputs from an unhealthy partition are needed, the mitigator requests the hypervisor to restart that partition. As of right now, this capability is not foreseen in ARINC 653 and is hence hypervisor dependent [2].

G. Mitigator Logic

Algorithm 2 outlines the mitigator logic, which is also illustrated for each application partition by Fig. 4. Below is an overview of its operation:

- *Input and output*: The algorithm takes as input a list of partitions to be mitigated as specified in the mitigator configuration and produces a mitigation decision for each partition in the list as output.

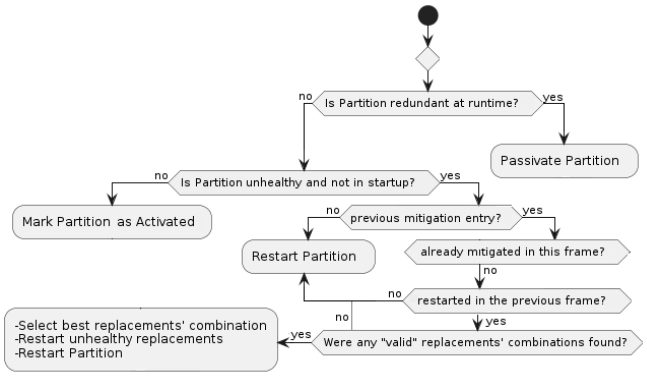

Fig. 4. Mitigation Logic for each application partition.

- *Main loop (Lines 1 - 26)*: The algorithm iterates until all partitions in the input list are mitigated. Within each iteration, it determines a mitigation action on the selected partition based on the partition's redundancy status and health state.
- *Mitigation of redundant partitions (Lines 2 - 3)*: Redundant partitions are passivated regardless of their health states.
- *Mitigation of unhealthy partitions that aren't in startup (Lines 5 - 21)*: For non-redundant partitions that are unhealthy and not in startup, the mitigator evaluates their mitigation history, restarting if necessary, or searches for valid combinations of replacements, selecting the best one based on the process described in Sec. IV-D. Finally, it restarts all unhealthy replacements in the selected combination, as well as the original partition.
- *Mitigation of healthy partitions or partitions that are in startup (Lines 22 - 23)*: Non-redundant partitions that are healthy or unhealthy but are in startup are marked as *Active* where their outputs could potentially be forwarded by the router as inputs to another partition as will be explained in the next section.

Algorithm 2: Mitigator Algorithm

```
   Input  : List of partitions to be mitigated
   Output: Mitigations
1  for i = 1 to (number of partitions in list) do
2      if partition i is "Redundant" at run-time then
3          Passivate partition i
4      else
5          if partition i is unhealthy and not in startup then
6              if partition i has no previous mitigation entry then
7                  Restart partition i
8              else if partition i was already mitigated in this frame then
9                  Continue to next partition
10             else if partition i was not restarted in the previous frame then
11                 Restart partition i
12             else
13                 Look for 'valid' replacements' combinations for partition i
14                 if Valid replacements' combinations are found then
15                     -Select the best combination
16                     -Restart the unhealthy replacements in the combination
17                     -Restart partition i
18                 else
19                     Restart partition i
20                 end
21             end
22         else
23             Mark partition i as "Active";
24         end
25     end
26     i++
27 end
```

H. Mitigator Outputs

Once all partitions in the input configuration receive a mitigation decision, the mitigator is left with a list of *Active* partitions that should provide input data to other partitions. At this point, it is possible that multiple activated partitions output the same type of data, requiring the data to be disambiguated before forwarding it as input to another partition. The mitigator determines the optimal selection of partition outputs based on their quality as specified in the Run-time Blueprint. Therefore, not all outputs of an activated partition are necessarily used as input for other partitions, as another activated partition may produce a similar output of higher quality. The mitigator forwards this selection to the dedicated message router partition (see Sec. III-C2) using an ARINC 653 sampling port.

