UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ANTONIO AUGUSTO DA FONTOURA

Design Automation for Avionic Reconfiguration Schemes and Schedulability Analysis

Thesis presented in partial fulfillment of the requirements for the degree of Master of Computer Science

Advisor: Prof. Dr. Edison Pignaton de Freitas

Porto Alegre
May 2022

CIP — CATALOGING-IN-PUBLICATION

da Fontoura, Antonio Augusto

Design Automation for Avionic Reconfiguration Schemes and Schedulability Analysis / Antonio Augusto da Fontoura. – Porto Alegre: PPGC da UFRGS, 2022.

81 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2022. Advisor: Edison Pignaton de Freitas.

1. Reconfiguration. 2. Avionic Systems. 3. Distributed Real Time Embedded Systems. 4. Schedulability Analysis. 5. Model Checking. 6. Design Automation. I. Pignaton de Freitas, Edison. II. Título.

“Luck Is What Happens When Preparation Meets Opportunity.”

— SENECA

AGRADECIMENTOS

Meus sinceros agradecimentos a todos que de alguma forma contribuíram para o desenvolvimento deste trabalho e que contribuíram com o meu crescimento profissional e pessoal durante esses anos dedicados ao mestrado.

Ao meu orientador, Prof. Dr. Edison Pignaton de Freitas, pela oportunidade e pela confiança depositada em mim. Agradeço pela disponibilidade em responder meus questionamentos, pela paciência, pela troca de experiências e a dedicação em ensinar de forma clara e didática contribuíram muito para meu aprendizado e crescimento.

Aos coatores do artigo previamente publicado, Francisco Assis Moreira do Nascimento e Simin Nadjm-Tehrani, que trouxeram valiosas contribuições ao trabalho.

À universidade por fornecer meios para que alunos que já estão no mercado de trabalho possam realizar seus estudos.

À minha família, em especial à minha esposa, por me apoiar e me incentivar sempre.

ABSTRACT

Reconfigurable avionics systems can tolerate faults by moving functionalities from failed components to another available system component. This work 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. Scheduling analysis 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. Finally a design automation framework proof-of-concept is implemented and a thoroughly analysis on different algorithms is performed to demonstrate its functionality and flexibility.

Keywords: Reconfiguration. Avionic Systems. Distributed Real Time Embedded Systems. Schedulability Analysis. Model Checking. Design Automation.

Automação de Projeto para Esquemas de Reconfiguração de Aviônicos e Análise de Escalonabilidade

RESUMO

Os sistemas aviônicos reconfiguráveis podem tolerar falhas movendo funcionalidades de componentes com falha para outro componente de sistema disponível. Este trabalho propõe uma arquitetura reconfigurável distribuída para migração de aplicativos de módulos com falha para módulos em funcionamento. Os estados viáveis de reconfiguração do sistema são determinados de forma prévia, ainda na fase de design, para fornecer a configuração necessária nas situações previstas. A análise de escalonabilidade é usada para determinar se os estados provenientes das novas configurações são viáveis avaliando propriedades temporais específicas. Um estudo de caso é usado para mostrar a aplicação da abordagem apresentada como prova de conceito. Por fim, uma prova de conceito do framework de automação de design é implementada e uma análise com diferentes algoritmos é realizada para demonstrar sua funcionalidade e flexibilidade.

Palavras-chave: Reconfiguração, Systemas Aviônicos, Systemas Embarcados de Tempo Real Distribuídos, Análise de Escalonabilidade, Model Checking, Automação de Design.

LIST OF ABBREVIATIONS AND ACRONYMS

AADL	Architecture Analysis and Design Language
AMP	Asymmetric Multi-Processing
APEX	Application Executive
ARINC	Aeronautical Radio, Incorporated
DAL	Design assurance level
EDF	Earliest Deadline First
EMF	Eclipse Modeling Framework
FAA	Federal Aviation Administration
FDAL	Function design assurance level
FHA	Function Hazard Analysis
FIFO	First in First Out
FPS	Fixed Priority Scheduling
GRM	Global Resource Manager
HDG	Hierarchical Dependency Graph
HMI	Human Machine Interface
IDAL	Item design assurance level
IMA	Integrated Modular Avionics
KPI	Key performance indicators
LRM	Local resource manager
LTA	Labeled Timed Automata
MRO	Maintenance, Repair and Overhaul
PDI	Parameter Data Item
PDIRM	PDI Resource Manager
PPS	Priority preemptive scheduling

