Towards Enabling Level 3A AI in Avionic Platforms

Resilience through Dynamic Reconfiguration

Wanja Zaeske[1], Clemens-Alexander Brust[2], Andreas Lund[3], Umut Durak[4]

Abstract: The role of AI evolves from human assistance over human/machine collaboration towards fully autonomous systems. As the push towards more autonomy subsequently removes the reliance on a human overseeing the system, means of self supervision must be provided to enable safe operations. This work explores dynamic reconfiguration to provide resilience to unforeseen environmental conditions that exceed the systems capabilities, but also against normal faults. We focus on providing the means for this in an ARINC 653 compliant environment, since we target avionics platforms. Scheduling and communication are two major aspects of dynamic reconfiguration. Hence, we discuss multiple respective implementation approaches. The third pillar of reconfiguration, the process of deciding *when* to reconfigure is also investigated. Combining these yields the building blocks for a self-supervising system.

Keywords: Avionics; Resilience; ARINC 653; DO-178C; Fault-Tolerance

1 Introduction

Software development in avionics is driven by assurance. The first two questions to arise when innovation is about to happen read: *What level of assurance is required for this functionality?* and *(How) can we prove that it is safe?*. The likelihood of a technological advancement in avionics is directly conditioned to the practicability of its assurance. Keeping this in mind, Artificial Intelligence (AI) is very interesting: it promises advancement in many fields of study, but is particular hard to assure. EASA enumerates three levels of AI which again are subdivided: Level 1A, 1B, 2, 3A & 3B [EA21]. Briefly put, the distinction between Level 2 and Level 3A lays in the supervisory role. For Level 2, the end-user's intervention shall always be possible, reserving the supervisory role to a human. Beginning from level 3A, at least some ability to override an AI-based system is retained from human

[1] Deutsches Zentrum für Luft- und Raumfahrt, Flugsystemtechnik, Abteilung Sichere Systeme und Systems Engineering, Lilienthalplatz 7, 38104 Braunschweig, Deutschland wanja.zaeske@dlr.de

[2] Deutsches Zentrum für Luft- und Raumfahrt, Institut für Datenwissenschaften, Datengewinnung und -mobilisierung, Mälzerstr. 3-5, 07745 Jena, Deutschland clemens-alexander.brust@dlr.de

[3] Deutsches Zentrum für Luft- und Raumfahrt, Institute for Software Technology, Software for Space Systems and Interactive Visualization, On-board Software Systems, Münchener Str. 20, 82234 Weßling, Deutschland andreas.lund@dlr.de

[4] Deutsches Zentrum für Luft- und Raumfahrt, Flugsystemtechnik, Abteilung Sichere Systeme und Systems Engineering, Lilienthalplatz 7, 38104 Braunschweig, Deutschland umut.durak@dlr.de

doi:10.18420/se2023-ws-18

end-users, i.e. human intervention is limited to those cases where human judgement is required to safeguard the operation [EA21]. Today, once a fault grows to become a system failure or even a failure condition *after* redundancy mechanisms failed, the mitigation of the problem is up to the human. When pursuing the goal of Level 3A/B autonomy, naturally the question arises: *What would it look like, when the machine supervises itself?*. The effect of (partial) self-supervision is an increased system resilience, without further reliance on humans. Thus, in this work we exploit dynamic reconfiguration paired with learning-based mitigation strategies to increase a system's resilience, even if some of the system's components comprise hard to verify AI-based functionalities.

The most prominent standard for software development in airborne systems is DO-178C. In it, the usage of partitioning to separate applications running on the same hardware from one another is demanded [RT11]. The promise is, that errors local to one application are contained inside the application's partition [Co95]. In the ARINC 653 standard Application Executive (APEX) is described, an execution environment for partitioning. It specifies aspects like scheduling, spatial isolation, Application Programming Interface (API) for the software contained inside a partition, and means of inter- and intra-partition communication [AR19a, AR19b].

We focus on the dynamic reconfiguration of partitions built for APEX. The reasoning behind this is threefold. First, failure in partially AI-based components has to be mitigated in software as well, as a large share of AI happens in algorithms. Secondly, when considering the different layers of software, the partition level sticks out for being already designed to form classical software items that provide fault containment [AR19a, Co95]. Last, restricting reconfiguration to software enables vast degrees of freedom while keeping the scope focused on one domain.

The structure of this paper unfolds as follows. First in section 2, we introduce the relevant bits and pieces of ARINC 653, dynamic reconfiguration and EASA's ambitions towards AI in avionics. Then, a view on the related work about dynamic reconfiguration of avionic software is provided in section 3. Next, section 4 elaborates on the various means of implementing reconfiguration, considering both their advantages and caveats. Furthermore, aspects regarding fault detection and selection of proper mitigation strategies are detailed. Finally, we summarize our findings in section 5, also pointing out the next steps to be done.

2 Background

2.1 Dynamic Reconfiguration of Resilient Avionics

The ground laying assumption for this work is as follows: advanced methods such as AI can — under certain environmental conditions — provide functionality that is not achievable to the same degree with traditional methods [EA21]. This especially seems true when considering autonomy functions. However, if these conditions do not hold, the function may be impaired or become unavailable. Unless the loss is compensated for, it is not allowable to depend upon said function. The goal is therefore to benefit from these functionalities when provided, but not to fail once their availability is endangered. A reconfiguration to another operation profile or an alternative implementation of a similar function at runtime enables that.

