Enhanced Dispatchability of Aircrafts using Multi-Static Configurations

Christian Engel*, Eric Jenn†, Peter H. Schmitt*, Rodrigo Coutinho‡, Tobias Schoofs§

* Karlsruhe Institute of Technology (KIT)
Institute for Theoretical Computer Science
D-76128 Karlsruhe, Germany
pschmitt@ira.uka.de engelc@ira.uka.de

† Thales Avionics (THAV)
F31036 Toulouse, France
eric.jenn@fr.thalesgroup.com

‡ Embraer Research and Development
São José dos Campos, Brazil
rmcoutinho@embraer.com.br

§ GMV Portugal
1998 - 025 Lisboa, Portugal
tobias.schoofs@gmv.com

***Abstract*—This paper describes the reconfiguration strategy and mechanisms adopted in the Integrated Modular Avionics (IMA) based platform designed and evaluated in the scope of the European research and development project DIANA. The mechanisms aim at improving dispatchability of aircrafts while keeping a reasonable and limited impact on certification costs.**

The paper first introduces the concept of *multi-static reconfiguration* i.e., a set of pre-qualified configurations from which the active one will be autonomously selected according to the system health state at system start-up. A configuration selection mechanism, exploiting a *Byzantine Agreement* algorithm, is discussed. Particular attention is paid to the proof of correctness of the adopted algorithm. Practical considerations concerning its implementation, like, for instance, the authentication protocol to be used are also considered. Finally, the implementation of the mechanism on top of an ARINC 653 Application Executive is briefly described.

Keywords: Transportation; Architectures and Algorithms; Distributed and Reconfigurable Architecture; IMA; Byzantine Agreement

I. Introduction

The Distributed, equipment Independent environment for Advanced avioNics Applications, short DIANA, is an aeronautical research and development project funded through the European Commission's 6th Framework Programme, supported by Windriver, OIS and CES and led by GMV, Portugal.[1] It aims at the definition of an advanced avionics platform, called AIDA (Architecture for Independent Distributed Avionics), supporting execution of object-oriented applications over virtual machines, secure data distribution services, and a tool chain supporting model-driven engineering.

The overall objective of AIDA is to reduce the costs of avionics software development thanks to reduced development, validation, integration and recertification time and effort. These reductions are achieved by using efficient infrastructure mechanisms such as publish and subscribe services inspired by the OMG standard DDS (Data Distribution Services) [15], loosely coupled architectures based on partitions (and inspired by conventional component-oriented programming), new platform services such as redundant logbooks, "real-time" virtual machines, and modern development approaches like the use of models inspired by the OMG standard MDA (Model Driven Architecture) [14].

Backward compatibility with technological standards widely accepted in civil aviation is a mandatory condition to achieve acceptance of new solutions and related cost reduction. AIDA is therefore based on the Integrated Modular Avionics (IMA) approach and exploits the ARINC 653 Application Executive (APEX) [2] for all middleware implementations.

IMA and APEX decouple software functions from underlying hardware devices and allows, in consequence, a more fine-grained assignment of software components to processing nodes. AIDA extends IMA by supporting a first and limited, yet extensible, level of reconfiguration. To avoid a growth of software complexity beyond acceptable limits (in particular, in terms of certification effort), reconfiguration capabilities are actually restricted: At start-up, an AIDA

[1]Contract FP6 AST-CT-2006-030985, see www.dianaproject.com

compliant system selects autonomously the configuration that matches the system's health state among a pre-defined and pre-qualified set of configurations. This approach is called *multi-static reconfiguration*.

In principle, multi-static reconfiguration could be applied to in-flight reconfiguration as well. However, since this would introduce new challenges to safety, DIANA focuses on on-ground reconfiguration for the development and demonstration of the approach.

By reassigning applications to processing resources, multi-static reconfiguration represents a means to ensure that the conditions for the aircraft to be allowed take-off, namely, those of the Minimum Equipment List (MEL), are satisfied in the presence of one or several failed equipments. The final result is a potential reduction of the amount of hardware resources needed to achieve the required level of dispatchability.

Section 2 discusses multi-static reconfiguration in and beyond the IMA context. Section 3 introduces the Byzantine Agreement algorithm used to reach consensus on the health state of a system. The selected protocol and its adaptation to the current problem are described and a proof of its correctness is outlined. Section 4 shows how multi-static reconfiguration has been implemented on top of the ARINC 653 API. Finally, Section 5 provides some conclusions and discusses open issues left for future work.

