This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TAES.2019.2915406, IEEE Transactions on Aerospace and Electronic Systems

Timing Assurance of Avionic Reconfiguration Schemes using Formal Analysis

A. A. da Fontoura[1] and F. A. M. do Nascimento[1] and S. Nadjm-Tehrani[2] and E. P. de Freitas[1]

***Abstract*—Reconfigurable avionics systems can tolerate faults by moving functionalities from failed components to another available system component. This paper proposes a distributed reconfigurable architecture for application migration from failed modules to working ones. The feasible system reconfiguration states are determined off-line to provide the expected configuration in foreseen situations. Model Checking is used to determine feasible configurations evaluating specific temporal properties. A case study is used to show the application of the presented approach as a proof of concept.**

Keywords: Reconfiguration, Avionic Systems, Distributed Real Time Embedded Systems, Schedulability Analysis, Model Checking

I. Introduction

Reconfiguration of distributed real time embedded systems consist of changing or modifying subsystems and/or subsystem configurations in order to better serve a certain purpose [1]. In an avionics system, mode changes are naturally used to adapt to changing operational flight conditions. While modes are predetermined, their realization can be through reconfigurations. Reconfiguration can be applied to tolerate faults which could cause the loss of a certain critical function in response to an external environmental change or under the request of a system user or even to a timed event in an application. The survey by Löfwenmark et al. [2] shows that fault-tolerant architectures continue to be an important area of research, and combining fault tolerance with timing guarantees is still unresolved, e.g. in presence of multicore architectures.

When a system component fails, a reconfigurable avionics platform moves the functionalities, which were allocated previously in the failed component, into another available system component. Such a reconfiguration scheme, in addition to enhancing reliability, can also be beneficial in terms of evolution capability throughout the aircraft life cycle.

The lifespan of commercial aircraft has been increasing since the end of the 20th century to the present 21st century [3] and has now reached stability. Additionally, the Maintenance, Repair and Overhaul (MRO) market is expected to produce a strong future demand as world wide military Air Forces decide to upgrade legacy aircraft rather than procuring new platforms [4], which gives military fleets an increased service life. In Brazil, for instance, a recent overhaul has brought a 70s vintage fleet the ability to extend its service life beyond 2020 [5].

Aircraft projects, either new platforms or overhauled, have increased the development time and thereby the costs substantially in recent years. Avionics technology obsolescence occurring earlier than the aircraft airframe lifespan is also a cause of the MRO market trend. The reconfiguration flexibility can partially alleviate such problems. Early deliveries with basic capabilities can be performed and more advanced functionalities can be incorporated into the system by changing the configuration.

Given the above-described landscape, this work proposes a distributed reconfigurable architecture in which a global agent and local agents cooperate to oversee that applications transition from failed modules to working ones. The feasible reconfigurations determined off-line are stored in the system to be used by the agents, which then keep the computers in a previously defined configuration in a foreseen situation.

The reconfiguration of one subsystem does not affect the rest of the system in any way. In other words, the original specified real-time constraints would still be satisfied. The sequence of necessary steps for the completion of a reconfiguration must be atomic, in the sense that they should entirely succeed or be discarded. In the case where a reconfiguration is aborted, the avionics system operation must not be affected in any way. In either case it is important to highlight that we assume that the failures occur one at a time.

This paper proposes the use of model checking [6] to determine the feasible reconfigurations, taking into account all possible sequences of necessary steps. From a specification, provided in Architecture Analysis and Design Language (AADL) [7], the proposed approach generates a network of automata [8], representing the timing aspects of an avionics system, in order to perform schedulability analysis of each possible reconfiguration. This is done by evaluating specific temporal logic properties on the timed trace of the avionics system tasks and observing their deadlines. This ensures predictability of the system after each reconfiguration and facilitates the airworthiness approval by the certification authorities.

The main contributions of this work are:

- a modeling approach where fault models augmenting the AADL specifications are combined with the reconfiguration logic to formally represent fault tolerance by transitions in a reconfiguration state space;
- a method to evaluate alternative reconfiguration strategies through model checking, in order to find suitable config-

[1]Informatics Institute, Federal University of Rio Grande do Sul, Brazil afontoura at inf.ufrgs.br, fanascimento at inf.ufrgs.br, epfreitas at inf.ufrgs.br

[2]Department of Computer and Information Science, Linköping University, Sweden simin.nadjm-tehrani at liu.se

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TAES.2019.2915406, IEEE Transactions on Aerospace and Electronic Systems

urations that satisfy timing requirements and provide the highest degree of fault tolerance in the considered space.

The structure of this paper is as follows. Section II presents related work and Section III describes the proposed reconfigurable system architecture. The proposed reconfiguration approach is detailed in Section IV and Section V presents its application in a representative avionics case study and discusses the obtained results. The conclusions are reported in Section VI, providing also directions for future work.

II. Related Work

Housseyni et al. [9] propose a multi-agent reconfiguration approach in a distributed real-time system with energy harvesting constraints. The objective is to optimize global Quality of Service (QoS) measured in terms of deadline success ratio, the degree of criticality and the energy harvesting. Three different strategies are applied for tasks adaptation depending on the reconfiguration environment and the task constraints: Decomposition, which decomposes software tasks and migrates their branches from a faulty processor to a non-faulty one; degradation, which modifies scheduling mode; and removal, which deletes branches or tasks. However, the proposed strategy of decomposing tasks is hard to achieve due to the high demands from aerospace software certification processes such as the DO-178C [10], especially in software with the highest degree of criticality. Moreover, in an avionics environment, all failure modes are usually identified in the development stage and analyzed in the safety assessment process. Therefore, all possible reconfigurations can be analyzed prior to the system implementation, making the multi-agent solution suitable for including fewer local agents as needed. The work by Housseyni et al. [9] did not address how the reconfiguration process affects time-critical tasks with hard deadlines as our approach does.

Cui et al. [11] suggest a decentralized reconfiguration technique, applying a concept called backward reconfiguration. A global component is responsible to assess the system reconfiguration state. The decentralization causes the system to adapt faster to the identified fault in a certain computer module or communication bus, but it increases the complexity of every node in the system. The avionics software development process dictates that unnecessary complexity is to be avoided. Moreover, a local reconfiguration can lead to effects encountered in heuristic algorithms such as *hill climbing* [12]. Also, *local maxima* [12] could mean the system has recovered from a component fault, but could end up in a failure state if a less critical node fails, bringing the overall probability of failure to an undesirable level.

Zhou et al. [13] propose a framework to support to the reconfiguration of avionic applications that adopt the distributed Integrated Modular Avionics (IMA) architecture. In the proposed framework, an action model conforming to the Behavioral Annex of AADL [7] is built to represent the sequence of all the steps required to perform a given application reconfiguration, aiming at fault tolerance. This behavioral model in AADL is then used to compute the total execution time required for the completion of the reconfiguration, as a sum of the execution times required for each step. The work does not include further steps linking this computed total time to application constraints or model checking. In addition, the model assumes that all steps are performed in sequence, when in fact, some steps can be executed in parallel. In our proposed approach, model checking is used to determine the feasible reconfigurations, considering all possible sequences of necessary steps.