QoS	Quality of Service
RMC	Resource Manager Client
RMU	Resource Manager Unit
RSD	Reconfiguration States Diagram
RTOS	Real-Time Operating System
SAE	Society of Automotive Engineers
SLOC	Software lines of code
WCET	Worst-case execution time
OS	Operating System

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Reconfiguration of distributed real-time embedded systems consists of changing or modifying subsystems and/or subsystem configurations to better serve a certain purpose (JÖZWIAK; NEDJAH, 2009). 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 that 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. (LÖFWENMARK; NADJM-TEHRANI, 2018) 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 from the end of the 20th century to the present 21st-century (JIANG, 2013) and has now reached stability. Additionally, the Maintenance, Repair, and Overhaul (MRO) market is expected to produce a strong future demand as worldwide military Air Forces decide to upgrade legacy aircraft rather than procuring new platforms (BALIS; BERENSON; JOVOVIC, 2013), 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 (Airforce Technology, 2009).

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 offline 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.

Having a successful transition to reconfigured state due to a failure does not guarantee the feasibility of such a state. Therefore is clear the need for final verification to assess the correctness of the real-time systems set to perform the desired function. For that purpose, the schedulability analysis is used to check if every task in the system will meet its timing constraints.

Different approaches can be used to perform that verification. Model checking (BAIER; KATOEN, 2008) can be used 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) (FEILER; GLUCH, 2012), one of the proposed approaches includes the creation of a network of automata (LARSEN; PETTERSSON; YI, 1997), representing the timing aspects of an avionics system, 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. Alternatively, the scheduling simulation approach is also used by applying the scheduling algorithms during a period to compute the schedule of the system (SINGHOFF et al., 2009).

The schedulability analysis ensures the predictability of the system after each reconfiguration and facilitates airworthiness approval by the certification authorities.

Taking into consideration that systems are getting exponentially more complex every year (AVSI, 2009b). During the 1980s a typical airplane project had less than 1 million software lines of code (SLOC), while from the 2000s onwards that number exploded. For instance, the reported SLOC for the F35 program was around 24 million (AVSI, 2009a). This scenario urges for more abstractions and design automation during development. Software applications engineers should be focused on their scope, as they are not able to cope with the always increasing complexity of the underlying computer

systems and their design constraints. During this work, the proposed design automation is implemented in a form of a framework proof-of-concept.

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 to find suitable configurations that satisfy timing requirements and provide the highest degree of fault tolerance in the considered space;
- Verification through schedulability analysis;
- Design automation framework proof-of-concept.

The structure of this work is as follows. Chapter 2 presents related work and basic concepts. Chapter 3 the system reconfiguration approach and its underlying system architecture. Additionally, a representative avionics case study is presented, including a summary of how the design constraints are derived and preliminary reconfiguration results using model checking as the schedulability analysis method. Chapter 4 presents a detailed description of the proposed automation framework. Chapter 5 provides the result analysis from the framework output considering the proposed case study. The conclusions are reported in Chapter 6, providing also directions for future work.

2 CONCEPTS AND RELATED WORKS

2.1 Concepts

2.1.1 Schedulability Analysis

The schedulability analysis is a process to evaluate if the time requirements of a real-time system are met. It checks if the tasks execution times and periodicity are not violated in run-time.

Schedulability test and the theory foundations were introduced in the 1970s (LIU; LAYLAND, 1973), and it brings several ways to perform the actual analyses, such as feasibility test and scheduling simulations.

2.1.2 Formal verification and model checking

Formal verification means having a mathematical model of a system and a method of proof to verify that the specified proprieties are satisfied (MCMILLAN, 1993). It is a form of design verification and its goal is to avoid design revision and run-time failure by identifying errors early in the design process.

Model checking is a subset of the formal verification. in this approach the system model is expressed in a finite state machine, and specifications are written in a specialized language call a propositional temporal logic. An efficient search procedure is used to determine automatically if the specification are satisfied by the transition system. It can be used for instance to perform schedulability tests of real-time systems.

2.1.3 Mixed Criticality systems

A Mixed-Criticality system is a system that can run different functions from different criticality levels, safety-critical or non-critical, in the same platform. For instance, a brake system for cars and a media center for audio and navigation. One is much more important (critical) than the other, however, the driver expects both of them to be fully functional in his vehicle.

2.1.4 Integrated Modular Avionics and ARINC653

Integrated Modular Avionics (IMA) is a proposed architecture to simplify the development and certification efforts for the software of avionics systems. The first reported use of such an approach dates from the early 1990s (PRISAZNUK, 1992), first introduced Boeing 777 project (MORGAN, 1991), and it is widely used in the aerospace industry, for both, commercial and military aircraft.