II. From Static Configurations to Multi-Static Configurations

A. The IMA Paradigm

AIDA is based on the concepts of Integrated Modular Avionics and supports the ARINC 653 APEX [2]. ARINC 653 defines robust partitioning in onboard systems, such that one processing unit, usually called a module, is able to host one or more avionics applications and to execute them independently. Such segregation relies on a combination of software and hardware mechanisms that provide fault containment, such that an error in one application cannot propagate to another application, and more generally that decouple applications. As consequence, partitioning eases integration, verification, validation, and certification.

The unit of partitioning is called a partition. A partition correlates to a program (and the underlying operating environment) in a single application environment: it comprises data, code and its own context configuration attributes.

Partitioning segregates applications in the space and time domains:

- In the space domain, segregation means that the state space of an application (its variables, code, logbooks, IOS, etc.) is protected against any unexpected modification by another application residing in another partition.
- In the time domain, segregation means that the temporal characteristics of an application (response time, jitter, etc.) cannot be affected by the activity of another application residing in another partition. In practice, this means that there is no competition for system resources, such as the CPU, between partitioned applications. This is ensured, in particular, by a statically configured scheduling performed by the so-called Module Operating System, or MOS.

IMA introduces a paradigmatic change in how avionics systems are seen. In traditional federated approaches, the avionic system was assembled by subsystems, consisting of hardware devices inseparably compound with software items.

IMA substitutes subsystems by standardised modules, linked together to form a network. This network serves as a hardware platform to host software applications. Of course, there are still hardware elements that are specific to certain functions, like actuators/sensors, data concentrators and so on, limiting the possible combinations of software and hardware components, but the much looser coupling between software and hardware is an essential difference to the federated design.

B. Configuration and Reconfiguration in IMA-based Systems

A configuration of an IMA system is to a certain extent a mapping of software components to hardware resources. In more formal terms, we will define a configuration as a *function of a set of applications to a set of modules*. The set of modules, a set of applications can be mapped to, is called the *configuration domain* of these applications.

IMA configuration design is a critical activity of avionics system design and integration. The configuration has to ensure that all functional and non-functional requirements (integrity, availability, performance, etc.) are met. An application controlling cabin pressure, for instance, requires triple redundancy to meet its availability and integrity requirements. A less critical application, an on-board maintenance system, to pick a simple example, may not require any redundancy at all.

Fleet managers are concerned with operational dispatchability rates of aircrafts due to costs associated to ground aircraft management and flight cancellation. To ensure high dispatchability levels and at the same time assure appropriate safety operations, avionic system design usually demands additional hardware redundancy, increasing equipment costs, weight and power consumption, resulting in higher fuel consumption.

An aircraft is only able to take-off (dispatch) if the equipments listed in the Minimum Equipment List (MEL) are operating properly. To ensure this, the aircraft system's health state is determined at start-up. If any essential aircraft functions (rolled on MEL) do not meet the required redundancy level, the aircraft is not able to take-off, and a NO GO indication is placed.

The objective of AIDA's on-ground reconfiguration is to meet required dispatchability levels, but without the same equipment, weight and power consumption penalties. The idea is to have more than one configuration for subsets of the available hardware. The alternative configurations are designed to provide alternative allocations of essential

functions to available hardware resources in a way that MEL requirements still being attended. This functional reallocation process could potentially leave some of the non-critical functions out.

A large number of possible configurations may exist among which an optimal one shall be selected, possibly dynamically at runtime. However, to keep complexity and, hence, certification effort to a reasonable level, acceptable configurations are defined and qualified offline, during aircraft development. If failures are detected at system start-up, the system selects the new configuration among a static set of configurations. This approach is called *multi-static configuration*.

How reconfiguration can be implemented by platform-independent IMA-based middleware is currently investigated in the FP7 project SCARLETT[2]. Similar to DIANA, SCARLETT aims at on-ground reconfiguration, by reassigning partitioned software components on faulty modules to healthy ones.

SCARLETT foresees a set of platform functions to support reconfiguration, such as data-loading, monitoring and fault detection and power supply to switch modules on and off [5]. Failures are detected and repaired if possible by monitoring systems. In the case of a non-repairable failure, an event is sent to a central supervisor component that starts a transistion to an alternative configuration, by unloading and uploading partitions or by restarting and stopping modules.