Fohler et al. [14] describe a similar approach for a reconfigurable avionics system. Their work also includes more than one agent: a global, called Global Resource Manager (GRM), and a local, called local resource manager (LRM). The authors provide an independent local reconfiguration with no changes in other system units whatsoever. However, the paper does not include an analysis showing that erroneous outcomes of any reconfiguration attempt will not affect system timing.

Atitallah et al. [15] propose a converged unified environment for the simulation and test domains as well as the verification and validation of an avionics system focusing on reconfigurable architectures. Field programmable gate arrays are used in the system under test and in the test benches to accomplish a unified development environment with the objective to reduce cost and time-to-market. Our approach also targets system verification but in a more specific sense, to assure the timeliness of a certain system.

Montano et al. [16] present an approach to solve the complex combinatorial problem of IMA reconfiguration in real-time whilst providing support for pilots involvement by means of automatic generation of explanations of reconfiguration actions. The approach is based on Explanation-based Constraint Programming.

Porcarelli et al. [17] describe a framework providing fault tolerance of component-based applications by detecting failures through monitoring and by recovering through system reconfiguration. The framework is based on Lira, a distributed agent infrastructure for remote control and reconfiguration, and a decision maker for selecting suitable new configurations. The proposed solution is based on run-time calculations. This strategy is hard to employ in avionics contexts with certification, without the sort of pre-analysis that we suggest in this paper.

Hollow et al. [18] focus on the reallocation problem and propose a fitness function to be applied in conjunction with a search algorithm in order to find possible system states which can still fulfill the system requirements. However, the proposed solution does not include the means to confirm whether the new system schedule is still feasible. Our work proposes model checking of all possible end results to assure timeliness.

Finally, it is worth mentioning that the present work does not focus on the original allocation and mapping problem. There are relevant works that handle these problems, including the work by Annighofer et al. [19], who use multi-objective mathematical optimization to perform software and hardware mapping within a distributed IMA architecture while designing avionic systems.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TAES.2019.2915406, IEEE Transactions on Aerospace and Electronic Systems

III. Reconfigurable System Architecture

This section starts with the introduction of important terms used throughout this paper. *Function* is defined as an intended behavior of a product based on a defined set of requirements regardless of implementation; *Item* as a hardware or software element having bounded and well-defined interfaces; and, *System* as a combination of inter-related items arranged to perform specific function(s).

Following the D-178 standard [20], software development terms are used: *Parameter Data Item (PDI)* as a set of data that, when in the form of a *Parameter Data Item File*, influences the behavior of the software without modifying the *Executable Object Code* and that is managed as a separate configuration item (examples include databases and configuration tables); *Partitioning* as a technique for providing isolation between software components to contain and/or isolate faults; and, *Software Partition* as the process of separating software components, usually with the express purpose of isolating one or more attributes of the software, to prevent specific interactions and cross-coupling interference.

Figure 1 illustrates the proposed reconfigurable architecture, where C1, C2, and C3 represent processing units. They are the basic units in which faults are modeled. A basic unit is called *System Item*, or just *Item*.

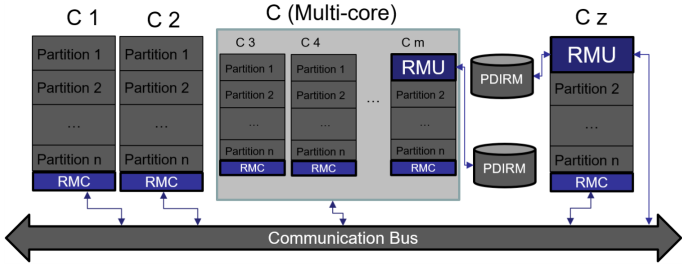

Fig. 1. Reconfigurable System Architecture

In order to perform the reconfiguration, three main components are proposed in the reconfigurable architecture:

- Resource Manager Unit (RMU): acts upon system items failure and triggers system reconfiguration according to a previously offline determined reconfiguration mapping;
- Resource Manager Client (RMC): assess reconfiguration request in relation to erroneous item failure detection by the RMU and manages each processing unit reconfiguration state. It is present on every processor on the platform throughout the system as part of our reconfiguration approach;
- PDI Resource Manager (PDIRM): contains the information about all processing unit schedules in every system state possible during successive reconfiguration and item failures.

The proposed reconfigurable architecture takes into consideration single core computers (for instance, C1 and C2 in Figure 1) and Asymmetric Multi-Processing (AMP) computers (for instance, C3 in Figure 1). For the AMP solution, every single core has an RMC to provide health monitoring and the reconfiguration execution.

The need for the RMC is to prevent the loss of a critical system function due to an erroneous RMU reconfiguration. With the absence of the RMC, the RMU would be an evident system single point of failure.

In order to decrease the reconfiguration complexity, it is assumed that all processing units in the system involved in critical functions are synchronized at each partition. In case of reconfiguration, the new system schedule is activated synchronously throughout the modules, avoiding communication mismatches.

The proposed architecture follows the basic DO-297 principles such as space and temporal isolation. The partition within each computer or core is bounded to its resource by the ARINC 653 compliant operating system. The OS guarantees a certain partition in a certain computer to be run in a predefined time slot with no preemption even though there is no process assigned to it. All the processes inside the partition, on the other hand, are subject to a preemptive policy, the rate monotonic in this study.

Memory is not a critical resource in modern computers. On the other hand, memory access time management can pose a challenge in system timing analysis. The communication bus is a time-constrained resource and should be well managed as it is shared between several processing units.

In a reconfiguration scenario, transmitting big blocks of data to be loaded in a remote module at runtime saturates the communication bus and eventually affects functionalities which were not directly affected by the triggering failure. Therefore, in the proposed system architecture, all software items (for instance, executable object code) planned to be allocated to a certain computer in any of the possible feasible reconfigurations previously determined are stored in the target memory in advance, instead of being transferred at runtime as usually proposed in earlier works [21]. In a typical ARINC 653 software this is implemented by defining different schedules for each processing unit.

The RMC manages the schedule selection. When a new reconfiguration is triggered, the RMU sends the schedule that every processing unit must be configured to. Every RMC impacted by the reconfiguration must first confirm the trigger and request the health status of the failed component directly from the client associated with it, with no RMU intervention. If the failure is not confirmed, the schedule change is aborted, and the RMC keeps the processing unit in the latest state. However, if the failure is confirmed, a new schedule is configured to be active in the next processing unit major cycle absorbing the functionalities from failed components.

The applications must be designed to tolerate the worst case in which its function will remain in failure until it is reconfigured to a new processing unit.

The RMU responsibility is then to monitor the platform overall health status. The failure modes identified in runtime are mapped to a certain transition in the reconfiguration state diagram stored in the PDIRM. The database gives the exact

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TAES.2019.2915406, IEEE Transactions on Aerospace and Electronic Systems

state to which the system must be reconfigured and keep running as expected. The new indicated configuration is sent to all involved processing units.

The communication bus is a deterministic Ethernet and a realization of the ARINC 664 part 7. The end systems include a dedicated hardware to handle A664 traffic and the network topology is set to comply with the latency requirements of each application. Annighofer et al. [22] propose an algorithm to generate aircraft data and communication network topologies, taking into account message flows and network component characteristics. The algorithm presented there could be used to complement, in terms of data communication efficiency, the work presented here.