The ARINC653 standard brings the software specification for space and time partitioning. It was first adopted in 1997 and rapidly accepted by the main players in the industry (PRISAZNUK, 2008). It became the foundation to build mixed-criticality applications on top of the same processing unit by providing a common basis for testing and qualifying each software item, besides specifying a standard interface called APplication EXecutive (APEX), which decouples the operating systems to the applications.

In a system that applies ARINC653 and IMA approaches, the concept of partition is introduced. A partition provides an isolated environment for software applications to run. It contains its own memory space which can not be accessed directly by a different partition. In addition, it has a dedicated time slot, so that any application running in another partition overruns, it won't affect other software applications time constraint.

2.1.5 Failure Conditions and Severity Classification

A condition affecting either the airplane or its occupants, which is caused by one or more failures or errors (Federal Aviation Administration, 2011).

As defined by Advisory Circular 23.1309-1E from the Federal Aviation Administration (FAA), failure conditions may be classified according to their severity as follows:

- **No safety effect**: Failure conditions that would not affect safety.
- **Minor**: Failure conditions that would not significantly reduce airplane safety and involve crew actions that are within their capabilities. Some physical discomfort to passengers or cabin crew
- **Major**: Failure conditions that would reduce the capability of the airplane or the ability of the crew to cope with adverse operating conditions to the extent that there would be a significant reduction in safety margins or functional capabilities. Possibly including injuries.

- **Hazardous**: Failure conditions that would reduce the capability of the airplane or the ability of the crew to cope with adverse operating conditions. Serious or fatal injury to an occupant other than the flight crew.
- **Catastrophic**: Failure conditions that are expected to result in multiple fatalities of the occupants, or incapacitation or fatal injury to a flight crewmember normally with the loss of the airplane.

2.1.6 Terms

This section starts with the introduction of important terms used throughout this work. *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; *System* as a combination of inter-related items arranged to perform a specific function(s); and, *Application* as a software instance of a function or a part of a function.

Following the D-178 standard (RTCA, 2012), software development terms are used: *Software Item* as software component or module (a part of a complete "software system"). *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.

2.2 Related Works

Housseyni et al. (HOUSSEYNI et al., 2018) 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 energy harvesting.

Five different agents are defined, one global coordinator and four locals for each

subsystem. The local agents are responsible to assess the sub-system feasibility according to the proposed reconfiguration. Such an approach makes possible independent local reconfiguration as well as coordinated global ones. Three 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.

The results showed a higher success ratio in meeting deadlines in comparison to other non-multiagent approaches.

In an avionics environment, all failure modes are identified in the development stage during the safety assessment process, therefore all possible reconfigurations can be analyzed prior to the system implementation causing the multiagent solution to be simplified as fewer local agents are necessary 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 (GIGANTE; PASCARELLA, 2012), 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 before the system implementation, making the multi-agent solution suitable for including fewer local agents as needed. The work by Housseyni et al. (HOUSSEYNI et al., 2018) did not address how the reconfiguration process affects time-critical tasks with hard deadlines as our approach does.

Cui et al. (CUI; SHI; WANG, 2018) 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 due to the high development cost implied for highly critical applications. Moreover, a local reconfiguration can lead to effects encountered in heuristic algorithms such as *hill climbing* (KLEINBERG; TARDOS, 2005). Also, *local maxima* (KLEINBERG; TARDOS, 2005) 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. (ZHOU et al., 2013) propose a framework to support the reconfiguration of avionic applications that adopt the distributed IMA architecture. In the proposed framework, an action model conforming to the Behavioral Annex of AADL (FEILER; GLUCH, 2012) 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 schedulability analysis. 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, schedulability analysis is used to determine the feasible reconfigurations, considering all possible sequences of necessary steps.

Fohler et al. (FOHLER et al., 2018) 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. (ATITALLAH et al., 2018) 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 (FPGAs) are used in the system under test and in the test benches to accomplish a unified development environment to reduce cost and time-to-market. Design Constraints were taking in consideration when exploiting the main criterias of reconfigurable circuits in terms of performance, flexibility and dynamicity. The proposed approach in this work also targets system verification supporting the design and verification phases of the product development. However it explores in more details the timeliness assurance of the system in all identified scenarios, taking in consideration the certification aspects.