A concrete example is detection of intruder aircraft, as it is necessary for a Collision Avoidance System (CAS). Traditionally this function is implemented using transponders. Aircraft are equipped with transponders that both broadcast their current position and receives the position of other near aircraft. The information is then used to avoid mid-air aircraft collision. An AI approach for gathering the required information could be based on Computer Vision (CV); other nearby planes are detected using cameras and advanced algorithms. The CV approach provides enhanced functionality: it enables even the detection of aircraft that either are not equipped with a transponder or whose transponder is defective. From an autonomy perspective this is a powerful enhancement, with the CV application less cooperation between aircraft is necessary to avoid collisions. On the other hand, the CV system is sensitive to environmental conditions. Most likely it requires VFR conditions, thus its performance is optimal only during daytime with a clear sky. The sum of these constraints describing the operational domain for which the CV application is designed are also called Operational Design Domain (ODD) [KF19]. To counter failure once the ODD is left, two independent CV applications could be used, one tailored for daylight using normal cameras and a second one adapted for night flight, equipped with IR cameras. With the nighttime CV having a separate ODD, the combination of both systems can cover a broader set of environmental conditions. But even the union of both ODDs does not cover all relevant cases. After all, the environment is not necessarily passive. For example, CV is susceptible to adversarial attacks, where a specifically crafted object that may appear harmless to the human eye results in wildly incorrect results. To sum up the challenges: The best provider (daylight CV vs. nighttime CV vs. transponder) for information on intruder aircraft has to be selected in order for CAS to function. This decision affects the operational constraints of the system, thus environment and system performance should be monitored as well to further guide the decision. Monitoring is also required to detect sudden degradation of the function, e.g. when equipment fails or when an adversary carries out an attack. Complete verification of the CV is out of scope due to the countless failure scenarios, thus instead a resilience layer has to be added around it. When implementing that as dynamic reconfiguration, safety guarantees need to be upheld for the various configurations but also while transitioning from one to another configuration. The resulting flow is depicted in Figure 1: the applications comprising the platform emit data, for example telemetry, which is monitored. If indications of failure or serious performance degradation becomes evident to the monitor, a supervisory component is informed about the issues observed. Then the supervisory component chooses one among multiple mitigation strategies to resolve the issue, i.e. replacing the malfunctioning application. The process is a cycle, thus the success or failure of previous mitigation attempt is observable.

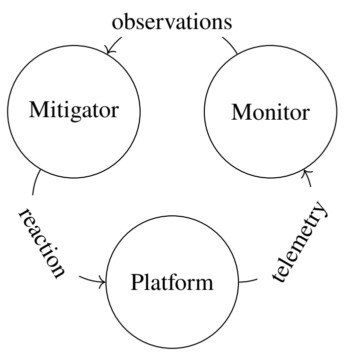

Fig. 1: Resilience Cycle. The platform's behavior is monitored for indications of failure and degradation. If issues are detected, a decision for a suitable mitigation strategy is made. The mitigation strategy is applied to the platform, success in addressing the issue can be verified by the monitor

Remarkable is the similarity between the described resilience cycle and the decision-making process of a human supervisor like a pilot. The system is monitored for anomalies, which may indicate failure. If such an indication of (future) failure is present, actions are performed to mitigate the effects. A specialty of human supervisors is their capability to learn: when a mitigation strategy does not seem to work, often human supervisors *try something different* even if logic strongly suggests the first mitigation strategy to be most suitable. This is elaborated further in subsection 4.3.

Now in order to make the system more resilient, dynamic reconfiguration can be utilized, as depicted in Figure 1: the system and its environment is monitored, both for anomalies and for the environmental conditions that are deemed necessary for AI components to function properly. Based on the observations of the monitor the decision is made to reconfigure the system to either utilize a traditional or an AI functionality. As the reconfiguration happens during runtime, one might speak of dynamic reconfiguration. When combined with traditional health monitoring, the overall system resilience can be increased further, by mitigating the faults detected through the monitor.

2.2 Integrated Modular Avionics & ARINC 653

Integrated Modular Avionics (IMA) is the established architectural framework for modern avionics. The design philosophy of IMA circles around the concept of "flexible, reusable and interoperable hardware and software resources" [RT05]. This manifests in the consolidation of multiple applications on one hardware module. Multiple of these modules are installed together in a cabinet, and multiple cabinets are distributed through the aircraft. As one module is expected to host many independent applications, appropriate isolation mechanisms have to ensure that applications do not interfere with each other. This non-interference is referred to as partitioning in DO-178C [RT11]. To achieve the aforementioned goal of interoperable hardware and software, the behavior and software interfaces to the module should follow a common design philosophy.

APEX, the Application Executive, is just that. It is a common execution environment for software in avionics, implementing up on the partitioning mandated by DO-178C. The API is real-time friendly, extensible, independent of programming languages and processor architectures [AR19a, Co95]. The reasoning for the creation of APEX' is driven by goals closely related to the IMA philosophy. By reducing language and hardware dependence for the execution environment, porting to other platforms becomes easier. In addition to that, reusability and modularity is improved by declaring a common abstraction layer — APEX. Further, the integration of functionality related to partitioning into APEX supports the use-case of mixed criticality applications. In fact, DO-178C explicitly allows two partitions running on the same hardware module to be assigned different assurance level [RT11]. The software that runs on a module while managing the partitions is also referred to as a hypervisor or separation kernel.

Inter-partition communication in APEX is conducted using two types of ports: sampling, and queuing ports. Sampling ports retain the last message written to them. Only one partition can write to a sampling port, but multiple partitions can read from the same sampling port. Queuing ports instead form a single producer, single consumer fixed-depth queue: up to n messages can be enqueued. When the destination partition reads a message from the queue, it is consumed, freeing one slot in the queue. The source partition can only insert a new message when there are less than n messages already enqueued. Both the size of each message and the maximum number of messages in a queuing port n are design time parameters. [AR19b]

The scheduling of partitions itself is static and deterministic, following a major-minor timeframe model. Figure 2 illustrates the elements comprising a schedule: Partitions can be assigned multiple time windows (or minor frames), each consisting of a continuous interval of time and an offset to t_0 of the given major frame. During a time window, the respective partition is executed on the core of the time window, otherwise the partition's execution is paused. The major frame as well has a finite duration, in which multiple time windows can occur. During execution, duration and offset of the time windows determine how long and when a partition can run on the CPU. The major frame is repeated infinitely, resulting in a static schedule with completely deterministic CPU quotas for each partition per major frame [AR19b]. An interesting addition is the possibility of changing between multiple major frames at runtime, described in ARINC 653 P2 [AR19c]. The possibilities opened through dynamically changing the current schedule is further elaborated in subsection 4.1.