IV. Reconfiguration Approach

In the next subsections, the proposed reconfiguration approach is detailed, by presenting the design flow, the meta-models to capture the avionics system specification, as well as, the implementation and deployment information for the feasible reconfigurations. The adopted algorithms are also described and illustrated.

A. Design Flow

Figure 2 shows the design flow of the proposed approach to system reconfiguration, which starts by modeling the platform, the application to be deployed on the platform, and the properties, as design constraints to be satisfied by any valid implementation of the specified system. The main focus of this paper is the reconfiguration (highlighted in Figure 2).

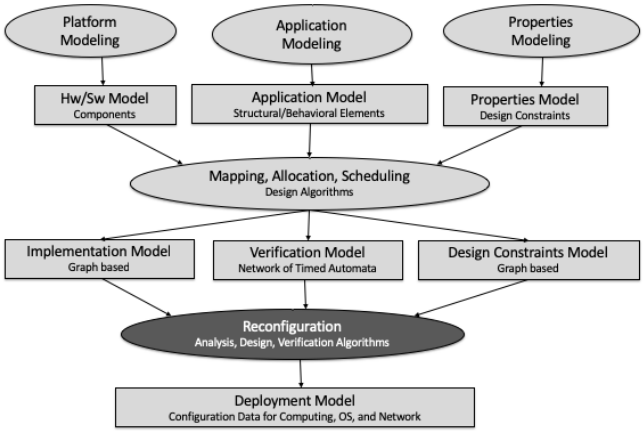

Fig. 2. Design Flow Overview

For the platform, application, and properties modeling, AADL [7] is adopted since it is already a well-studied format for the specification of avionics systems [23]. Next, it is shown how AADL resources are used in the modeling process, which includes processors with partitions, representing a virtual processor with a specific fixed time slack to perform some action, memories, buses and devices for the platform modeling; intercommunicating processes with threads inside and interconnected by means of ports for application modeling; and, property sets for the design constraints modeling.

By means of model transformations, an AADL specification is transformed into models conforming to the proposed meta-models, described in the next subsections. On these models, a mapping, allocation and scheduling algorithm is applied, which determines which software items will be mapped into each hardware item from the platform, allocated to each available partition and scheduled at specific time steps. All the information generated by the design algorithms is annotated in the implementation, verification, and design constraint models in order to be used by the reconfiguration algorithms, which produces a deployment model with the necessary data to perform the system reconfigurations at runtime.

B. Avionics System Specification

In this proposal, an avionics system is specified by means of Platform and Application models, conforming to the meta-models described in the following. The meta-models were created by using the EMF (Eclipse Modeling Framework) based modeling tool, available in the Eclipse version 4.3.7a Oxygen [24]. The AADL models were created using the OSATE, version 2.3.5, modeling tool from CMU-SEI [25].

1) Platform: consists of hardware and software components, as well as, communication buses (see Figure 3). A hardware component has one or more computers (mono or multi-cores), and each computer can have many partitions.

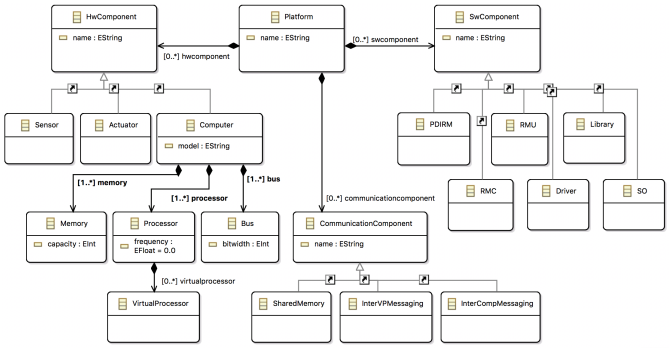

Fig. 3. Platform Meta-model

A software component can be an RMU or an RMC, which are responsible for the reconfiguration actions in the avionics system. A software component can also be an Operating System (OS), a reusable library, a driver, or the source code for the implementation of a given system function.

A communication bus can be modeled in AADL as a shared memory, when the communication occurs inside the same partition of a computer; as an inter-virtual processor messaging, when the communication is between two different partitions at the same processor; or as an interprocessor messaging, when the communication involves two different computers in the platform. Figure 4 shows an example of Platform in AADL containing four processors C1, C2, C3, and C4, each one with three, one, two, and four partitions, respectively.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TAES.2019.2915406, IEEE Transactions on Aerospace and Electronic Systems

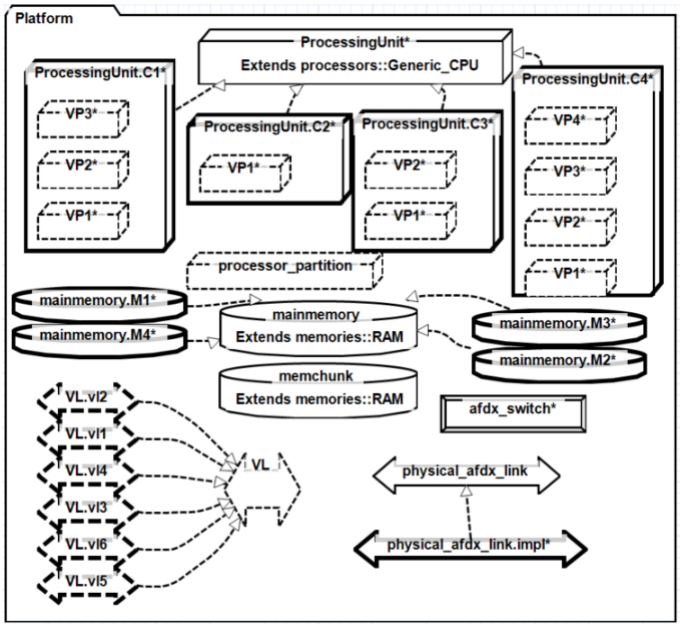

Fig. 4. A Platform in AADL

Figure 4 shows an ARINC 664 compliant bus with six virtual links to implement the communication between the computers, and memory components. Table I presents the specified properties for the processing units in the example presented in Figure 4.

TABLE I
PARTITION DURATIONS (IN MS)

C1 M. frame: 20 ms RAM: 256 Kb Flash: 1 Mb			C2 M. frame: 10 ms RAM: 256 Kb Flash: 1 Mb	C3 M. frame: 10 ms RAM: 256 Kb Flash: 1 Mb		C4 M. frame: 20 ms RAM: 256 Kb Flash: 1 Mb			
VP1	VP2	VP3	VP1	VP1	VP2	VP1	VP2	VP3	VP4
5	5	10	10	5	5	4	6	6	4

As shown in Table I, each computer has a major frame (in milliseconds), indicating how much time all partition executions take, memory capacities, and the time slot of each virtual processor (also in milliseconds). Other design properties can also be specified in the AADL model, such as the latency of the virtual links, which were set as 1 ms in this example, the size and width of the memory components, etc.

2) Application: consists of functions that will be performed by the considered application. An application function can have sub-functions and software items.