V. Evaluation

A. System Tests

The system was evaluated based on two primary criteria: communication latencies between partitions and the time required to reconfigure the system in response to faults.

1) Communication Latencies: Communication latencies in the system are primarily influenced by the router, which is responsible for handling all message passing between partitions. The latency between two partitions, say A and B, depends on the scheduling order within a major frame:

- If partition A is scheduled first, followed by the router, and then partition B within the same major frame, the messages from A are routed before B executes. This results in minimal latency—only the run-time of the router is added between A and B. In this scenario, partition B always has access to the most recent messages from A.
- If partition A is scheduled first, followed by B, and then the router is scheduled after that, then B will not receive the messages sent by A until the next major frame. This is because in this scheduling order, the router does not transmit messages until both A and B have already executed within the current major frame.

To mitigate this, the router can be scheduled multiple times within a major frame—for instance, directly after each application partition. Provided the distance of the partitions in the schedule does not increase through the additional time needed for the router, this setup minimizes the latency between consecutive partitions, as messages are routed promptly after they are generated.

2) System Reconfiguration: The second key aspect tested was the system's reconfiguration latency – specifically, the time from fault detection in a partition to the application of a new configuration that mitigates the fault. Three setups shown in Fig. 5 were considered:

1) *Single Router Execution per Major Frame:* In this setup, the router, health monitor, and mitigator are scheduled as the final partitions in the major frame.
   - The router forwards all messages, including any fault-indicating messages, to the health monitor.

- The health monitor detects faults and sends health reports to the mitigator.
- The mitigator generates a new routing table and forwards it to the router.
- In the next major frame, the router applies the updated routing configuration.

This setup results in reconfiguration being applied in the frame following fault detection.

2) *Router Executed Twice per Major Frame:* In this improved setup, the router is scheduled twice—once before and once after the health monitor and mitigator.
   - During its first execution, the router routes fault-indicating messages to the health monitor.
   - The health monitor and mitigator then generate and send an updated configuration.
   - During its second execution, the router can already apply the new configuration within the same major frame.

   This enables faster mitigation of faults and prevents faulty messages from being forwarded to other application partitions.
3) *Router Executed after every Application Partition:* In this setup, the router runs after each application partition.
   - This results in minimal communication latency between application partitions.
   - However, like the first case, faulty messages may be forwarded before the fault is mitigated.

B. Mitigator Standalone Unit Tests

We evaluated the mitigator's impact on system performance by deploying it as a standalone partition and conducting both baseline and scalability unit tests, each running for 1,000 iterations. Since our focus is on identifying general trends—and because there are no hard real-time requirements for reconfiguration time within the scope of this work—strict timing constraints were not enforced. In each iteration, we measured the average mitigation time and the average channel coverage over 30 consecutive standalone runs of the mitigator code.

1) Baseline Experiments:

a) Effect of Redundancy Levels on System Performance: First, to modify redundancy levels, we fixed the number of system partitions at 7 partitions and randomly adjusted flight data channel assignments—determining which flight data each partition consumes or provides. This test was conducted under three scenarios: with no faulty partitions, with half of the partitions faulty, and with all partitions faulty.

Fig. 6 presents the effect of redundancy on the average channel coverage. When half of the partitions are faulty (blue line), increasing redundancy leads to improved average channel coverage. However, some fluctuations (indicated by the region around the blue line) suggest that the distribution of faults across partitions with varying criticality impacts performance.