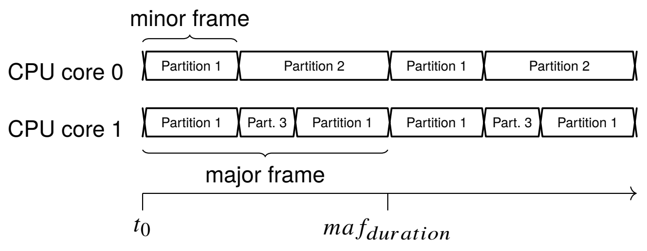

Fig. 2: Visualization of the major-minor timeframe scheduling concept. The major frame, repeated infinitely, describes the complete schedule. Minor frames are time windows for the execution of individual partitions.

3 Related Works

The idea of (dynamic) reconfiguration in APEX is as old as APEX itself, in fact it was already formulated almost one year *before* the initial publication of ARINC 653 [Co95]. It follows that there has been a steady flow of projects investigating dynamic reconfiguration (of avionics) over the past decades.

At almost the same time Seeling reported in [Se96] on the fault-tolerant architecture of the integrated avionics platform for the F-22 air vehicle. The described architecture foresees three different level of reconfiguration: **minor, major, degradation**. In a **minor** reconfiguration the system will replace a failed component by an equivalent on the same hardware. A **major** reconfiguration involves the migration of functionalities to other hardware components. In **degradation** the system will switch off low priority components. The actual reconfiguration is then implemented by using mapping tables and a state automaton for each component.

[Ló12] elaborates the limitations of ARINC 653 regarding dynamic reconfiguration for the sake of fault tolerance. It promotes the decoupling of partitions by routing inter-partition communication through a mediator (residing in its own partition). Using a real-time routing table, the mediator implements the logical information flow between partitions as required for the current configuration. The proposed solutions are tailored to be compatible with an ARINC 653-based IMA architecture. Furthermore, the idea of a dynamic trust level per partition is presented. If a redundant partition's output deviates from the voted consensus, its trust level is reduced. In subsequent votes, only those partitions with a trust level above a certain threshold are considered. Partitions can however also increase their trust level if they again produce results equal to the voted consensus. A notable limitation of the described solution is that reallocation of partitions is only considered for stateless partitions. It is clear how this limitation came to live: migrating state when an application is migrated to another module due to failure is a tricky problem. Many applications however require state, excluding them from the aforementioned implementation. [Ló12]

[Du16] describes a middleware geared towards addressing failure through reconfiguration. The assumed failure model recognizes two types of faults: **permanent core failure** & **temporal overload**. Permanent failures are mitigated through re-allocating functions to different hardware, which is called *reconfiguration*. The possible allocations are pre-computed at design time. Temporal overload is addressed by re-assigning time from *best-effort applications* to *critical applications*. This so-called *adaptation* allows assigning less than the worst case execution time (WCET) to *critical applications*. [Du16]

In the context of the **SCAlable & ReconfigurabLe Electronics plaTforms and Tools project (SCARLETT)** Bieber et al. presented in [Bi09] an approach for reconfiguring avionic applications among clusters of modules by means of pre-calculated, certifiable directed acyclic configuration graphs. The proposed approach relies on spare modules and partitions. This work, strongly focussed on control law, reports on the simulation-based testing of a pitch control application. Unfortunately the topic of (dynamic) reconfiguration is not deepened in the work, and further literature about the SCARLETT project is sparse.

In contrast to aforementioned work the authors of [En10] developed a decentralized reconfiguration approach for integrated modular avionic systems. The *Distributed, equipment Independent environment for Advanced avioNics Applications (DIANA)* project also provides an off-line, pre-computed set of configurations, i.e., a set of mappings between applications and modules. Each and every module executes the necessary applications for health monitoring and reconfiguration. In case of a failing module, the remaining modules will utilize a Byzantine agreement protocol to determine the transition to the next configuration in order to keep the aircraft dispatchable.

The work described in [Se96] describes a hierarchical approach to mitigation. The work focuses on rather dated technology, but introduces the idea of associating various degrees of failure with reconfiguration to varying degree, a concept that we also embrace. The [Ló12] lays important groundwork for reconfiguration of APEX-based avionics, while putting a stronger focus on software. Noteworthy is the idea of the partitions trust-level, which we pick up with the idea of a learning algorithm for the decision-making. [Du16] investigates both the migration of applications away from failed cores (like [Ló12] as well), while also enhancing the normally static schedule of the XNG hypervisor with a dynamic component to catch occasional run-offs when a safety critical application did not complete within its time window. We believe this to be dangerously close to dynamic preemptive scheduling which might break the temporal isolation, hence we refrained from further considering this mechanism. As we were unable to find comprehensive literature about software centric reconfiguration from the *SCARLETT* project, no significant inspirations from it where considered. The literature about the DIANA project is geared more towards reconfiguration on the ground to keep aircraft dispatchable, but it introduces a very interesting concept: the utilization of Byzantine agreement protocols is something we most definitely like to integrate into our implementation of dynamic reconfiguration in order to maintain system coherency across multiple modules.