A software item can be a process, a thread, or a device, where a process is a group of threads. For each thread it is possible to have the source code of the program to be executed. These concepts in the Application meta-model allow specifying an application in a hierarchical way. Figure 5 shows an example of an application in AADL, in which there are three processes P1, P2, and P3, which have three, two, and five threads, respectively.

The data flow between the devices, processes, and threads

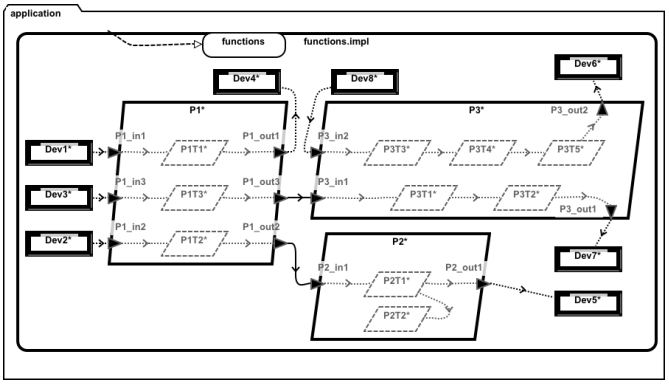

Fig. 5. An Application in AADL

are specified by means of ports and connections between them and determines the dependencies between the software items. Thus, in the example in Figure 5, the process P1 has threads P1T1, P1T2, and P1T3, which have no dependencies between them and therefore can run concurrently. Unlike the P3 process, where there are dependencies between the P3T3, P3T4, and P3T5, which requires their sequential execution.

These dependencies are captured by a Directed Acyclic Graph [12] that is called Hierarchical Dependency Graph (HDG), where the nodes represent the software items and the edges represent data flow and also control flow dependencies between the nodes. Figure 6 presents the HDG corresponding to the application in Figure 5.

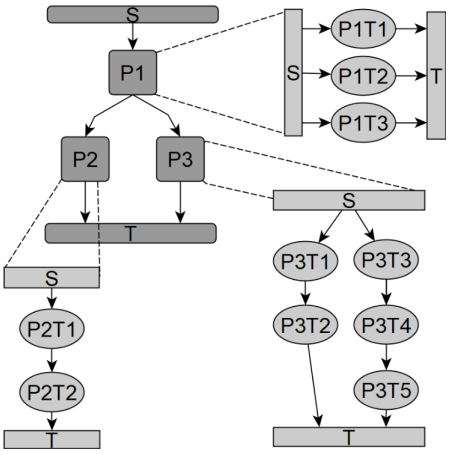

Fig. 6. Example of Hierarchical Dependency Graph

In the HDG in Figure 6, there are three nodes in the higher level representing the processes in the application and ten nodes in the lower level of the hierarchy for the threads. In the properties model, for each thread, the following is captured: the specified period, deadline, worst-case execution time (WCET), and necessary memory, given by the designer in the AADL modeling. Table II shows the specified properties for the application in Figure 5.

As shown in Table II, thread T1 of process P1 has specified

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TAES.2019.2915406, IEEE Transactions on Aerospace and Electronic Systems

TABLE II
THREADS PROPERTIES FOR APPLICATION

Prop.	P1			P2		P3				
	T1	T2	T3	T1	T2	T1	T2	T3	T4	T5
Period (ms)	20	20	20	20	20	30	30	40	40	40
Deadline (ms)	20	20	20	20	20	30	30	40	40	40
WCET (ms)	4	4	4	2	2	4	4	4	4	4
Memory (Kb)	90	50	30	40	70	90	80	60	50	95

period, deadline, WCET, and demanded memory given by 20ms, 20ms, 4 ms, and 90Kb, respectively.

3) Properties: allows capturing the design constraints specified by the designer that must be satisfied by any valid implementation for the system. A Property can represent criticality, priority, period, deadline (soft and hard), and dissimilarity characteristics of elements in the models, conforming to the proposed meta-models.

C. Implementation Modeling

The Implementation meta-model defines how to model the design decisions that are taken during the execution of the design algorithms and by the designer. It captures the mapping, allocation, and scheduling information that is produced by the design tools.

D. Design Constraints

The Design Constraint meta-model defines how to associate the properties of the system specification to the properties in the implementation model, which is generated by the design process. A Design Constraint associates a property to a given design item or multiple items. For instance, it is possible to pre-allocate a specific software item to a specific virtual processor and to specify WCET of each thread.

E. Formal Verification

In order to perform model checking of some specific properties, specified as temporal logic expressions, a network of timed automata must be generated from the system specification. The Verification meta-model defines how to model such network of timed automata as a Labeled Timed Automata (LTA) System [6], which can then be expressed as concepts introduced by the UPPAAL model checking tool [8]. An LTA System consists of Declarations and one or more LTA Templates, which represent the automata. The states are modeled as LTA locations and the state transitions as LTA transitions, which are represented by LTA edge sources and LTA edges targets. Each transition can be annotated with guard, update, selection, and synchronization expressions. The guard expression must be satisfied in order for the transition to be enabled. When an enabled transition occurs the corresponding update expression is executed, which can modify the values of some specified variables. The synchronization expression allows two automata to synchronize and the transitions at both automata are simultaneously triggered.

From the platform, application, and property models and the corresponding HDG, the network of automata is automatically generated, based on the framework for schedulability analysis proposed by David et al. [26]. According to this framework, a resource automaton for each processor partition and a task automaton for each of the process threads is instantiated.

Each one of the task automata starts in an initial state where it keeps waiting until all other tasks that it depends on finish their execution and then send a request to the resource automaton, corresponding to the processor partition in which the task was allocated. The resource automaton models some specified scheduling policy, that can be Earliest Deadline First (EDF), Fixed Priority Scheduling (FPS), or First In First Out (FIFO), by managing a tasks queue. When the task execution is concluded, the corresponding resource automaton notifies it by a finished signal.

At each task automaton, there is a transition from the Ready state to an Error state when the elapsed time is greater than the task deadline, which means that the specified design constraint was not satisfied. Thus, to perform the schedulability analysis of the system, the following temporal logic expression is checked on the generated network of automata, using the UPPAAL model checker: `A[] for all (i : t_id) not Task(i).Error`. This expression means that, for all possible execution paths, no task automaton reaches the Error state.

When the model checker concludes that this property is valid, all the specified deadline constraints are satisfied, i. e., it has found a feasible mapping, allocation, and scheduling reconfiguration for the application on the given platform.

F. Reconfiguration States Diagram

The Reconfiguration States Diagram (RSD) represents the possible feasible reconfigurations as states, and the state transitions as the valid transformation from a currently feasible reconfiguration to a next feasible reconfiguration, which can tolerate some fault in each system component, indicated as a label at the corresponding state transition. Figure 7 shows an example of a reconfiguration states diagram.