Fig. 7 illustrates the relationship between redundancy and mitigation time. Across all fault conditions, the average mitigation time generally increases as redundancy levels rise, suggesting that the mitigator requires additional processing time when handling higher redundancy configurations. Additionally, the graph shows a trend where mitigation time increases with fault severity, with the red (all partitions are faulty) line mostly at the top, followed by the blue (half of the partitions are faulty), and lastly the green (none of the partitions are faulty), indicating greater processing time under higher fault conditions. The variations in mitigation time, shown by the shaded regions around each line, arises from uneven fault distribution across partitions, causing delays, especially if critical partitions (partitions servicing multiple channels) fail. Additionally, scheduling and system load on the testing platform can create temporary spikes or drops in the average mitigation time measured during testing.

b) Effect of Fault Levels on System Performance: Next, using a specific configuration of 7 partitions with fixed channel mappings (a fixed redundancy level of approximately 60%), we analyzed how varying the percentage of faulty partitions affected system performance. Fig. 8 shows that for the specified configuration, the average channel coverage generally decreases as the number of faulty partitions increases, as expected. The same figure also shows that for the specified configuration, the mitigation time does not exhibit a strong increasing trend as the percentage of faulty partitions increases. Instead, it remains fairly stable, with only minor variations. As mentioned earlier, the fluctuations in the measurements may result from random variability in fault distribution, critical partition failures, or testing system overhead.

2) Scalability Experiment: In the scalability experiment, we varied the number of partitions to evaluate performance under load. The system starts with a baseline of 7 partitions. Additional backup partitions are gradually added, reaching a total of 20, while maintaining a fixed fault level throughout. Fig. 9 shows that as the number of extra backup partitions increases, the average channel coverage generally improves. However, the same figure also indicates that the average mitigation time increases significantly with the growing number of total partitions. This suggests that while increasing redundancy by adding additional backup partitions improves system resilience, it also introduces additional computational overhead, resulting in longer mitigation times.

VI. Discussion and Conclusions

By decoupling the health monitoring, mitigation decision-making, and message routing, into separate partitions, crashes of the mitigator or health-monitor cannot affect the router. Even if the health-monitor or mitigator permanently cease operation, the router will continue to forward messages based on its most recent configuration. The router performs some general checks on the configuration: Is there exactly one configured source for each used port, is every destination only connected to one port, and do their message sizes match up? If all the checks pass, the configuration is immediately