The AIDA platform adopts an alternate solution to such a centralised approach. The main argument to do so is integrity and availability. In a centralized architecture, the non-detected failure of the centralized supervisor has a potential impact on *all* controlled functions, so the dependability requirements applicable to the supervisor are at least as high as the dependability requiremnts for the most demanding function, without even considering the impact of the joint failure of multiple independent functions. In practice, the supervision function will itself rely on a redundant physical architecture.

In a non reconfigurable architecture, this is not necessarily a big issue: in any case, the aircraft will be stucked on ground if any critical hardware equipment of the MEL is not available, so the availability of the aircraft (hence, its dispatchability) is directly determined by the availability of any of those equipments (if we are only considering that aspect). In that case, the supervisor does not need to be more available than any critical function. Actually, it shall be as available as the combination of all elements in the MEL.

In a reconfigurable architecture, things are different: decoupling software and hardware and providing some flexibility to map the one to the other allows critical functions to be available as long as there is a compatible configuration of available hardware. This does not stand for the supervisor which shall be available *before* any reconfiguration. This does not mean that the centralized approach does not work, but the reconfiguration function is treated in a specific way whereas in the distributed approach, all functions, including the one managing reconfiguration benefit from multi-static reconfiguration.

AIDA, instead, uses a distributed approach where middleware components, controlling reconfiguration events, are hosted on each module in the configuration domain. This way, the dependability of a critical application is distributed among all nodes in the configuration domain, no single module is alone in charge of the critical action of selecting an alternative configuration.

In an AIDA system, each module determines its own health state on start-up, by means of power-up built-in tests (PBIT). Afterwards, all modules within the same configuration domain exchange their health state, again represented, for instance, by the PBIT result, using a Byzantine Agreement protocol. The protocol ensures that all non-faulty modules will eventually obtain the same view of the health states of all interacting modules, even in case of arbitrary faults. This view is called the *system health state*.

The system health state is mapped to a configuration using a static list of configurations. If the configuration corresponding to the current system health state differs from the active configuration, all non-faulty modules select the new configuration and reboot. The new configuration then acts as the default configuration until the system is repaired and the default configuration is reset.

Figure 1 shows a simplified reconfiguration scenario. There are two configurations, denoted C0 and C1. C0 is the default configuration, C1 is a configuration that remaps applications on modules M1 – M3 in the case of failure of module M4.

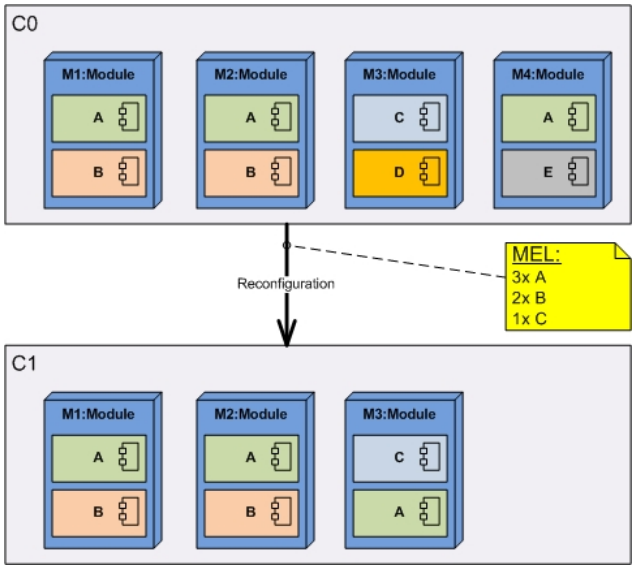

Figure 1: A Reconfiguration Example

In C0, five applications are hosted on the modules. Applications A – C are listed on the MEL, A with triple redundancy and B with double redundancy. With module M4 lost, the third instance of A must be hosted somewhere else or a NO GO indication will be placed. Configuration C1 takes advantage of the non-criticality of D and E: D

[2]Contract FP7 AAT-2007-RTD-1-211439, see: www.scarlettproject.eu

on module M3 ist substituted by the third instance of A, E is dropped. C1, thus, complies to the MEL and the aircraft can remain in service. The dropped functions D and E will be reactived when the aircraft reaches a location where the faulty module can be replaced and C0 restored as default.