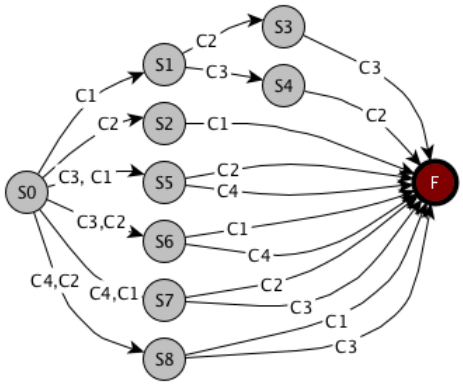

Fig. 7. An Example of Reconfiguration States Diagram

At the initial state S0, the system is operating normally, according to an initial mapping, allocation, and scheduling

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TAES.2019.2915406, IEEE Transactions on Aerospace and Electronic Systems

algorithm, which considers the specified properties of the system. The state transition from S0 to S1, labeled C1, indicates that when the component C1 fails, the system has a feasible reconfiguration, represented by S1 state, which tolerates the fault. When at state S1 and component C2 fails, the system can be reconfigured according to the state S3. The system has no feasible reconfiguration from state S3 when component C3 fails, which is represented by state transition from S3 to F (Fail state), labeled with C3. The transitions from S0 to S2 and from S2 to F indicate that after component C2 fails a fault in C1 cannot be tolerated by the system. Note that the appearance of two component failures on one transition is a way of abbreviating the graph. Failures happen one at a time and dealt through one reconfiguration, as stated in Section I.

G. Deployment Model

The Deployment meta-model defines how to capture the necessary information to perform the reconfigurations of the system. Each configuration determines how each design item will be deployed, associating elements of the system specification to elements of the platform, as well as, all the necessary design information.

H. Design Automation Algorithms

The design automation process for the proposed reconfiguration approach includes algorithms to build the Reconfiguration Diagram, to perform mapping, allocation, and scheduling, to generate the network of timed automata for the schedulability analysis, and to generate the Deployment Model for each given reconfiguration. The next subsections describe each of these algorithms.

1) Mapping, Allocation, and Scheduling: algorithms that build the implementation model and the deployed model given a current platform, system, and properties model as follows.

First, a list of tasks with their properties, for example the period, the deadline, and the WCET are created. Then, the hierarchical dependencies graph is generated from the AADL application model. A list of available computers and their virtual processors with corresponding properties are created from the platform and properties models.

By means of a depth first traversal in the hierarchical dependency graph, starting from the begin node, each task is visited and mapped into an available computer, and allocated into one of their virtual processors, that should satisfy the specified design constraints. This step produces an allocation matrix, containing the information to be used by the algorithm for the generation of the network of automata. It also instantiates a corresponding deployment model to be used when building the reconfiguration states diagram, described in the next subsection. Table III shows the allocation matrix produced from the platform and the application models, shown in Figures 4 and 5, respectively, and platform and application properties listed in Tables I and II, respectively.

As shown in Table III, when no computer fails, threads T1, T2, T3 of process P1, threads T1 and T2 of process P2, and thread T3 of process P2 are mapped to the computer C1

TABLE III
ALLOCATION MATRIX (PARTIAL)

	C1			**C2**	**C3**		**C4**			
VPs	VP1	VP2	VP3	VP1	VP1	VP2	VP1	VP2	VP3	VP4
None fails	P1T1	P1T2	P1T3 P2T1 P2T2 P3T3	P3T1 P3T2 P3T4 P3T5	-	-	-	-	-	-
C1 fails	X	X	X	P1T1 P1T2 P3T1 P3T2 P3T4	P1T3 P2T1 P2T2 P3T5	P3T3	-	-	-	-
C2 fails	P1T1	P1T2	P1T3 P2T1 P2T2 P3T3	X	P3T2 P3T5	P3T1 P3T4	-	-	-	-
C1, C2 fail	X	X	X	X	P2T1 P2T2 P3T5	P1T3 P3T3 P3T4	P1T1	P1T2	P3T2	P3T1
C1, C3 fail	X	X	X	P1T1 P1T2 P3T1 P3T2 P3T4	X	X	P1T3	P3T5	P2T1 P2T2	P3T3
C2, C1 fail	X	X	X ***P3T3***	X	P3T2 P3T5	P3T1 P3T4	P1T1	P1T2	P1T3	P2T1 P2T2
...	...	...	...	...	...	...	...	...	...	...

and allocated on its virtual processors VP1, VP2, VP3, which are the first ones available with enough time slot and memory resources to execute the corresponding threads. When C1 fails, the threads from C1 can migrate to C2 and C3, which has available virtual processors with enough time and resources. After that, if C2 fails, its threads can further migrate to C3 and C4. However, when C3 fails, the threads P3T3, P3T4, and P3T5 in VP2 of C3 cannot be mapped into any computer, and so there is no feasible reconfiguration that could tolerate this fault in C3 at this point. Figure 7 shows the Reconfigurable States Diagram that will be generated based on this allocation matrix.

2) Build the Reconfiguration States Diagram: represents all possible feasible reconfigurations, starting from the initial valid implementation of a system and going to each possible feasible reconfiguration that can mask the fault in each system component. The corresponding algorithm is shown in Algorithm 1:

- At line 1, the generation of the initial deployment model consists of, for each element from the application model, a mapping to an element from the platform model determined by the design tool or by the system designer. This step will produce the initial state $S0$ in the Reconfiguration States Diagram.
- At line 2, the algorithm starts from a source state (Ss) as initial state $S0$.
- The entire algorithm (lines 3-38) will repeat until all existing states are marked, and this occurs when all the created states were considered by the algorithm.
- At each iteration of the Algorithm 1, the possible failure of some computers Ci is considered (lines 5-32) and for each possible feasible reconfiguration (*schedulable*, according to the model checking verification), a new target state (St) is created and a transition from Ss to St is created (lines 20-25), or

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TAES.2019.2915406, IEEE Transactions on Aerospace and Electronic Systems

Data: Platform, Application, and Properties models
Result: Reconfiguration States Diagram
1 $S0$ = generate initial Deployment;
2 $Ss = S0$;
3 **repeat**
4 **repeat**
5 **for** *each* Ci *in the current Deployment and* Ci *failed* **do**
6 **for** *each* $SwItem\_p$ *not yet mapped* **do**
7 **for** *each* Cj *in the current Deployment and* $Cj \neq Ci$ **do**
8 **if** *(is_compatible(*Cj*)* **or** *has_free_partition(*Cj*))* **then**
9 $SwItem_p$ is mapped to Cj;
10 **end**
11 **if** *there is* $SwItem_q$ *in* Cj *which is not critical or it was affected by the fault at* Ci **then**
12 unmap $SwItem_q$ from Cj;
13 map $SwItem_p$ to Cj;
14 **end**
15 **end**
16 **end**
17 **if** *there is no unmapped critical* $SwItem_p$ **then**
18 build Verification model based on current mappings;
19 perform Model Checking on Verification model;
20 **if** *schedulable* **then**
21 St = generate new Deployment model from current mappings;
22 create transition from Ss to St with label Ci;
23 **if** *there is unmapped non critical* $SwItem_p$ **then**
24 Mark St as $Degraded$
25 $Ss = St$;
26 **else**
27 create transition from Ss to $Serror$ with label Ci;
28 **end**
29 **else**
30 create transition from Ss to $Serror$ with label Ci;
31 **end**
32 **end**
33 **until** *no new state* St *was created*;
34 mark Ss as done;
35 **if** *(there is any non-marked state* Snm*)* **then**
36 $Ss = Snm$;
37 **end**
38 **until** *all states are marked*;

Algorithm 1: Build Reconfiguration States Diagram

- If there is no feasible reconfiguration a transition from Ss to an error state ($Serror$) is created (lines 27 and 30).
- A new Deployment model is built for each newly created target state (line 21), according to the current mappings.
- At lines 8-10, a $SwItem_p$ is mapped into Cj if it is compatible with Cj, i.e., if there is no $SwItem_q$ already mapped to Cj, which would conflict with $SwItem_p$. The conflict between software items can be specified in the Properties model by the designer. A $SwItem_p$ can also be mapped into Cj if there is a free partition available in Cj.