Montano et al. (MONTANO; MCDERMID, 2008) present an approach to solving the complex combinatorial problem of IMA reconfiguration in real-time whilst providing support for the pilot's involvement by employing automatic generation of explanations of reconfiguration actions. The approach is based on Explanation-based Constraint Programming. The paper bring a very interesting discussion on if and how the pilots should be involved in the reconfiguration process. However it goes beyond the scope of this

work, taking additional inputs such as operational scenario, operational modes, mission objectives.

Porcarelli et al. (PORCARELLI et al., 2004) 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 using online evaluation of a stochastic dependability model which represents the whole system. The model, created at run time, depends on a set of pre-specified reconfiguration policies, on the requirements of the application and on the system status at the reconfiguration time. This strategy is hard to employ in avionics contexts with certification, specially when events triggering the reconfiguration are unanticipated and the new system state is evaluated in run time. This work focus on evaluating the possible failure modes and valid reconfiguration, in which the real time constraints can be assessed within a degrades system state.

Hollow et al. (HOLLOW; MCDERMID; NICHOLSON, 2000) focus on the reallocation problem and propose a fitness function to be applied in conjunction with a search algorithm 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 results to assure timeliness.

Annighofer et al. (ANNIGHOFER; THIELECKE, 2012), tackles the task mapping problem through a multi-objective mathematical optimization to perform software and hardware mapping within a distributed IMA architecture while designing avionic systems. Resource allocation is a well-studied area in cloud computing also (GONG et al., 2019). Where it deals mostly with proprieties such as bandwidth control and QoS. What they miss is to evaluate how the such algorithm behaves when faced with a cascading set of computer item failures.

3 SYSTEM RECONFIGURATION APPROACH

3.1 Reconfigurable System Architecture

Figure 3.1 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*.

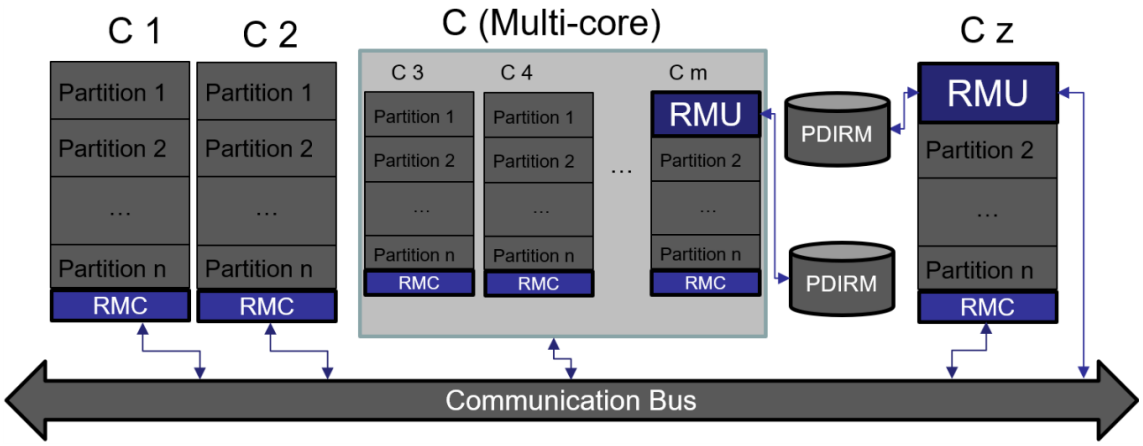


Figure 3.1 – Reconfigurable System Architecture
Source: author

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 concerning erroneous item failure detection by the RMU and manages each processing unit's 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 3.1) and Asymmetric Multi-Processing (AMP) computers (for instance, C3 in Figure 3.1). For the AMP solution, every single core has

an RMC to provide health monitoring and reconfiguration execution.

A typical reconfiguration would be triggered by the RMU if it identifies a failure in the system. Therefore, a system health monitoring mechanism should be implemented within the RMU context. The need for the RMC is to prevent the loss of a critical system function due to an erroneous RMU reconfiguration. An unnecessary state transition or even a partial reconfiguration, could create a scenario where a critical software item is not present in the new running system. For that reason, the RMC must evaluate if the reason the reconfiguration was triggered is confirmed, by an additional health monitoring. With the absence of the RMC, the RMU would be an evident system single point of failure.

Having that in mind, a few assumptions must be taken. A reconfiguration must be atomic, the system is either completely in the new state or reject the trigger. 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.

Both health monitoring, from RMU and RMC, are not subject of study of this work.

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.

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 Operating System (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.

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 that 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 (BIEBER et al., 2009). In a typical AR-

INC 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 status remains in failure until it is reconfigured to a new processing unit.