In our approach, human involvement has been reduced to a minimum. It is assumed that the pilot is always informed (in terms of some high-level view of the system, such as available functions or equipments), but not forced to confirm the change of a configuration. The only situation where configurations have to be replaced manually, is the resetting of the default configuration. However, this task will be part of the fault correction that already implies human activity.

A Byzantine Agreement protocol has been selected to take into account faults that may result in arbitrary behaviour of the affected module – more precisely, such a behaviour, called a Byzantine fault, *is a fault presenting different symptoms to different observers* [7].

There are two arguments that strongly suggest considering Byzantine faults:

- There is evidence that those faults actually occur (see [7]);
- The effort to prove that they cannot occur or are, at least, not relevant in the current context appears to be much higher than using well known algorithms designed to cover this kind of faults.

The idea to use Byzantine Agreement protocols in avionics systems is not new. The theoretical underpinnings have been established in the pioneering papers [16], [12]. One of the first applications to avionic systems was proposed in [20] for clock synchronisation. In the late 70ies and 80ies, the SIFT and MAFT architectures have been proposed [9] to detect failures in on-board computers and to switch to redundant equipment to tolerate them. This comes already close to the use of Byzantine Agreement protocols in AIDA. However, AIDA differs in various respects. First, the architectural environment, IMA, is radically distinct from the federated approach of the SIFT and MAFT architectures. MAFT, for instance, foresees two computers forming a node in an avionic network: the application processor, hosting the application component itself and the operation controller, acting as processor in the Byzantine Agreement protocol. In sharp contrast, AIDA reduces on-board hardware to save volume, weight and consequently operational costs. Application components are distributed on a set of modules, such that each module hosts one or more applications.

Second, AIDA exploits IMA to enable a more fine-grained level of redundancy: the unit of redundancy is not the module, but the partition. There are much more possible reconfiguration scenarios for this granularity level. This implies stronger reliability; but it also implies a more complex definition and selection of configurations. Multi-static reconfiguration aims at tackling this complexity.

III. The Byzantine Agreement Protocol

We consider a set M of available modules that may either be faulty or non-faulty. The protocol is initiated by a distinguished module, called the *transmitter*, sending some information. In our application this information will be the result of its own PBIT, to all other modules, which will be called the *receivers*. The receivers then communicate among themselves to reach a consensus on the health state of the transmitter. To obtain the complete health state of the system this protocol has to be repeated with every module in turn in the role of the transmitter.

Byzantine agreement protocols come in two major varieties, which are for historical reasons called protocols with *oral* messages or with *written* messages. In the oral message model a faulty receiver may either drop a received message or resend messages with arbitrary values. In the written message model a faulty receiver may drop a message or forward the (extended) message but only with the unchanged received value. To guarantee this behavior in an implementation, messages must be authenticated. In the literature the term signed messages is therefore sometimes used instead of written messages.

The goal is to find algorithms that compute for every receiver rr its opinion $hst(rr)$ on the health state of the transmitter satisfying the following two properties:

Validity If the transmitter tt is non-faulty then for all non-faulty receivers $hst(rr)$ is the value sent by tt.

Agreement Any two non-faulty receivers rr_1, rr_2 agree, i.e., $hst(rr_1) = hst(rr_2)$.

Numerous algorithms have been published that ensure these two properties for oral and written messages. Most proposed algorithms divide message communication into individual rounds. The number of rounds is thus one dimension in the evaluation of the algorithm. Another dimension is the number of modules required to withstand a certain number of byzantine faults. Let us denote by f the number of faulty modules. The protocols with oral messages need at least $3f+1$ modules and $f+1$ rounds while in the case with written messages $f+2$ modules and $f+1$ rounds suffice, see e.g., [12], [8] or the comprehensive survey on consensus problems in [4]. Since in our application the benefits from greater fault resilience and the possibility to add application specific optimizations outweigh the additional efforts of using some signature scheme we decided to use a protocol with signed messages.

We will shortly describe our version of the Byzantine agreement algorithm $Byz(m, f)$, where m is the number of modules in M and f is the number of faulty modules. In the worst case $m = f + 2$. But in general, dependability analysis will allow us to determine a smaller number f such that the probability of a joint failure of more than f modules is so small that it can be neglected with respect to the overall dependability objective.