This algorithm produces a Reconfiguration States Diagram, where each state represents a feasible reconfiguration and each transition, labeled Ci, from a state Ss to St indicates that if the system is currently configured as determined by the Deployment model associated to state Ss and the computer Ci fails then the system tolerates the fault, and goes to a new configuration given by the Deployment model associated to state St. In the case where St is at a failure state, the entire system fails since the fault could not be tolerated.

V. CASE STUDY

In order to evaluate the proposed approach for avionics systems development, an illustrative case study from the avionics domain will be used. A top-down approach is chosen as recommended by [27], defining at first the system functions and subsystem functions (Table IV). The system functions are the highest-level definitions in a system and specify its basic functionalities. From this definition, the break down is performed for the system realization.

In order to determine the criticality and therefore the certification efforts of the bottom level system items, the Function

TABLE IV
FUNCTIONS

Flight Control	Navigation
Fuel Management	Guidance
Flight Control	Route Control
	Provide Map
HMI	Provide Charts
Display Symbology	
System Control	System Monitoring
Flight Control Inputs	Fuel Monitoring
	Enginer Monitoring
Auto Pilot	

Hazard Analysis (FHA) is performed. It identifies top-level failure conditions, effect, and their severity. The loss of, or undetected erroneous flight control, for instance, can cause the loss of the aircraft giving the severity classification as catastrophic. On the other hand, the loss of the Map provider functionality causes a slight increase in crew workload which gives a Minor in the classification. According to SAE guidelines [28], all the developed functions related to catastrophic events must comply to the highest design assurance level (A), while the ones related to minor events must comply at least to the level D which is the second lowest in a scale from E to A.

In parallel the top-level system architecture with the corresponding software item can be created. At this point the architecture is still independent of the platform and should be tied only to the top-level system functions. Figure 8 illustrates what is supposed to be the created items dependencies which can generate the HDG described in Section IV-B2. The real dependency diagram was created in AADL using annex ARINC653 and is not included in this paper due to its complexity and size. The complete model can be found in the project repository in Github[1]. Each node identified in Table V is modeled in AADL as a thread which includes besides the item interface, the descriptions of its properties. For the dependency diagram, a system implementation model was created including all the threads encapsulated in processes and their connections.

Figure 8 also partially shows how the system functions are realized by the software items. The *flight control* for instance is realized by *FuelConsuptionMgr*, *Va_Control* and *Vz_Control*. These two components were based on an Open-Source Avionics and Control Engineering case-study [29]. The other items and their properties were created according to previous experiences of the authors and interactions in the industry.

From the presented relation, together with the FHA, it is possible to infer the item design assurance level (IDAL) as it can be seen in Table V. For the flight control subsystem, it is possible to see the assurance levels in the table as nodes (column 1) 12, 13, 14, 15, 16 and 17.

In the AADL model, the IDAL becomes an item property as described in Section IV-B3. This particular property carries

[1]https://github.com/aafontoura/reconfigurationAvionicsCaseStudy.git

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TAES.2019.2915406, IEEE Transactions on Aerospace and Electronic Systems

TABLE V
SOFTWARE ITEMS PROPERTIES FOR CASE STUDY

Node #	Software Item	Per.	WCET	IDAL	Redund.
0	SystemInputHandler	20	5	A	Simple
1	MapServer	40	15	D	None
2	Charts	640	12	D	None
3	RouteCtrl	40	2	C	Simple
4	EngineMon	20	1	B	Simple
5	Guidance	20	4	B	Simple
6	FuelMon	40	1	B	Simple
7	AP Monitor	20	2	A	Simple
8	AP	20	5	C	Simple
9	FlightStickHandler	20	1	A	Voter
10	DisplayMgr	20	6	A	Simple
11	Altitude_Hold	20	1	A	Simple
12	FuelConsumptionMgr	80	8	A	None
13, 14	Vz Control	20	1	A	Dissimilar, Voter
15, 16	Va Control	20	1	A	Dissimilar, Voter
17	Flight Control Mgr	20	2	A	Simple
18,19	DisplayX_Server	30	10	B	Simple
20,21	ActuatorXControl	10	0.5	A	Simple

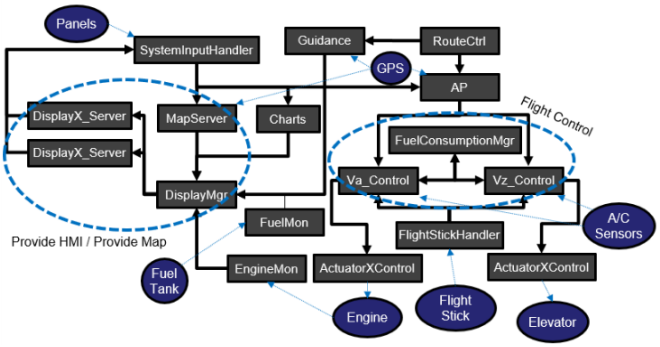

Fig. 8. Initial System Dependencies

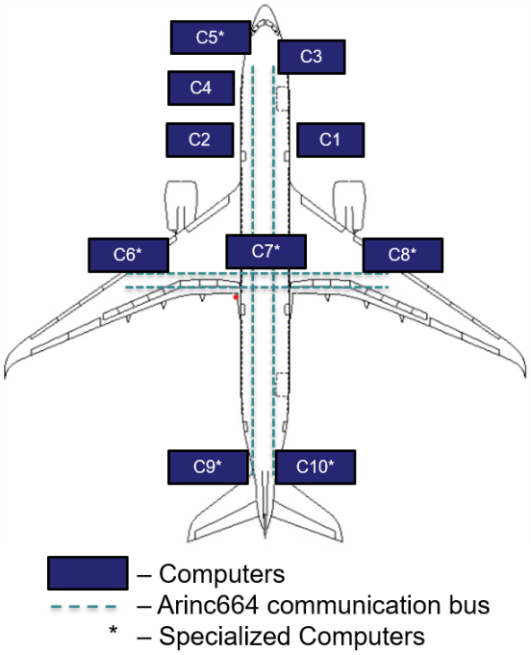

Fig. 9. Allocation of functions to nodes within the platform

an important design constraint for the allocation algorithm mentioned in Sections IV-D and IV-H. The computer of the platform that will accommodate a set of items must be developed following the assurance level of the most critical item chosen to be allocated on that specific computer. Which means if a certain computer is developed to IDAL C, it will be ineligible to accommodate software items developed to IDAL A. In addition, partitions can only hold items with the same design assurance level. The *Provide Map* function demands a lower functional design assurance level (FDAL) therefore a lower IDAL as shown in Table V (node 01 in the first column).