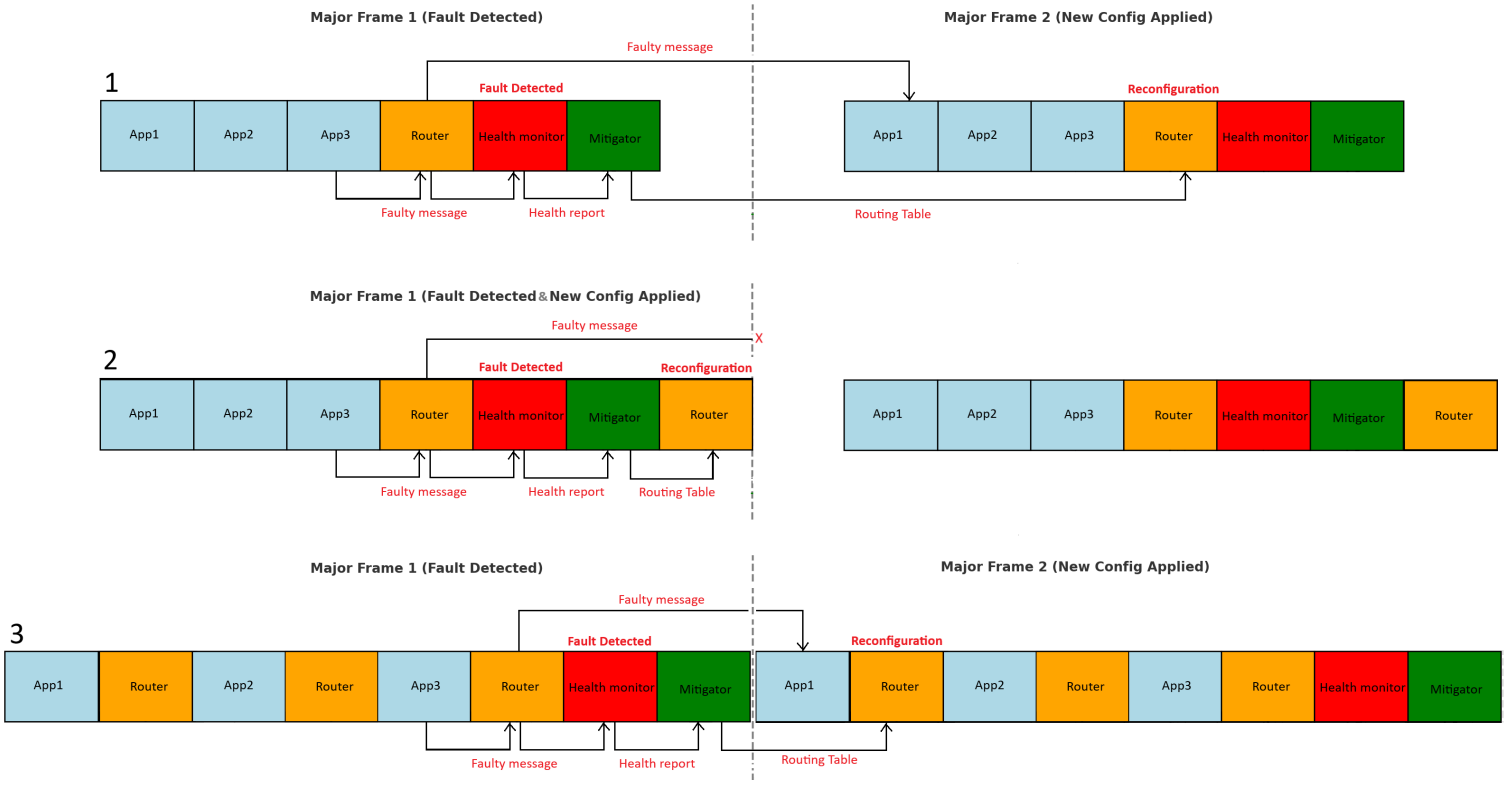

Fig. 5. Three reconfiguration setups.

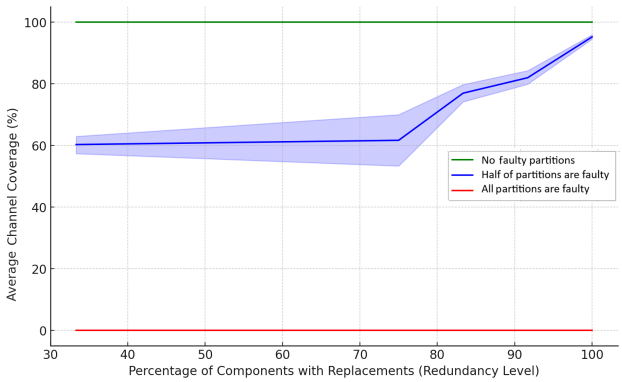

Fig. 6. Average Channel Coverage vs. Redundancy Levels for Different Fault Conditions (No faults – Green; half of the partitions are faulty – Blue; all partitions are faulty – Red); fluctuations are indicated by the regions around each line.

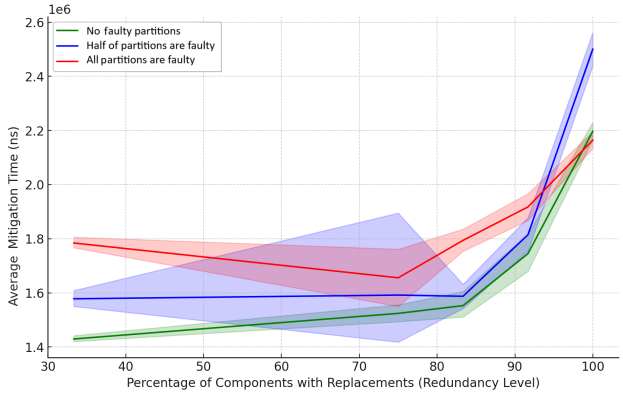

Fig. 7. Average Mitigation Time vs. Redundancy Levels for Different Fault Conditions (No faults – Green; half of the partitions are faulty – Blue; all partitions are faulty – Red); fluctuations are indicated by the regions around each line.