Every module $m \in M$ keeps an initially empty set $val(m)$ of seen values and every message contains the list of its previous senders snd. A message sent by a non-faulty

module will reach its receiver in less than T time units. More precisely, T is an upper bound for the time that will pass between a non-faulty module i receiving a message, processing and sending it to j and the moment when j receives it.

Step 1 The transmitter sends a signed message to all receivers.
Receiving a message with value v, sender list snd at time t receiver rr proceeds as follows

Step 2 If $t \geq (f+1) * T$ goto step 4 (global time-out).

Step 3 If

- $\mathbf{v} \in \mathbf{val(rr)}$ or $\mathbf{t} > \mathbf{length(snd)} * \mathbf{T}$ discard message, goto step 2.
- $\mathbf{length(snd)} \geq \mathbf{f+1}$ Add v to $val(rr)$, goto step 2.
- otherwise Add v to $val(rr)$. Receiver rr signs the message and resends it to all receivers not in snd and updates snd by adding itself and goes to step 2.

Step 4 Return $hst(rr) = sel(val(rr))$.

A sensible choice for sel is the function that yields $sel(\{v\}) = v$ and $sel(V) =$ faulty if V is empty or contains more than one element.

The algorithm $Byz(m, f)$ differs from those presented in [8] and [18, Section 2.2] in that it is formulated as an iterative instead of a recursive algorithm, mentions some of the data structures needed for implementation, and addresses the issue of time-out explicitely. Algorithms close to ours are to be found in [12] and [10, Section 2.6.1]. We follow the usual abstraction that a failure in communication is attributed to the sending module. This has the consequence that any message sent by a non-faulty module is assumed to arrive.

Theorem 1. *$Byz(n, f)$ satisfies the validity and agreement property.*

Proof Since by the signed message assumption no new values can be introduced by the receivers validity is trivially statified. For any message with value v and sender list snd it is easily established that for all $j \in snd$ we must have $v \in val(j)$.

To argue that agreement is satisfied, we will prove the stronger claim that $val(i) = val(j)$ for all non-faulty modules i, j.

Since the selection function is common to all modules this entails $hst(i) = hst(j)$. We thus assume for two arbitrary modules $i, j \in M$ that $v \in val(i)$ holds and aim to show $v \in val(j)$.

Consider the message with value v and sender list snd when i first saw v. For future reference let t denote the time when i did receive this message. If $j \in snd$ then $v \in val(j)$ by the previous observation. So we assume from now on $j \notin snd$. We have in any case $t \leq length(snd) * T$. If $length(snd) \leq f$ then i did send value v to j, who did receive it within the given time bounds. If $length(snd) \geq f+1$ then there must for cardinality reasons be a non-faulty module $k \in snd$. Since the sender list snd_1 of the message that k forwarded to i is an initial segment of the list snd, we know that j is not in snd_1. Thus k forwarded it also to j. Since k was chosen as a non-faulty module the assumptions of our model imply that j did in fact receive the message, say at time t_1. We still need to convince ourselves that j did not discard it. Since we found k on the sender list snd we know that k did not discard the message snd_2 it received at time, say t_2. From the protocol specification we thus know $t_2 < length(snd_2) * T$. Since $lenght(snd_1) = lenght(snd_2) + 1$ and the assumption on T guarantees that at $t_1 \leq t_2 + T$ this implies $t_1 < length(snd_1) * T$ and j did indeed not discard the message. Thus $v \in val(j)$. □

As can be seen from the proof the correctness of the algorithm strongly depends on the proper choice of the time bound T.

If we strengthen the assumptions on the time bound T and require in addition for *all* modules i, j with i faulty: either i does not send a message to j or it is guaranteed that the message arrives within T time units. Under these assumptions the guard $t > length(snd) * T$ can be dropped in Step 3 and only the global time-out is retained.

The algorithm $Byz(m, f)$ has been specified with the Event-B tool and its correctness formally verified using the Rodin platform, [1]. To the best of our knowledge this is the first time a Byzantine protocol using written messages has been analysed using deductive verification. The formal verification in [18, Section 8.4] uses model checking and covers only systems with three modules, and under some simplifications also systems with 4 modules. Details of our verification may be found in the technical report [11]. Formal correctness proofs for Byzantine algorihms with oral messages using the interactive verification system PVS have been reported in [13].

As a next step towards an implementation we should decide on which signature schemes should be used. In [18] the use of cyclic redundancy checks (CRC) (see [17]) is proposed. This is a plausible choice since CRC works well to detect accidental changes.