Table V also presents other important item properties that need to be taken into consideration by the allocation algorithm, i.e., the Worst case execution time (WCET), the period, the IDAL and the general redundancy policy. The *flight control* items, *Va_Control* and *Vz_Control* are to be implemented in a voter system, which brings the constraint of having two nodes for each of them. Dissimilarity is also a requirement due to the criticality of their system function and implies that the pair must run in different types of computers. The values for WCET and period are synthetic but consistent with values used in real avionics systems projects, which cannot be explicitly mentioned here due to non-disclosure agreements.

Figure 9 presents the platform node hardware in which all the previously described software items will be allocated. Certain computers such as the ones placed in the back of the aircraft are specialized, being able to interface with actuators (e.g. Rudder control) and sensors besides the ability to communicate through the airplane data bus. The different types of computer specializations also are taken into consideration during the allocation process.

The computers will accommodate processing software items, such as the *FuelConsuptionMgr*, which is a software item responsible for running algorithms to enhance the fuel consumption performance by the flight control. The specialized computer C5, for instance, is attached to the main displays in the cockpit which have rendering engines and are able to present critical information for the pilot. Therefore it is eligible to accommodate the item *DisplayX_Server* which interprets commands from other computers and translates them into drawing commands for the rendering engine, generating the image. Such restrictions are evaluated during the allocation algorithm at the compatibility check (line 8 of Algorithm 1). Table VI specifies the computers properties of the platform shown in Figure 9. The RMU and RMCs are embedded within the computers in this platform as presented in Figure 1.

As previously presented in Section IV, the HDG is generated from the dependency diagram created in AADL, shown in Figure 10. Here, the number on each node refers to the node # in Table V. The HDG is used as one of the inputs for

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TAES.2019.2915406, IEEE Transactions on Aerospace and Electronic Systems

TABLE VI
CASE STUDY PLATFORM PROPERTIES

C1 MF: 20 ms				C2 MF: 10 ms		C3 MF: 10 ms			C4 MF: 20 ms			
VP1	VP2	VP3	VP4	VP1	VP2	VP1	VP2	VP3	VP1	VP2	VP3	VP4
5	5	5	5	7	3	5	4	1	4	6	8	2

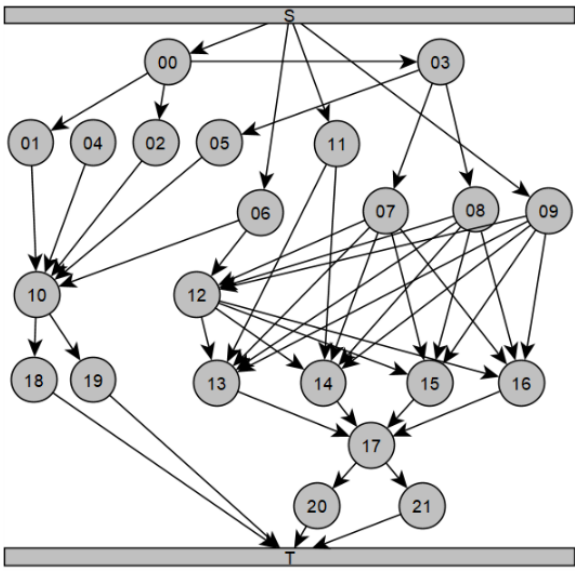

Fig. 10. HDG for case study

the UPPAAL model. Figure 11 shows the obtained diagram for the case study, after the execution of Algorithm 1, where the timing constraints for the reconfiguration associated with each state in the RSD is verified by applying the UPPAAL model checker. Each node is a location in the UPPAAL where a reconfiguration state is represented. The transitions are the failure triggers and here they represent a complete failure of the computer.

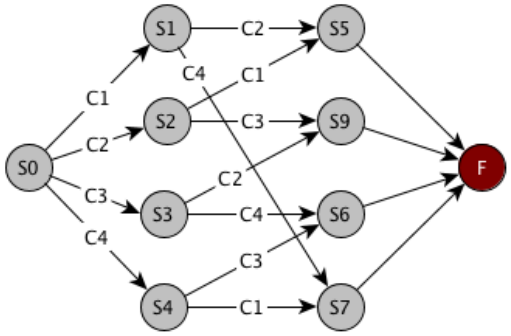

Fig. 11. RSD for case study

To verify if the initial allocation, corresponding to state S0 in the RSD, was schedulable, UPPAAL consumed 13,503 seconds (i.e., 225.05 minutes) running in Mac OS X system on an Intel i7 2,2GHz with 16Gb RAM. We used the option *over approximation*, available in UPPAAL (by means of parameter `-A` for the command `verifyta`), which reduces the number of explored states by applying a convex-hull based approximation technique [26]. Even so, in this case, UPPAAL reached more than 62 million states.

VI. CONCLUSION

In this paper, a reconfiguration approach to deal with fault management and its associated timing analysis with model checking is presented. This checking guarantees that all foreseen situations are evaluated in determining that the design time timing constraints are effectively satisfied.

The proposed approach was illustrated using a synthetic example to explain its algorithms. Then it was further applied in an avionic system case study to show that reasonable problem sizes in terms of the number of nodes, dependencies, criticality constraints, and software/hardware mappings can be dealt with. The proof of concept shows that the proposed methodology provides a feasible design flow for avionics systems to be further evaluated in industrial settings.

The state explosion problem in the model checker execution was an expected concern, but it can be minimized considering the hierarchical and modular nature of AADL-modeled applications. Applying model checking in a modular way on each process separately, then using the HDG to perform a global analysis based on the analysis of the processes is a feasible strategy to handle this issue.

A further source of complexity in the analysis is the order in which the components can fail. Currently, the quantity of possible combinations is factorial in the number of components. So, it is important to develop heuristics to minimize the size of this design space in order to be able to deal with larger applications. Regarding this aspect, the hierarchical and modular nature of the AADL models and application-specific or domain-specific knowledge can be explored.

An interesting subject to be investigated in future works is the development of algorithms that improve the current solution by a) considering optimization techniques and avoiding the local minima, and b) making the timing analysis more efficient, leading to faster analyses of each strategy.

ACKNOWLEDGEMENT

The third author was supported by the Swedish Governmental Agency for Innovation Systems under grant number NFFP7-04890. The last author is partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TAES.2019.2915406, IEEE Transactions on Aerospace and Electronic Systems

REFERENCES

[1] L. Jözwiak and N. Nedjah, "Modern architectures for embedded reconfigurable system - a survey," *Journal of Circuits, Systems, and Computers*, vol. 18, no. 2, pp. 209–254, 2009.

[2] A. Löfwenmark and S. Nadjm-Tehrani, "Fault and timing analysis in critical multi-core systems: A survey with an avionics perspective," *Journal of Systems Architecture*, vol. 87, pp. 1 – 11, 2018.