4 Key Aspects of Dynamic Reconfiguration

To achieve dynamic reconfiguration in a partitioned environment, three problems need to be solved. To begin with, reconfiguration involves changing the set of applications which are executed. The information, when each partition is to be executed is defined in the hypervisor's schedule(s). Therefore, the first problem is detailed further in subsection 4.1. Further on, APEX foresees a tightly defined set of communication channels, which are predefined during design time [Co95, AR19b]. As the flow of information is also affected by reconfiguration, the second problem lays in the implementation of configuration dependent communication channels that adapt during runtime. This topic is outlined further in subsection 4.2. Last is the question *when* to reconfigure. An approach to this third problem is described in subsection 4.3.

Our basic understanding of a software system in IMA is as follows: various functions are implemented in applications, which in term are allocated to multiple partitions. These partitions communicate via inter-partition communication channels. Dependencies are described as consumer/producer relationship between a partition and specific data. Some functions are redundant, as in multiple dissimilar applications can host equivalent functions. If multiple providers of some data are available, the system configuration determines which provider's data are to be consumed by the partitions dependent on that data.

4.1 Reconfiguration of Scheduling

In APEX, dependability is obtained thanks to the available knowledge of many module properties at design-time. Consequently, the schedule of an APEX module is static at least at the partition level and cannot be mutated during runtime. APEX P2 however allows for multiple static schedules to be defined at designi-time combined with the possibility of specially privileged system partitions to select the current schedule in use [AR19c]. Within the framework provided by APEX, this leaves two options to approach change of schedule, but not changing the schedule is an option worth considering as well.

Option 1: Changing between pre-computed schedules. A reconfiguration will be conducted by switching to another a priori computed schedule of the hypervisor (see figure 3), which then includes a different set of partitions, thus also different applications.

Figure 3 illustrates an example, where effectively partition 3 (green) is replaced by partition 5 (magenta). This is achieved by having two almost identical schedules, where only the application scheduled third in the major frame differs. This option lends itself, since it relies on using services provided by APEX P2. Since all schedules have to be pre-computed during designtime, all of them can be validated at designtime as well. This avoids the complex problem of finding a valid schedule in bounded time and space during runtime. The main issue with this approach is the combinatorial explosion of pre-generating all required schedules. Let's consider the example depicted in Figure 3 with the additional constraint

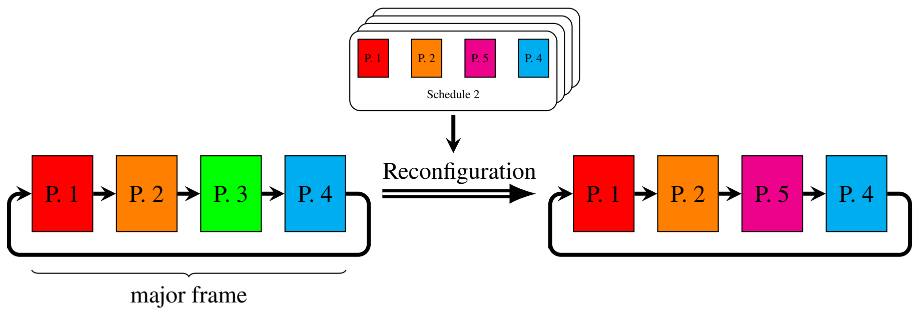

Fig. 3: Reconfiguration by changing schedules. Each application is implemented in its own partition. In case of a necessary reconfiguration, the schedule of the hypervisor will be changed. All possible schedules need to be pre-computed at designtime.

that the switch from partition 3 to partition 5 has to happen without downtime. Furthermore, we assume for this example that the reason for the reconfiguration is not a failure, but simply that partition 5 is expected to perform better under the current conditions. Partitions in APEX require a warm-up, therefore at least during the first partition window of partition 5 only initialization routines are executed. For a brief period both partition 3 and partition 5 should be executed while the outputs of partition 3 are still used until when partition 5 transitions from warm-up to normal mode. Providing this graceful handover provides for a seamless operation of the system, but comes at a cost: The total number of schedules required to enable all reconfiguration scenarios then would be the cross product between all partitions that are alternatives to each other combined with intermediate schedules required to enable graceful handover during warm-up as described in the example. This creates yet another problem. Most likely different schedules have different major frame times. Keeping a system composed of multiple modules (and therefore multiple hypervisors) with major frame durations that change during runtime synchronous is complex. Of course this can be done by stretching the length of shorter major frames to match that of the schedule with the longest major frame duration, but that creates the necessity to over-provision the system since, due to gaps in some schedule. Finally, some APEX implementations might impose significant limitations towards the total number of schedules in a module, for example fentISS' LithOS[5] revision `r7919` allows only for up to 20 different schedules.

Option 2: Switching applications inside of partitions. Provide hypervisor with only one kind of partition, a generic partition which includes all applications of the system. In case of a reconfiguration the partitions will be instructed to change their active application, such that a different set of software components becomes active. The mechanics of this are depicted in Figure 4: because of a reconfiguration, the generic partition which is third in

[5] `https://fentiss.com/products/lithos/`

the schedule terminates application 3 (green), instead executing application 5 (magenta). This corresponds to the approach described in [Du16], where the so-called Local Resource Schedulers (LRS) is executed first in each partition time window, launching the various tasks to be finished in that time window.