We explain the method from [18] by an example with three modules and two rounds. Module 1, acting as the transmitter, sends messages with value D to module 2 and 3:

Mod1 → Mod2 : M12 = D:(CRC(D,K12),CRC(D,K13))
Mod1 → Mod3 : M13 = D:(CRC(D,K12),CRC(D,K13))

Here $Kij = Kji$ are keys (polynomials) commonly known to each pair $\{i, j\}$ of modules. Module 2 checks the integrity of the message, as far as it can do it, by computing CRC(D,K12) and compares it with the value received in the first position. If no error is found module 2 forwards the message to module 3. Module 3 acts symmetrically. There is not explicit slot for snd, the list of previous senders, but obviously this information can be retrieved. Let us look at an example where a faulty transmitter sends the following two inconsistent messages

Mod1 → Mod2 : M12 = 0:(CRC(0,K12),CRC(1,K13))
Mod1 → Mod3 : M13 = 1:(CRC(0,K12),CRC(1,K13))

Both modules, Mod2 and Mod3, believe in the integrity of the received messages and resend it

Mod2 → Mod3 : M23 = 0:(CRC(0,K12),CRC(1,K13))
Mod3 → Mod2 : M32 = 1:(CRC(0,K12),CRC(1,K13))

Module 3 discards M23 as not integral and votes for 1 as its guess for the value sent by the transmitter. Module 2 likewise discards M32 and votes for 0. Thus the agreement property of the protocol is violated.

How can Theorem 1 be true in the face of this example? The problem arises in that $Mod2$ is not able to detect that message $M23$ is inconsistent, but the receiver $Mod3$ will find this out. Thus the situation arises where $Mod2$ sends message $M23$, but this message is not received. According to our modeling assumptions $Mod2$ has to be counted as faulty. The same applies to module $Mod3$. We are thus looking at a system with three faulty modules. It is easily seen that any authentication scheme where different modules use different keys to check incoming messages will lead to the same behavior.

A possible solution, already investigated in [18], could rely on assymetric public/private keys protocols, and this is actually the one we employed for the development of AIDA. However, the cost of those algorithms is very high, so we are currently investigating how they could be "relaxed" e.g., by reducing the length of the key, so as to still provide a "sufficient" error detection coverage and to comply with both space and time constraints of airborne systems. Preliminary theoretical investigations have been put forward in [6].

IV. Implementation, application and results

The algorithm has been implemented on two demonstrators, a simplified Flight Warning System (FWS), developed by THALES and a cabin Airconditioning system (A/C), developed by NLR.

Each of the two demonstrators is implemented on three nodes; the FWS uses two PowerPC modules with the VxWorks 653 operating system by Windriver and an Intel PC with the VxSim simulator, also by Windriver. The A/C application is hosted on two Intel PCs, run by GMV's ARINC 653 simulator SIMA and a PowerPC module with VxWorks 653.

Two fault scenarios have been defined for each demonstrator, with either a PowerPC or an Intel node failing. The faults have been injected both, by simulation of wrong PBIT results and, simply, by not activating the respective module.

The heterogeneity of the systems had to be taken into account in the design, coding and parametrisation of the multi-static reconfiguration. First, an overall timeout, valid for all modules had to be found. To achieve this, the algorithm has been benchmarked on the different target systems. The main problem turned out to be the start-up procedure. No synchronised start-up had been defined for the demonstrators and, even worse, different start-up scenarios – for demonstration to an audience and for benchmarking in the lab – have been identified. Finally, different tolerance delays, between two seconds and two minutes, and overall timeouts, between twenty seconds and three minutes, have been chosen for different demonstration purposes.

Another issue that had to be solved, is the module reset and passivation mechanism. For the reset, the ARINC 653 health monitor has been used on VxWorks, VxSim and SIMA. An application error is raised by the reconfiguration engine that is not handled within the partition and, hence, propagated to the partition health monitor where RESET was defined as the corresponding error response action.

The passivation was handled differently; SIMA provides a *shutdown* function that can be called from a priviledged system partition. VxWorks 653, however, does not offer such a functionality. Instead, the Multiple Module Schedules service, defined in ARINC 653 Part 2 [3], has been exploited by switching to an empty schedule.