[3] H. Jiang, "Key findings on airplane economic life," *Boing white paper*, March 2013. [Online]. Available: https://aviation.report/view-resource.aspx?id=11

[4] C. Balis, D. Berenson, and A. Jovovic, "Top 5 trends in military aviation," *The European Security and Defence Union*, June 2013.

[5] Airforce Technology, "Embraer completes first batch deliveries of F-5EM fighter to Brazil," accessed 2018-12-02. [Online]. Available: https://www.airforce-technology.com/news/newsembraer-upgraded-f-5em-tiger-fighter-brazil/

[6] C. Baier and J. P. Katoen, *Principles of model checking*. Cambridge, MA, USA: MIT Press, 2008.

[7] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. New York, NY, USA: Addison-Wesley Professional, 2012.

[8] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, p. 1, March 1997.

[9] W. Housseyni, O. Mosbahi, M. Khalgui, Z. Li, and L. Yin, "Multiagent architecture for distributed adaptive scheduling of reconfigurable real-time tasks with energy harvesting constraints," *IEEE Access*, vol. 6, pp. 2068–2084, 2018.

[10] G. Gigante and D. Pascarella, "Formal methods in avionic software certification: The DO-178C perspective," in *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, T. Margaria and B. Steffen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 205–215.

[11] Y. Cui, J. Shi, and Z. Wang, "Backward reconfiguration management for modular avionic reconfigurable systems," *IEEE Systems Journal*, vol. 12, no. 1, pp. 137–148, March 2018.

[12] J. Kleinberg and E. Tardos, *Algorithm Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[13] Q. Zhou, T. Gu, R. Hong, and S. Wang, "An AADL-based design for dynamic reconfiguration of DIMA," in *IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC)*, October 2013, pp. 4C1–1–4C1–8.

[14] G. Fohler, G. Gala, D. G. Prez, and C. Pagetti, "Evaluation of dreams resource management solutions on a mixed-critical demonstrator," *9th European Congress on Embedded Real Time Software and Systems (ERTS)*, vol. 12, no. 1, pp. 1–10, January 2018.

[15] R. B. Atitallah, V. Viswanathan, N. Belanger, and J. L. Dekeyser, "FPGA-Centric design process for avionic simulation and test," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 54, no. 3, pp. 1047–1065, June 2018.

[16] G. Montano and J. McDermid, "Human involvement in dynamic reconfiguration of Integrated Modular Avionics," *AIAA/IEEE Digital Avionics Systems Conference*, no. 978, pp. 1–13, 2008.

[17] S. Porcarelli, M. Castaldi, F. Di Giandomenico, A. Bondavalli, and P. Inverardi, "A Framework for Reconfiguration-Based Fault-Tolerance in Distributed Systems," in *Architecting Dependable Systems*, R. D. Lemos, C. Gacek, and A. Romanovsky, Eds. Springer-Verlag, 2004, pp. 167–190.

[18] P. Hollow, J. McDermid, and M. Nicholson, "Approaches to certification of reconfigurable IMA systems," *the International Council on Systems*, pp. 1–8, 2000.

[19] B. Annighofer and F. Thielecke, "Multi-objective mapping optimization for distributed Integrated Modular Avionics," *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, pp. 1–13, 2012.

[20] RTCA, "DO-178C - software considerations in airborne systems and equipment certification," RTCA SC-205, Standard, January 2012.

[21] P. Bieber, E. Noulard, C. Pagetti, T. Planche, and F. Vialard, "Preliminary design of future reconfigurable IMA platforms," *SIGBED Rev.*, vol. 6, no. 3, pp. 7:1–7:5, October 2009.

[22] B. Annighoefer, C. Reif, and F. Thieleck, "Network topology optimization for distributed integrated modular avionics," *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, pp. 4A11–4A112, 2014.

[23] Q. Zhang, S. Wang, and B. Liu, "Approach for integrated modular avionics reconfiguration modelling and reliability analysis based on AADL," *IET Software*, vol. 10, no. 1, pp. 18–25, 2016.

[24] Eclipse, "EMF: Eclipse Modeling Framework," accessed 2018-12-02. [Online]. Available: https://www.eclipse.org/modeling/emf/

[25] OSATE, "OSATE 2: Open Source AADL Tool Environment," accessed 2018-12-02. [Online]. Available: https://wiki.sei.cmu.edu/aadl/

[26] A. David, J. Rasmussen, K. Larsen, and A. Skou, "Model-based framework for schedulability analysis using UPPAAL 4.1," pp. 1–32, November 2009.

[27] L. Xiaoxun, Z. Yuanzhen, F. Yichen, and S. Duo, "A comparison of SAE ARP 4754A and ARP 4754," *2nd International Symposium on Aircraft Airworthiness*, vol. 17, pp. 400 – 406, 2011.

[28] SAE, "ARP4761 - guideline and method for conducting the safety assessment process on civil airborne systems and equipment," SAE Aerospace, Standard, December 1996.

[29] C. Pagetti, D. Saussi, R. Gratia, E. Noulard, and P. Siron, "The ROSACE case study: From simulink specification to multi/many-core execution," in *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014, pp. 309–318.

Antonio Augusto da Fontoura has a Bachelor degree in Computer Engineer by UFRGS, Brazil (2012). Currently he is a MSc student in Computer Science at the Graduate Program on Computer Science at UFRGS, Brazil. Working for years in the aerospace industry, his research interests are avionics systems, formal methods and real-time embedded systems.

Francisco Assis Moreira do Nascimento has a MSc in Computer Science by UFRGS, Brazil (1992) and Bachelor degree in Computer Science by Federal University of Paraíba, João Pessoa, Brazil (1988). Since 2007 he is CTO at infisc.com.br and since 2001 he holds an assistant professor position at faccat.br, Brazil. His main research areas include: Distributed Real Time Embedded Systems, Avionics Systems, Formal Methods for Electronic Design Automation, and Computer Architecture.

Simin Nadjm-Tehrani (M00) received the B.Sc.(with honors) degree from Manchester University, Manchester, U.K., and the Ph.D. degree from Linköping University (LiU), Linköping, Sweden,in 1994. Prior to her Ph.D. studies, she was, for six years, with international companies Deloittes and PriceWaterhouse in the early 1980s. In 2006−2008, she was a Full Professor of dependable real-time systems with the University of Luxembourg, Luxembourg City, Luxembourg. Since 2000, she has led the Real-Time Systems Laboratory, Department of Computer and Information Science, LiU, involving around nine Ph.D. students and postdoctoral researchers. She is currently a Full Professor with the Department of Computer and Information Science, LiU. Her research interests include dependable distributed systems and networks with resource constraints, particularly the analysis of safety and fault tolerance, reliable communication, and measuring availability in the presence of failures, attacks, and overloads.

Edison Pignaton de Freitas has a PhD in Computer Science and Engineering by Halmstad University in Sweden (2011), MSc in Computer Science by UFRGS, Brazil (2007) and Bch degree in Computer Engineering by Military Institute of Engineering, Brazil (2003). Currently he is associate professor at UFRGS, acting in the Graduate Programs in Electrical Engineering and in Computer Science, focusing the following research areas: Real-Time Systems, Avionics Systems, Industry Automation, Computer Networks and Unmanned Aerial Vehicles.