applied, otherwise the last configuration is kept intact. However, some plausible, but faulty, mitigation decisions may still negatively affect the router if the detection of the fault would require knowledge of the system's health. Since the router immediately applies new configurations to minimize the reconfiguration delay, the mitigator has to take care to suppress any fast oscillations between system configurations that might hamper system health. A crash of the health-monitor however may also affect the decision accuracy of the mitigator. The decisions of the mitigator also positively or negatively affect the current health of the system, leading to a change in the health assessment of the health-monitor. The properties of this control loop need to be studied in detail to warrant certification of the system.

Additionaly, reconfiguring the system at the partition level, rather than the system level, presents notable certification challenges. Pre-validating all possible system configurations at design time is not scalable due to the combinatorial explo-

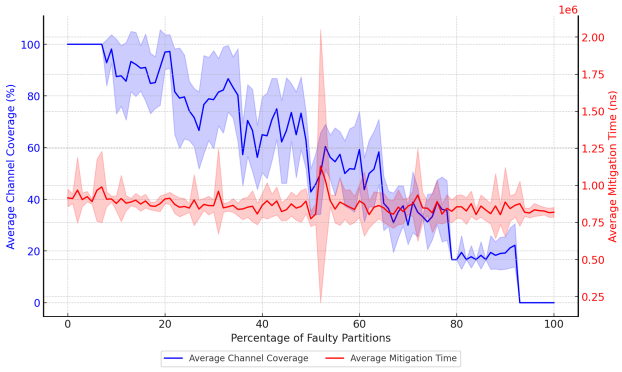

Fig. 8. Percentage Faulty Partitions vs. Average Channel Coverage vs. Average Mitigation Time; fluctuations are indicated by the regions around each line. Note that iterations with the same percentage of faulty partitions may involve different sets of faulty partitions.

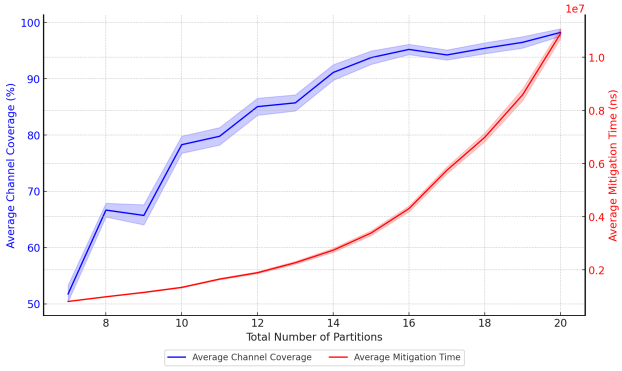

Fig. 9. Increasing Number of Backup Partitions vs. Average Channel Coverage vs. Average Mitigation Time; fluctuations are indicated by the regions around each line.

sion of potential configurations—especially as the redundancy level per partition increases. Even with a modest number of partitions, the cumulative effect of multiple replacements can lead to system-level configurations that fall outside the pre-validated set. Our current evaluation mechanism does not account for this, but it could be enhanced to consider whether a replacement keeps the system within a valid certified configuration. With such logic, replacements that result in uncertified system states would receive low scores, preventing them from ever being applied—ensuring safety, but potentially limiting the system's adaptability in dynamic conditions. In this context, we are also interested in the concept of virtual qualification, as introduced by [4] in their Plug&Fly Avionics Platform (PAFA). This approach performs model-based safety assessments and verification tests at runtime, mimicking the certification processes typically carried out by authorities under ARP4761 [1]. By qualifying new configurations on the fly, virtual qualification eliminates the need for exhaustive design-time validation, potentially offering a scalable and flexible path to safe system reconfiguration.

Timely fault handling is critical for certification. This includes analyzing the worst-case execution time of our reconfiguration approach and ensuring it can reliably meet deadlines under all operating conditions. Examining the real-time properties of our reconfiguration approach is a key direction for future research.