The system-specific code has been implemented in a system partition that answers service requests by the partition, hosting the reconfiguration engine. There is one generic system partition per module, implementing also other system-specific services that may be requested by applications. This way, the overhead, introduced by the reconfiguration approach, is kept to a minimum.

The reconfiguration engine is hosted on one partition per module. This partition is connected to the reconfiguration engines on other modules by queuing ports that implement the channels in the Byzantine Agreement protocol. This is depicted in figure 2. Note that it is possible to connect payload applications to the reconfiguration engine by queuing ports. A middleware component is available to request information about the active configuration.

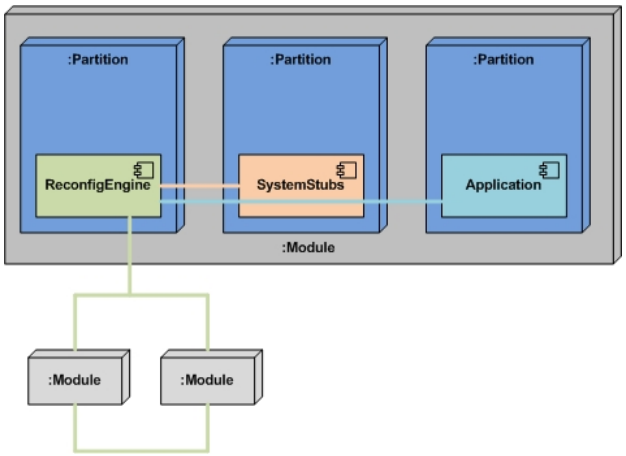

Figure 2: The AIDA Reconfiguration Engine

Again, the Multiple Module Schedules service has been used to keep the overhead as small as possible: After a successful completion of the algorithm, i.e., when at the end of the algorithm the new configuration is equal to the current configuration, a switch to a schedule is requested that does not contain the execution windows for the reconfiguration engine anymore. In consequence, the reconfiguration engine will not consume any time resources after the system has entered the operational phase. Note that this behaviour excludes communication between the

reconfiguration engine and applications. Which option to chose depends on application requirements and available system resources.

In the demonstrators, a standard RSA algorithm has been used for message authentication. By shortening the key length, as mentioned in Section III, the algorithm may be optimised in terms of memory and time consumption.

How configurations are defined with respect to files and formats, depends on the operating system. In general, each configuration consists of a set of binaries and configuration files that are loaded into memory at start-up. Typically, the loader searches for a file according to some naming convention which serves as entry point to a given configuration. In the demonstrators, this file is changed on behalf of the reconfiguration engine by means of a helper script, running on the host. For a real solution, this helper must be substituted by a component in the OS infrastructure.

V. Conclusion and Future Work

The AIDA platform makes a first step towards an optimal usage of available computation resources. It proposes solutions to take benefit from the genericity of the IMA platform, its capability to decouple application and hardware, and its strong partitioning features. With respect to other solutions, AIDA favors a truly distributed reconfiguration logic, and offers resilience to Byzantine faults.

In the paper, we have essentially addressed platform independence at hardware level, on the basis of IMA and ARINC 653 Application Programming Interface. Towards the same goal, DIANA also investigated the use of Java for the development of real-time airborne applications. This aspect is addressed in more details in paper [19]. In particular, both demonstration applications (simplified FWS and Airconditioning systems) have been developed using ATEGO's innovative real-time version of Java.

Among the various issues only partially covered in DIANA, is the management of configurations. Indeed, each configuration is a system of its own that has to be defined, integrated and validated separately. A tool to support this activity has been specified in the scope of the DIANA project, but, due to limitations on time and budget, it has not been developed. In consequence, the intragration of the alternative configurations had to be done manually. The problems encountered during this activity helped the project team to better understand the requirements for a tool chain supporting reconfiguration, in particular for supporting different target platforms.

An issue that is still open at project end, is a good integration of alternative configurations in one modelling approach. Currently, different models on all levels of the modelling tool chain, such as Platform Independent Models (PIM) and Platform Specific Models (PSM), have to be developed and maintained. This remains an issue for future research.

Management of physical inputs and outputs (IOS) is another complex issue that has not been studied in detail during the project and that definitely needs further analysis and tool support. Indeed, IO introduce a strong physical coupling between application and modules, and strongly constraints the applicable reconfiguration schemes. Similar constraints stem from the sharing of network resources, which are strongly affected by reconfigurations.