The RMU's responsibility is then to monitor the platform's 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 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 dedicated hardware to handle A664 traffic and the network topology is set to comply with the latency requirements of each application. Annighofer et al. (ANNIGHOEFER; REIF; THIELECK, 2014) propose an algorithm to generate aircraft data and communication network topologies, taking into account message flows and network component characteristics. The algorithm presented here could be used to complement, in terms of data communication efficiency, the work presented here.

3.2 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.

3.2.1 Design Flow

Figure 3.2 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 work is reconfiguration (highlighted in Figure 3.2).

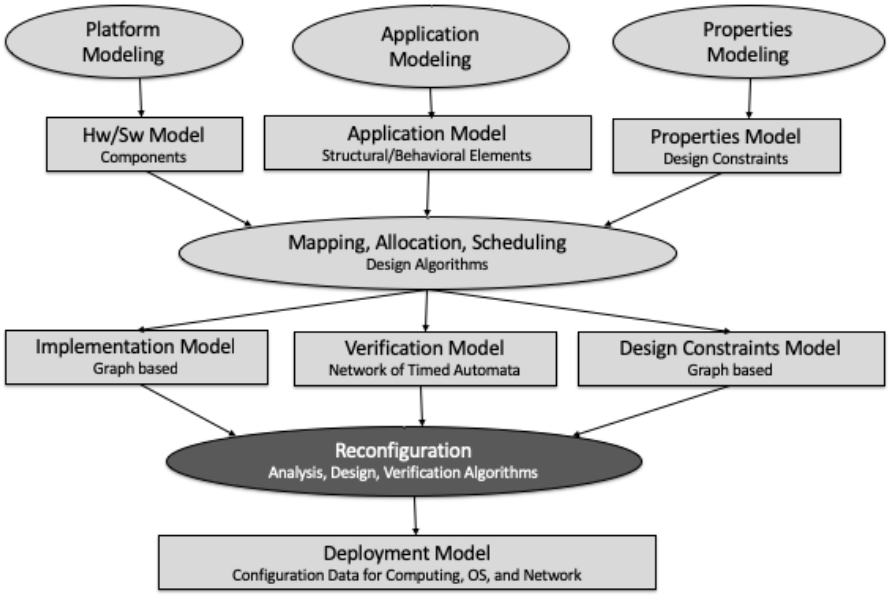


Figure 3.2 – Design Flow Overview
Source: author

For the platform, application, and properties modeling, AADL (FEILER; GLUCH, 2012) is adopted since it is already a well-studied format for the specification of avionics systems (ZHANG; WANG; LIU, 2016). 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 utilizing ports for application modeling; and, property sets for the design constraints modeling.

Using 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, allo-

cated 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 to be used by the reconfiguration algorithms, which produce a deployment model with the necessary data to perform the system reconfigurations at runtime.

3.2.2 Avionics System Specification

In this proposal, an avionics system is specified using 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. The AADL models were created using the OSATE, version 2.3.5, modeling tool from CMU-SEI (OSATE,).

3.2.2.1 Platform

consists of hardware and software components, as well as, communication buses (see Figure 3.3). A hardware component has one or more computers (mono or multi-cores), and each computer can have many partitions.