The current work focuses on a single-core setup as a proof of concept. For practical deployment, we plan to adopt a decentralized architecture in which each core hosts its own local instances of the health-monitor, mitigator, and router. In such a setup, application partitions—rather than reconfiguration partitions—could be dynamically migrated between cores upon failure. This allows for scalable recovery mechanisms without tightly coupling system health to any single component or core.

References

[1] Aircraft, S.., Dev, S., Committee, S.A.: Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. SAE International (1996)

[2] ARINC: Avionics Application Software Standard Interface part 1 required services. Standard, Aeronautical Radio Incorporated, Bowie, MD, USA (Dec 2019)

[3] Bieber, P., Noulard, E., Pagetti, C., Planche, T., Vialard, F.: Preliminary design of future reconfigurable IMA platforms. ACM Sigbed Review **6**(3), 1–5 (2009)

[4] Brunner, M., Darwesh, D.N., Annighoefer, B.: A safety process for self-adaptive safety-critical plug&fly avionics. In: 2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC). pp. 1–10 (2021). https://doi.org/10.1109/DASC52595.2021.9594388

[5] Chen, L., Avizienis, A.: N-version programming: A fault-tolerance approach to reliability of software operation. In: Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8). vol. 1, pp. 3–9 (1978)

[6] Durrieu, G., Fohler, G., Gala, G., Girbal, S., Gracia Pérez, D., Noulard, E., Pagetti, C., Pérez, S.: DREAMS about Reconfiguration and Adaptation in Avionics. In: ERTS 2016 - Embedded Real Time Software and Systems. Toulouse, France (Jan 2016)

[7] Engel, C., Jenn, E., Schmitt, P.H., Coutinho, R., Schoofs, T.: Enhanced dispatchability of aircrafts using multi-static configurations. In: ERTS2 2010, Embedded Real Time Software & Systems (2010)

[8] EUROCAE: Software considerations in airborne systems and equipment certifications. Standard, European Organisation for Civil Aviation Equipment, Malakoff, France (Jan 2012)

[9] Lyons, R.E., Vanderkulk, W.: The use of triple-modular redundancy to improve computer reliability. IBM Journal of Research and Development **6**(2), 200–209 (1962). https://doi.org/10.1147/rd.62.0200

[10] Prisaznuk, P.J.: Integrated modular avionics. Proceedings of the IEEE 1992 National Aerospace and Electronics Conference@m_NAECON 1992 pp. 39–45 vol.1 (1992), https://api.semanticscholar.org/CorpusID:267799871

[11] Pérez, J., Crespo, A., Ripoll, I., Masmano, M., Real, J.: Xtratum: A hypervisor for safety-critical embedded systems. Real-Time Systems **47**(6), 589–612 (2011)

[12] Romero, R.F., Saotome, O., Loubach, D.S., Nóbrega, E.G.O., Sander, I., Söderquist, I.: IMA platform computing module based on partial reconfigurable FPGA. In: Proceedings of the Aerospace Technology Congress. Solna, Stockholm, Sweden (October 2016), https://ftfsweden.se/wp-content/uploads/2016/11/FT2016_D10_Rodrigo_Romero_Fullpaper.pdf

[13] Trimberger, S.M.: Field-Programmable Gate Array Technology. Springer Science & Business Media (1994)

[14] Wessman, N.J., Malatesta, F., Andersson, J., Gomez, P., Masmano, M., Nicolau, V., Le Rhun, J., Cabo, G., Bas, F., Lorenzo, R., et al.: Derisc: the first RISC-V space-grade platform for safety-critical systems. In: 2021 IEEE Space Computing Conference (SCC). pp. 17–26. IEEE (2021)

[15] Zaeske, W., Brust, C.A., Lund, A., Durak, U.: Towards enabling level 3a ai in avionic platforms (2023), https://dl.gi.de/handle/20.500.12116/40203