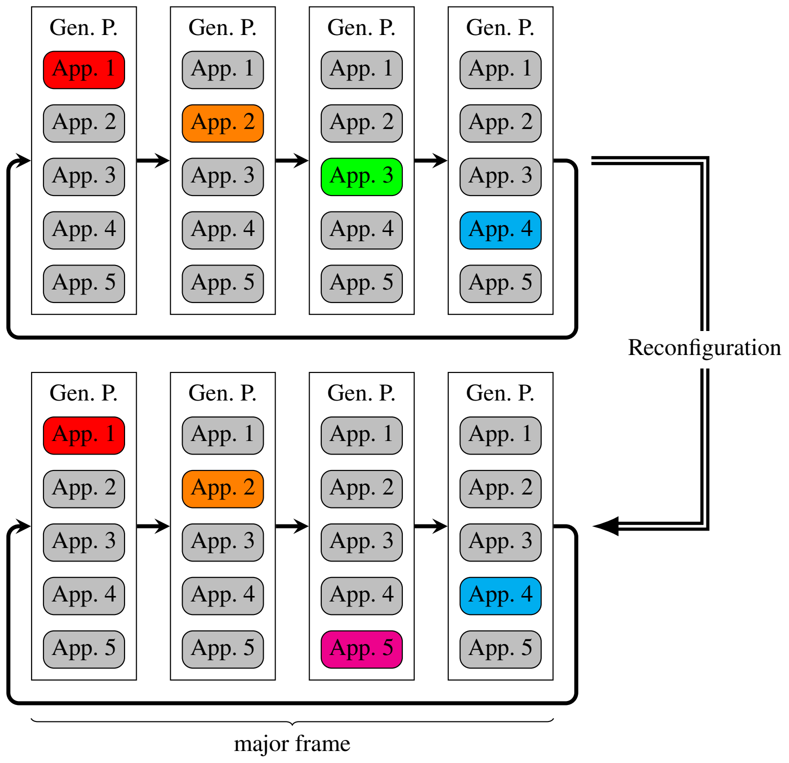

Fig. 4: Reconfiguration by branching in the partitions. All partitions include the code for each and every application, but execute only one at a time. In case of a reconfiguration, the partitions will be instructed to change the active application.

The benefits of this approach are its flexibility and responsiveness. While a change of the module schedule only comes into effect after the current major frame is finished, reconfiguration inside a partition can come into effect as soon as said partition is executed. This not only enables reactions with small delay irrespective of the major frame duration, but it also grants significant freedom regarding what code to execute when, as the scheduling of processes inside a partition is dynamic. One significant drawback is that this requires all different applications to be compiled into one partition image. From that follows as well that a partition voluntarily has to apply a pending reconfiguration, and conversely that a malicious partition is problematic. The other drawbacks are related to the designtime driven, static approach in APEX. A partition cannot change the duration of its partition windows: sufficient time has to be assigned to the partition to accommodate the worst case execution of its slowest configuration, again resulting in over-provisioning. In a sense, this approach relies on gaining more degrees of freedom by breaking the static assignment of responsibilities to partitions, which, we think, is not in spirit with the temporal and spatial isolation desired when using partitioning. DO-178C mandates the software assurance level to be determined per partition [RT11]. If multiple applications can be executed from one partition, all of them would be assigned the assurance level of the most critical application among them. It remains to be seen whether certification authorities would be willing to allow this, and if so under which conditions. There is the argument that a suitable initializer might be able to guarantee the execution of the correct application within a partition.

Option 3: Always execute all partitions. The most common approach to this issue is not to reconfigure the schedule at all, essentially ignoring the issue of changing what code is executed altogether. Instead, multiple partitions providing a function may be executed at the same time, with one being in active use while the other serves as a hot-spare. This approach simplifies the configuration (no multiple schedules are required) while not relying on the cooperative behavior of all partitions. Furthermore, it allows keeping most reconfiguration implementation details out of the normal partitions. In addition, this allows to monitor the output of the hot-spares when they are not active while omitting problems for a graceful handover. The drawback of course is that this again requires significant over-provisioning, as most hot-spare approaches do.

Ultimately the choice between these option boils down at which level the reconfiguration is to be implemented: inside or on top of the partitions. Fundamentally, over-provisioning is inherent to this design. However, maybe not all functions are computationally expensive. Thus, one could combine Option 3 (hot-spares) with Option 1 (changing between pre-computed schedules), where only the most demanding functions are mutually exclusive (requiring reconfiguration through a change of schedule), while all others are executed as hot-spares irrespective of whether their outputs are actually used or only monitored.

4.2 Reconfiguration of Communication

Whenever reconfiguration is implemented through more than just changing what code is executed inside each partition, also the communication channels change. But in APEX not only the scheduling is static, also the communication is configurable at designtime only; for example which partition may write to or read from a sampling port. Furthermore, APEX does not foresee communication channels that have multi-producer semantics, both sampling ports and queuing ports are single-producer only. Let's assume a setup in which one or another partition has to provide some data, depending on the current configuration. Now, after a reconfiguration it may be necessary to change which of the two partition's output is to be used. As already hinted, APEX does neither allow for on port to be writable from more than one partition nor does it allow redefining a port's source partition at runtime. But how can the communication change then, during runtime?

Option 1: Selection in the receiving partitions. Each partition providing some data has its own port to write to. A consuming partition has destination ports for all possible producers. Depending on the current configuration, the consumer partition selects from which of the multiple producer ports it consumes data. An example of this change is highlighted in Figure 5: at first partition 4 (blue) uses data provided from partition 2 (orange), but after the reconfiguration the data provided from partition 3 (green) is used instead. The benefits of this approach lay in its simplicity: As each partition decides what inputs to use own its own, no single point of failure is required. The scheduling is simple as well, the only constraint is that all producers of some specific data are executed before the first consumer of said data. Further on, this enables local error management; for example when one producer of data fails to provide new data, the consuming partition can switch to another producer ad-hoc to mitigate the error. This also implicates that the reconfiguration of communication can be instantaneous: once the condition determining the best provider for some data changes, all consumers of that data can just start using another provider.