Finally, tolerance to Byzantine faults is restricted to the management of the system, and the proposed mechanism is not accessible to applications. This limit could also be overcome, and this feature could also be proposed as a means to support specific quality of service for data distribution.

References

[1] Jean-Raymond Abrial. A system development process with Event-B and the Rodin platform. In Michael Butler, Michael G. Hinchey, and María M. Larrondo-Petrie, editors, *ICFEM*, volume 4789 of *Lecture Notes in Computer Science*, pages 1–3. Springer, 2007.

[2] Airlines Electronic Engineering Committee (AEEC). *Avionics Applications Software Standard Interface (ARINC Specification 653 Part 1 – Required Services)*. ARINC Inc., 2006.

[3] Airlines Electronic Engineering Committee (AEEC). *Avionics Applications Software Standard Interface (ARINC Specification 653 Part 2 – Extended Services)*. ARINC Inc., 2008.

[4] Michael Barborak, Miroslaw Malek, and Anton Dahbura. The consensus problem in fault-tolerant computing. *ACM Comput. Surv.*, 25(2):171–220, 1993.

[5] Pierre Bieber, Eric Noulard, Claire Pagetti, Thierry Planche, and François Vialard. Prelimenary design of future reconfigurable IMA platforms. In *APRES*, Grenoble, October 2009.

[6] Malte Borcherding. Partially authenticated algorithms for Byzantine agreement. In K. Yetongnon and S. Hariri, editors, *Proc. 9th Int.Conference on Parallel and Distributed Computing Systems (PDCS'96) , Dijon*, pages 8–11, September 1996.

[7] Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phili Zumsteg. Byzantine fault tolerance, from theory to reality. In S. Anderson et al., editor, *Proc. SAFECOMP*, volume 2788 of *LNCS*, pages 235–248. Springer, 2003.

[8] Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In Ravishankar K. Iyer, Michele Morganti, W. Kent Fuchs, and Virgil Gligor, editors, *Dependable Computing for Critical Applications—5*, volume 10 of *Dependable Computing and Fault Tolerant Systems*, pages 139–157, Champaign, IL, Sept. 1995. IEEE Computer Society.

[9] Roger M. Kieckhafer, Chris J. Walter, Alan M. Finn, and Philip M. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Transactions on Computers*, 37(4):398–405, April 1988.

[10] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers, 2007.

[11] Roman Krenický and Mattias Ulbrich. Deductive verification of a Byzantine agreement protocol. Technical Report 2010-7, KIT, Institute for Theoretical Computer Science, 2010.

[12] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages*, 4(3):382–401, July 1982.

[13] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault-Tolerant Computing Symposium, FTCS 23*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society.

[14] The Object Management Group (OMG). *Model Driven Architecture Guide - omg/03-06-01*. OMG, 2006.

[15] The Object Management Group (OMG). *Data Distribution Service for Real-Time Systems - OMG Specification omg/07-01-01*. OMG, 2007.

[16] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[17] W. W. Peterson and D. T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49:228 – 235, 1961.

[18] David Powell. *A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems*. Kluwer Academic Publishers, 2001.

[19] Tobias Schoofs, Eric Jenn, Stéphane Leriche, Kelvin Nilsen, Ludovic Gauthier, and Marc Richard-Foy. Use of PERC Pico in the AIDA avionics platform. In M. Teresa Higuera-Toledano and Martin Schoeberl, editors, *JTRES*, ACM International Conference Proceeding Series, pages 169–178. ACM, 2009.

[20] John H. Wensley, Milton W. Green, Karl N. Levitt, and Robert E. Shostak. The design, analysis, and verification of the SIFT fault-tolerant system. In *ICSE*, pages 458–469, 1976.

VI. Glossary

A/C	Airconditioning
AIDA	Architecture for Independent Distributed Avionics
APEX	Application Executive
API	Application Programming Interface
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DDS	Data Distribution Services
EC	European Commission
FP	Framework Programme
FWS	Flight Warning System
IMA	Integrated Modular Avionics
IO	Input/Output
MAFT	Multicomputer Architecture for Fault Tolerance
MDA	Model-Driven Architecture
MEL	Miminum Equipment List
MOS	Module Operating System
OS	Operating System
PBIT	Power-up Built-In Test
PIM	Platform Independent Model
PSM	Platform Specific Model
SIFT	Software Implemented Fault Tolerance