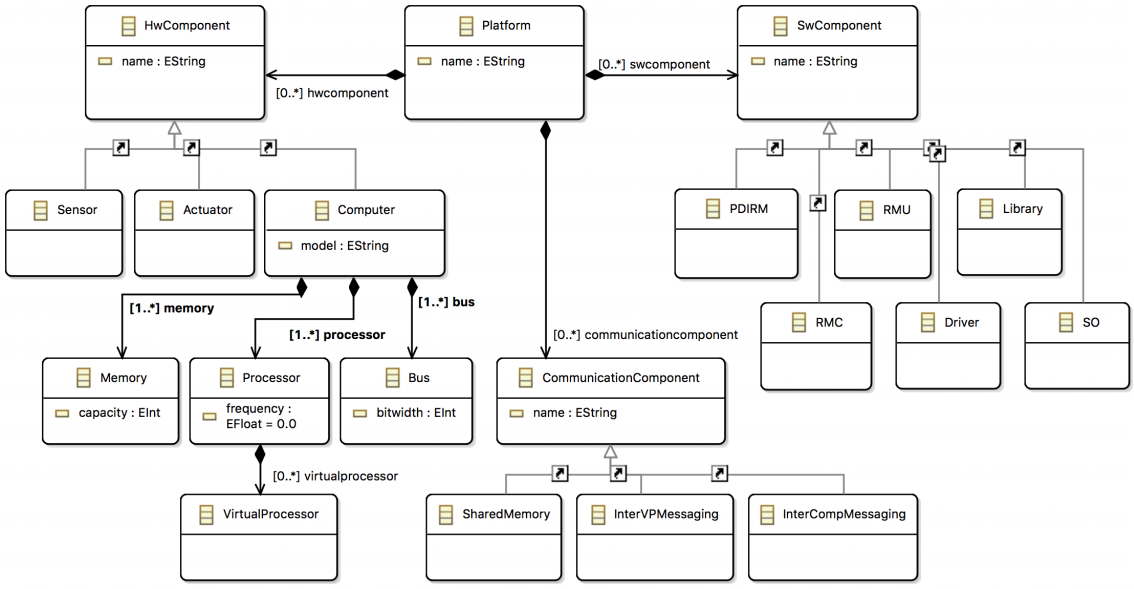


Figure 3.3 – Platform Meta-model
Source: author

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 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 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 3.4 shows an example of a Platform in AADL containing four processors C1, C2, C3, and C4, each one with three, one, two, and four partitions, respectively.

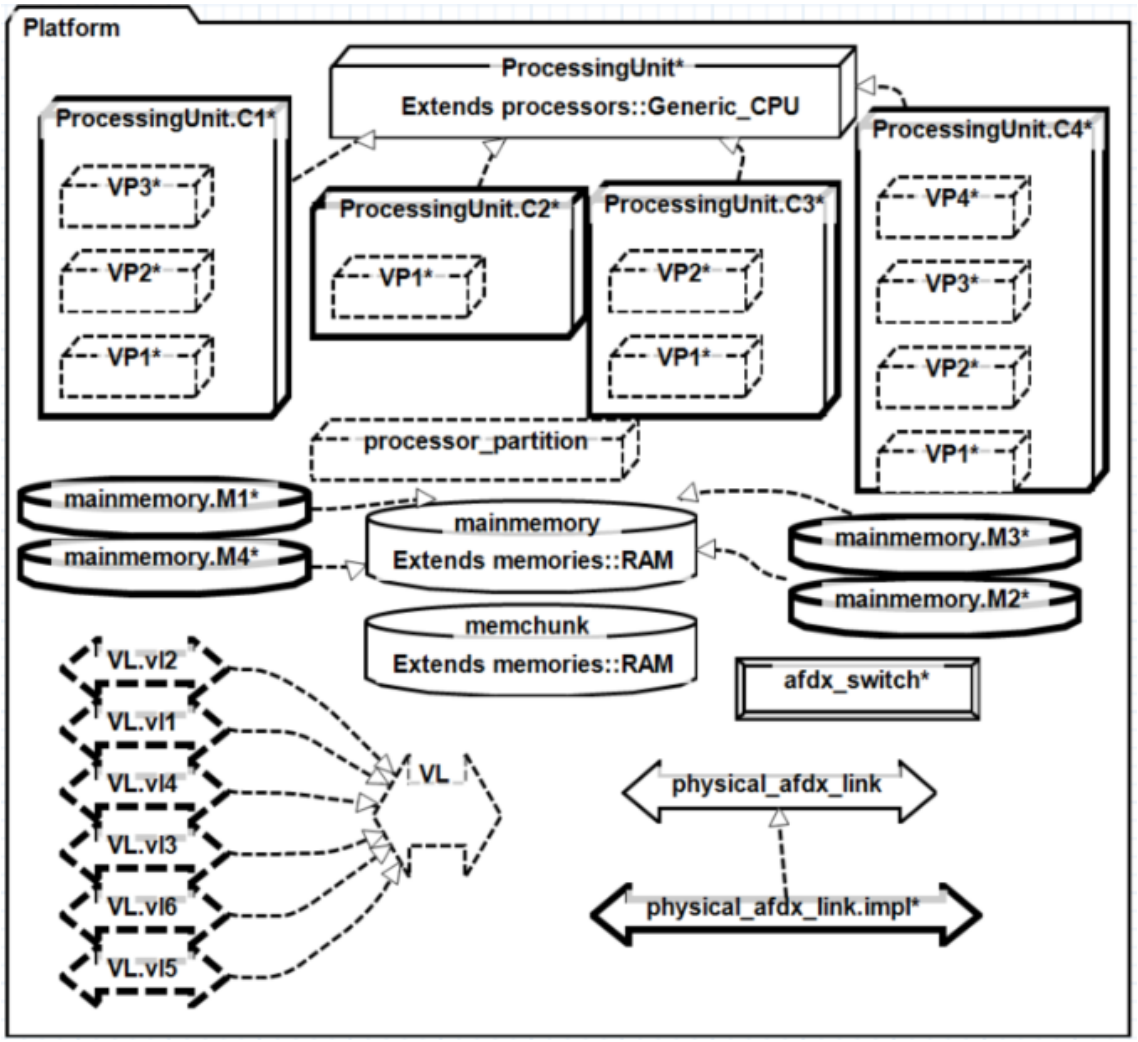


Figure 3.4 – A Platform in AADL
Source: author

Figure 3.4 shows an ARINC 664 compliant bus with six virtual links to implement the communication between the computers, and memory components. Table 3.1 presents

the specified properties for the processing units in the example presented in Figure 3.4.