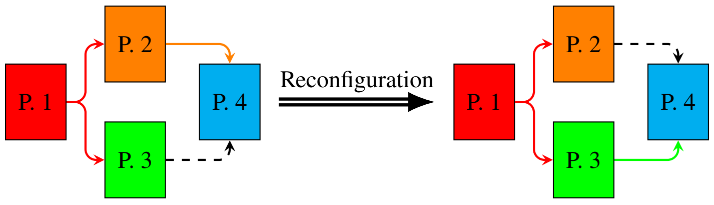

Fig. 5: Reconfiguration of channels inside the partitions. In order to reconfigure, a partition internally chooses a different channel as information source. Application 5 is effectively passivated before the reconfiguration, since no other partition actually considers its messages.

However, local error handling also poses a problem, as it hinders concise, coherent monitoring and mitigation. It certainly is easier to delegate error handling of non-trivial problems to a central entity, a fact that is reflected in ARINC 653's health monitoring. There is also friction from an architectural standpoint: this approach requires reconfiguration dependent code to be mixed into normal partitions. After all it is code residing in a partition that decides what producer's data to use. This also requires trustworthy partitions for monitoring, as only inside a partition one can really be sure which provider was picked if multiple were available. As reconfiguration can now affect the flow of information at any given time, special care has to be taken to avoid dirty reads: whenever the condition determining the best provider for some data changes, all partitions executed until then used a different value for said data then the partitions yet to be executed in the current major frame. In addition to that, it this option requires excessive amounts of ports to be allocated in the hypervisor configuration. This can both lead to high resource utilization (specifically RAM), but there also might be a numeric limit to how many ports a hypervisor can support.

Option 2: Port routing in a dedicated partition. Instead of direct communication, data providers submit their data to a routing partition, similar to the concept of a mediator introduced in [Ló12]. This routing partition then based on the current configuration forwards the data to all consumers and the monitor. Figure 6 illustrates the flow of information for this case: both partition 2 (orange) and partition 3 (green) provide data, one of which is to be preferred depending on the configuration. While the data provided by partition 2 is used prior to the reconfiguration, the router opts to forwarding partition 3's data instead after reconfiguring.

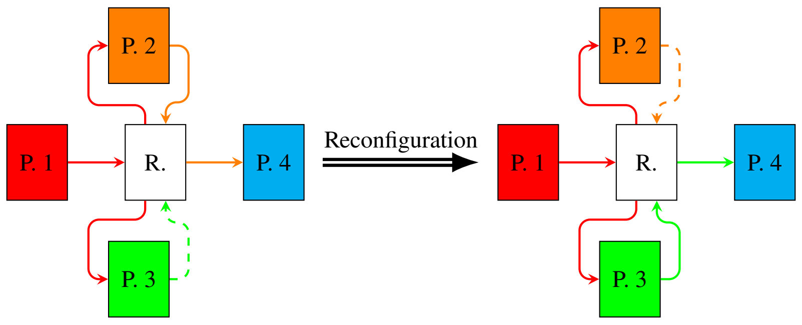

Fig. 6: Reconfiguration of channels with a routing partition. In order to rewire the communication, a special partition is introduced to route messages between the partitions/applications.

Doing so enables a significant simplification: none of the normal partitions needs to know anything about reconfiguration. The only partition that needs to know the current configuration is the router. Furthermore, since all communication is observable to the routing partition, the monitor can easily be tapped into all inter-partition communication. For the case of queuing ports it even becomes visible whether a partition failed to consume from a queuing port (as the routing algorithm notices the queuing port to be full) or if a partition failed to provide outputs (if the queuing port remains empty over multiple partition windows). Additional, the routing partition allows for transparent inter-hypervisor communication: when multiple routing partitions, each residing on a separate hypervisor, are connected, data can be routed between partitions on different hypervisors. This also enables the live migration of one application to a different hypervisor.

Still there are flaws with this; foremost regarding timing. End-to-end latencies increase, as the router has to be schedule in between producer and consumer partition as well. Moreover, the addition of a routing partition creates computational overhead that needs to be accounted for. Also, ARINC 653 P4 mandates "only one partition window within the partition's period" [AR12], further constraining the scheduling of the routing partition. Another issue with this approach is the reliance on one partition for the whole system to work, a fault in the routing partition would have devastating effects for all other partitions on that hypervisor. Therefore, the routing partition has to be developed to the highest assurance level hosted on the hypervisor. Of course the routing partition could be made redundant, but at the cost of further overhead and complexity.

4.3 Decision-making

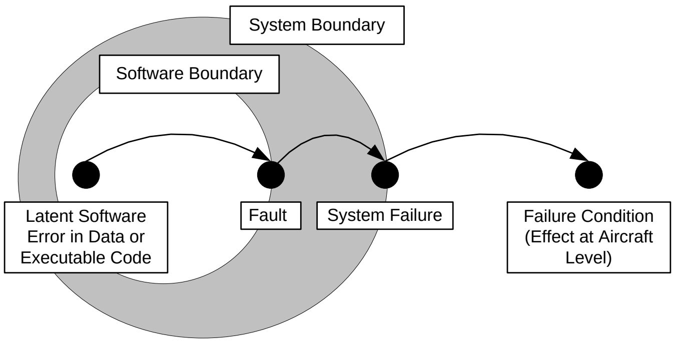
System Boundary
Software Boundary
Latent Software Error in Data or Executable Code
Fault
System Failure
Failure Condition (Effect at Aircraft Level)

Fig. 7: Chain of events leading up to a failure condition. **Extracted from RTCA DO-178C, ©RTCA/EUROCAE ED-12C, ©EUROCAE. Used with permission. All rights reserved.**

The common fault model assumed in DO-178C (illustrated in Figure 7) anticipates multiple stages for faults [RT11]. Initially faults originate from latent errors. When considering advanced AI-based applications, unforeseen inputs can cause erroneous behavior as well, and this is likely to happen due to enormous input spaces that often can't be verified fully. Once an error occurs, it might cause faults, which when unmitigated can cause system failure. As the software boundary is traversed, mitigating becomes harder or even impossible, depending on the faults immediate effect to non-software systems. For example, a software error that caused a fault manifesting in physical overloading of structures can not be solved in software after the fault occurred.

One of the goals for the reconfiguration decision is to catch faults before they reach the system boundary. Further on, in the case of AI-based functionality measures must be taken to keep the applications in question well within their respective ODDs as long as they are actively used. But errors and subsequent faults are not the only reasons to reconfigure. If the performance or the safety margin can be improved, reconfiguration is also justified. To approach the problem of deciding if and how to reconfigure, we divided it into three aspects. First we describe the fault model of our ARINC 653-based software stack. Secondly, we explore indicators of a component's health to aid our decision-making. Last, our considerations regarding the selection of an appropriate reconfiguration suitable to mitigate observed problems is illustrated.

4.3.1 Fault Model

ARINC 653 already foresees to encapsulate the various applications composing the whole system into partitions [Co95, Co95]. The partitions provide robust fault isolation and containment, despite them being executed on the same hardware module. However, a fault in one partition can still directly (e.g. through inter-partition channels) or indirectly (over the environment) affect other partitions and even the whole system.

The causes for faults are manifold. For example, latent software/data errors or incomplete requirements contribute towards the design-time errors inherent to software. It is important to remark that these may be masked for prolonged time-spans, potentially even longer than the projected life-time of an application. Despite that, given the specific chain of events that uncovers a latent error it may still cause a fault and a successive failure.

Not all errors are caused by defects though, soft errors — caused by radioactive decay or cosmic rays — can alter data or program code in the hardware module. Regardless of the error's nature, the disturbance caused can be either that the application won't deliver its functionality at all, or it will deliver faulty results. A system which relies on the applications functionality and doesn't have a mitigation for such a failure will suffer a total system failure. These errors do however not necessarily render an application useless; an error unmasked only when a timer overflows for example may be reliably mitigated with partition restarts.

In order for the mitigation to be focused, a unit of fault needs to be declared. For our proposed software stack we consider the partition, to be the unit of fault. This abstraction simplifies the formulation of mitigation strategies: first the set of partitions affected by a fault are selected. Then, considering the available information, one or multiple mitigations preserving the functions provided by affected partitions are executed. Having the partitions as fault containment units aligns well with the degrees of freedom granted by ARINC 653 environment [Co95].

4.3.2 Fault Detection

There are several health indicators of a partition or application. While self-diagnosis and system-level exception handling play an important role, they are not reliable on their own. In particular when a partition delivers faulty results, the exception handling will remain passive while the self-diagnosis might be impaired by the same reason the results were faulty to begin with. Both assume a correct and trustworthy implementation. Hence, we propose to also monitor applications externally to detect a wide range of faults. We consider the following monitoring aspects to determine a need for reconfiguration:

- **Timing**. Continuous monitoring cannot only detect clear timing violations, but also instances of barely meeting the deadline or improbably fast execution.

- **Data Plausibility**. Analysis of all inter-partition communication can reveal implausible messages, e.g. by comparison between alternative sources of identical information and range checking.
- **Anomaly Detection**. All inter-partition communication and diagnostic information on a partition's internal state is analyzed by machine-learning-based anomaly detection to identify unusual behavior and unexpected types of faults.
- **Operational Constraints**. Applications may require certain environmental or system conditions to perform adequately. For example, AI-based applications can be limited to one or multiple ODDs. Thus, the conditions are continuously checked, and the application is only used during suitable conditions.

4.3.3 Choosing a Mitigation Strategy

The overall goal of any mitigation strategy is the prevention of system failures resulting from faults, and ideally, a full recovery (see Fig. 7). Moreover, failure conditions are to be avoided at all costs. Preventive decisions should be made such that recovery is not needed in the first place. Finally, if the system configuration offers multiple degrees of freedom, these shall be exercised to enable optimal performance of all applications composing the system.

The simplest solution is only to reconfigure on observed failure. However, that does not play well with small, non-permanent errors and ODD monitoring. The reconfiguring of the system would approach the smallest, most robust configuration, ignoring configurations that could display higher performance or more autonomy. For this reason it might make sense to annotate the configurations with a static score, that indicates the configuration's performance. Now, provided that a higher scoring configuration is deemed feasible (e.g. all its ODDs are fulfilled and none of its applications is expected to be faulty), a reconfiguration may be triggered. This would be a greedy approach towards reconfiguration. The biggest caveat with this approach is, that a partition's failure might only be visible when it is in use. In that case configuration oscillation may occur: the system observes a failure in a partition, a reconfiguration is conducted removing the failing partition from active usage. The failure therefore becomes unobservable while the previous configuration had a higher score, immediately the decision is made to reconfigure back to the first configuration. Oscillations lead to overhead from constant reconfiguring and hence should be minimized. Furthermore, at least one of the configurations that are traversed is faulty and might remain faulty indefinitely. Consequently, that configuration should not be reached too often.

The underlying decision problem is complex. It involves a state that is only partially observable and subject to outside influences. If partitions belonging to a certain configuration exhibited observable failure in the near-past, configurations involving these partitions should be rejected for reconfiguration decisions. However, a partition should not be considered definitely faulty only because it once demonstrated failure in the far past. The same holds true for ODD-based reconfiguration. Considering these requirements, it becomes evident that the reconfiguration decision should not solely rely on current configuration and the system's health state.