Table 3.1 – 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

Source: author

As shown in Table 3.1, 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.

3.2.2.2 Application

An application consist of one or more software item. It fulfills a system function of part of a system function.

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 hierarchically. Figure 3.5 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 is specified through ports and connections between them and determines the dependencies between the software items. Thus, in the example in Figure 3.5, 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 (KLEINBERG; TARDOS, 2005) 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 3.6 presents the HDG corresponding to the application in Figure 3.5.

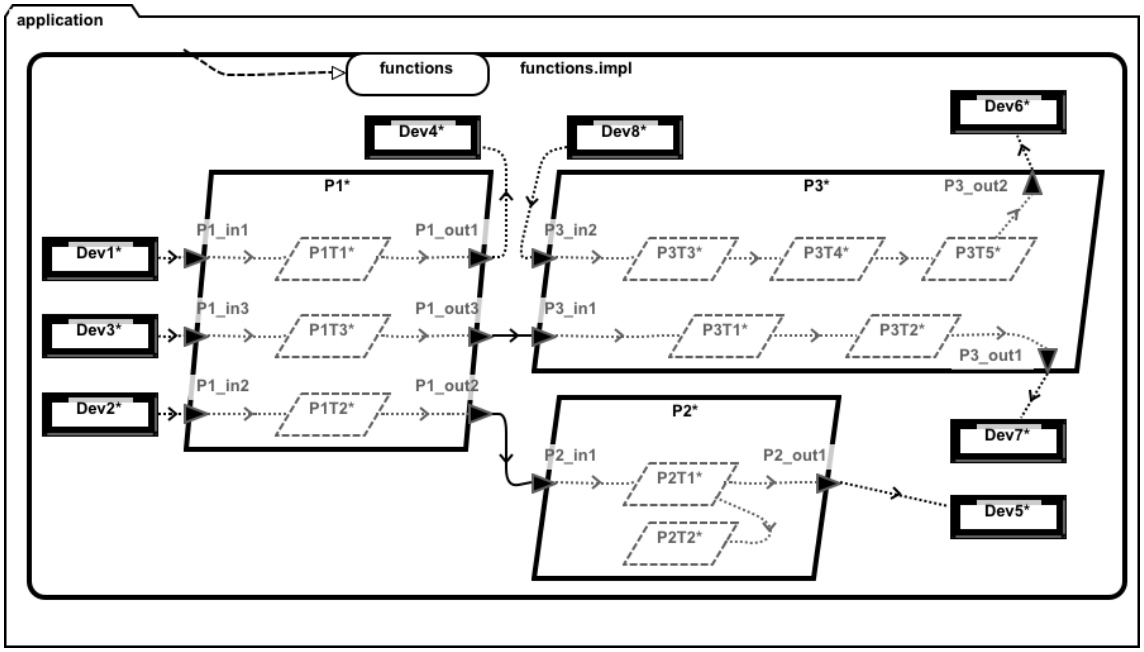


Figure 3.5 – An Application in AADL
Source: author

In the HDG in Figure 3.6, 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 3.2 shows the specified properties for the application in Figure 3.5.

Table 3.2 – 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

Source: author

As shown in Table 3.2, thread T1 of process P1 has specified period, deadline, WCET, and demanded memory given by 20ms, 20ms, 4 ms, and 90Kb, respectively.