One possible angle at this problem would be viewing it as a learning problem: given the system's current environmental situation, a policy has to be learned that enables a high configuration score paired with no observable failures. As we established the observable failures depend on also on external, invisible state. Therefore, some guided form of try-and error is required, which in term makes this a particularly challenging reinforcement learning problem. This allows the reconfiguration mechanism to explore the state space and adapt itself, even in uncharted territory. Still, in accordance with Level 3A autonomy, some areas may remain where a human in the loop can override the decision.

5 Outlook & Conclusion

Reviewing the aforementioned considerations, we conclude the following: dynamic reconfiguration will be a key component for dependable, autonomous avionic systems. To achieve level 3A autonomy, a system must be able to supervise itself, even under challenging conditions. Due to the nature of most autonomy functionalities, they do however not function in all imaginable environments. When challenges due to changing environmental conditions emerge, it is the supervisor's duty to mitigate the effects as far as possible. This is the problem that dynamic reconfiguration can address. The concepts introduced describe how the reconfiguration can be implemented, but also how to determine when to reconfigure.

For IMA-based platforms, reconfiguration at partition level is advantageous compared to reconfiguration inside the partitions. This is primarily the case as the latter threatens to break the principles of partitioning, which is demanded by DO-178C. When reconfiguring, the ARINC 653 P2 feature of multiple module schedules is to be used only sparsely. While changing between schedules does allow to maintain a higher CPU saturation contributing towards efficient usage of resources, doing so requires careful handling of partition warm-up times and awareness for the limitations of the hypervisor used. A more traditional hot-spares approach where the scheduling is not changed between reconfiguration is easier to get right. A combination of both strategies, where hot-spares are preferred, seems to be a feasible compromise. When considering the reconfiguration of communication channels, both a selection inside each partition and a dedicated routing partition seem feasible. The former clearly is beneficial for low-latency set-ups, while the latter has preferable properties in a greater scope concerning multiple IMA modules (and therefore multiple hypervisors). Only the latter is capable of supporting the runtime migration of an application from one hypervisor to another hypervisor. The decision when to reconfigure has to be based on a policy that both anticipates external, invisible factors. In practice that means that different reactions may be advisable even if two situations are perceived equally by the monitoring. Paying attention to this is crucial to achieve true self-supervision.

With this work, we have conducted a survey of design options for dynamic reconfiguration of APEX systems. For the future we aim to complete a demonstrator to validate our findings. This includes the following remaining steps: A routing partition has to be implemented and tested, that forwards messages between partitions according to the current configuration. How its various error cases are to be handled remains unclear at this stage and requires further work. Then, a decision algorithm has to be developed that fulfills the properties introduced in subsection 4.3. If indeed methods from the reinforcement learning domain were to be used, further work will be needed to prove that the algorithm's learning never turns toward inexpedient decisions. Finally, these code segments are to be combined with a handful of example partitions forming a working demonstrator, with which both the reconfiguration architecture and its implementation may be exercised. Provided that the demonstrator validation yields positive results, a further step will be to scale our approach to multiple hypervisors running on separate modules. That way, hardware redundancy can be relied upon and used by the reconfiguration implementation as well.

Bibliography

[AR12] ARINC: Avionics Application Software Standard Interface Part 4 Subset Services. Standard, Aeronautical Radio Incorporated, Bowie, MD, USA, December 2012.

[AR19a] ARINC: Avionics Application Software Standard Interface Part 0 Overview of Arinc 653. Standard, Aeronautical Radio Incorporated, Bowie, MD, USA, August 2019.

[AR19b] ARINC: Avionics Application Software Standard Interface Part 1 Required Services. Standard, Aeronautical Radio Incorporated, Bowie, MD, USA, December 2019.

[AR19c] ARINC: Avionics Application Software Standard Interface Part 2 Extended Services. Standard, Aeronautical Radio Incorporated, Bowie, MD, USA, December 2019.

[Bi09] Bieber, Pierre and Noulard, Eric and Pagetti, Claire and Planche, Thierry and Vialard, Francois: Preliminary design of future reconfigurable IMA platforms. ACM SIGBED Review, 6(3):1–5, oct 2009.

[Co95] Cook, A.: ARINC 653 — Challenges of the present and future. Microprocessors and Microsystems, 19(10):575–579, 1995.

[Du16] Durrieu, Guy; Fohler, Gerhard; Gala, Gautam; Girbal, Sylvain; Gracia Pérez, Daniel; Noulard, Eric; Pagetti, Claire; Pérez, Simara: DREAMS about reconfiguration and adaptation in avionics. In: ERTS 2016. Toulouse, France, January 2016.

[EA21] EASA: First usable guidance for Level 1 machine learning applications. Concept paper, RTCA, 2021.

[En10] Engel, Christian; Jenn, Éric; Schmitt, Peter H.; Coutinho, Rodrigo; Schoofs, Tobias: Enhanced Dispatchability of Aircrafts using Multi-Static Configurations. In: ERTS2 2010, Embedded Real Time Software & Systems. Toulouse, France, May 2010.

[KF19] Koopman, Philip; Fratrik, Frank: How Many Operational Design Domains, Objects, and Events? In: SafeAI@AAAI. 2019.

[Ló12] López-Jaquero, Víctor; Montero, Francisco; Navarro, Elena; Esparcia, Antonio; Catal'n, José Antonio: Supporting ARINC 653-based dynamic reconfiguration. In: 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture. IEEE, pp. 11–20, 2012.

[RT05] RTCA: DO-297 Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations. Standard, RTCA, 2005.

[RT11] RTCA: DO-178C Software Considerations in Airborne Systems and Equipment Certification. Standard, RTCA, 2011.

[Se96] Seeling, K.: Reconfiguration in an integrated avionics design. In: 15th DASC. AIAA/IEEE Digital Avionics Systems Conference. IEEE, 1996.