3.2.2.3 Properties

Modelling properties allows capturing the design constraints specified by the designer that must be satisfied by any valid implementation for the system. A Property can

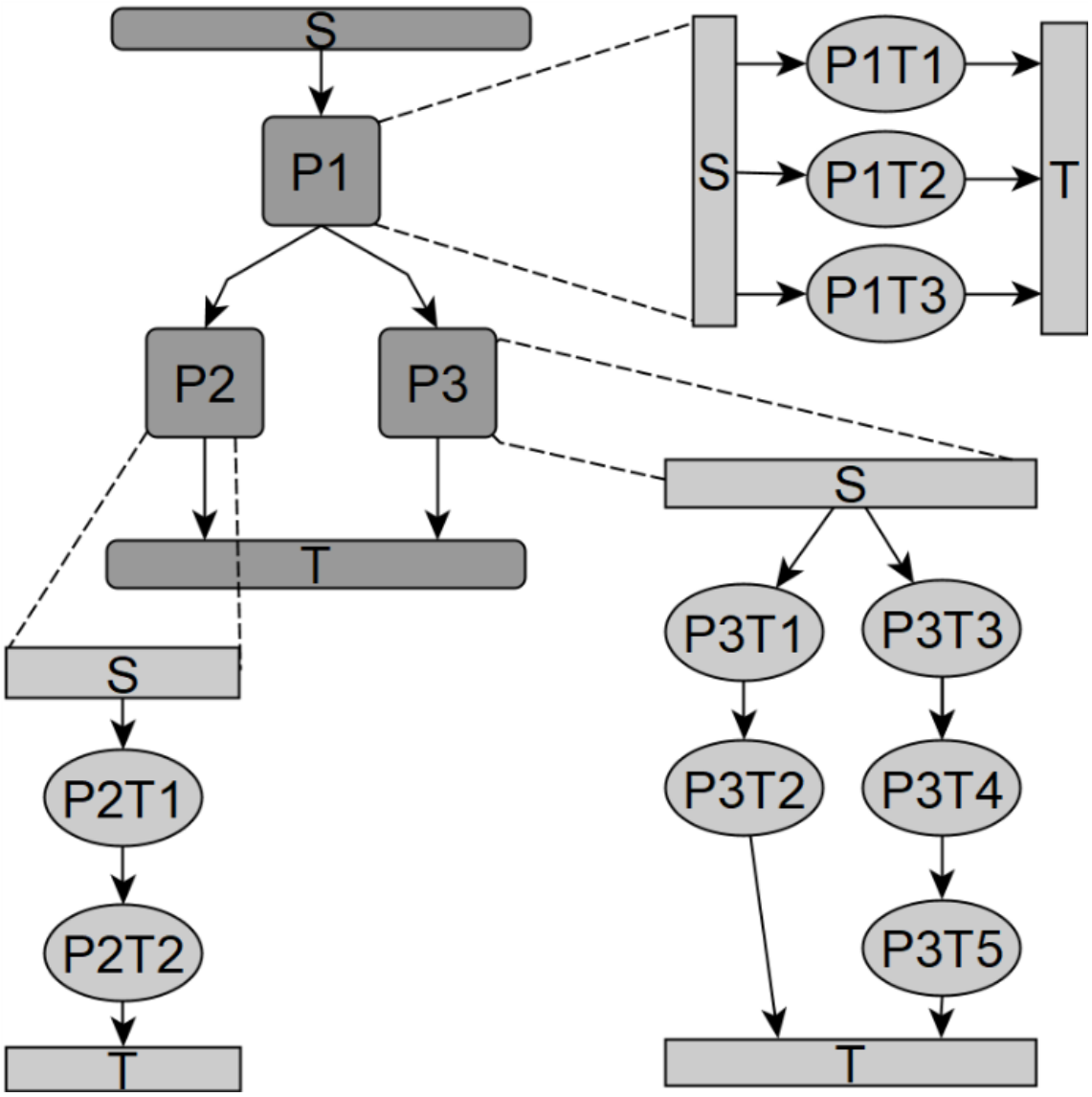


Figure 3.6 – Example of Hierarchical Dependency Graph
Source: author

represent criticality, priority, period, deadline (soft and hard), and dissimilarity characteristics of elements in the models, conforming to the proposed meta-models.

3.2.3 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.

3.2.4 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 the WCET of each thread.

3.2.5 Formal Verification

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 (BAIER; KATOEN, 2008), which can then be expressed as concepts introduced by the UPPAAL model checking tool (LARSEN; PETTERSSON; YI, 1997). 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 the guard, update, selection, and synchronization expressions. The guard's expression must be satisfied 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. (DAVID et al., 2009). According to this framework, a resource automaton for each processor partition and a task automaton for each of the process threads are 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 sends 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.

3.2.6 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 3.7 shows an example of a reconfiguration states diagram.

At the initial state S0, the system is operating normally, according to an initial mapping, allocation, and scheduling algorithm, which considers the specified properties of the system. The state transition from S0 to S1, labeled C1, indicates that when component C1 fails, the system has a feasible reconfiguration, represented by the S1 state,

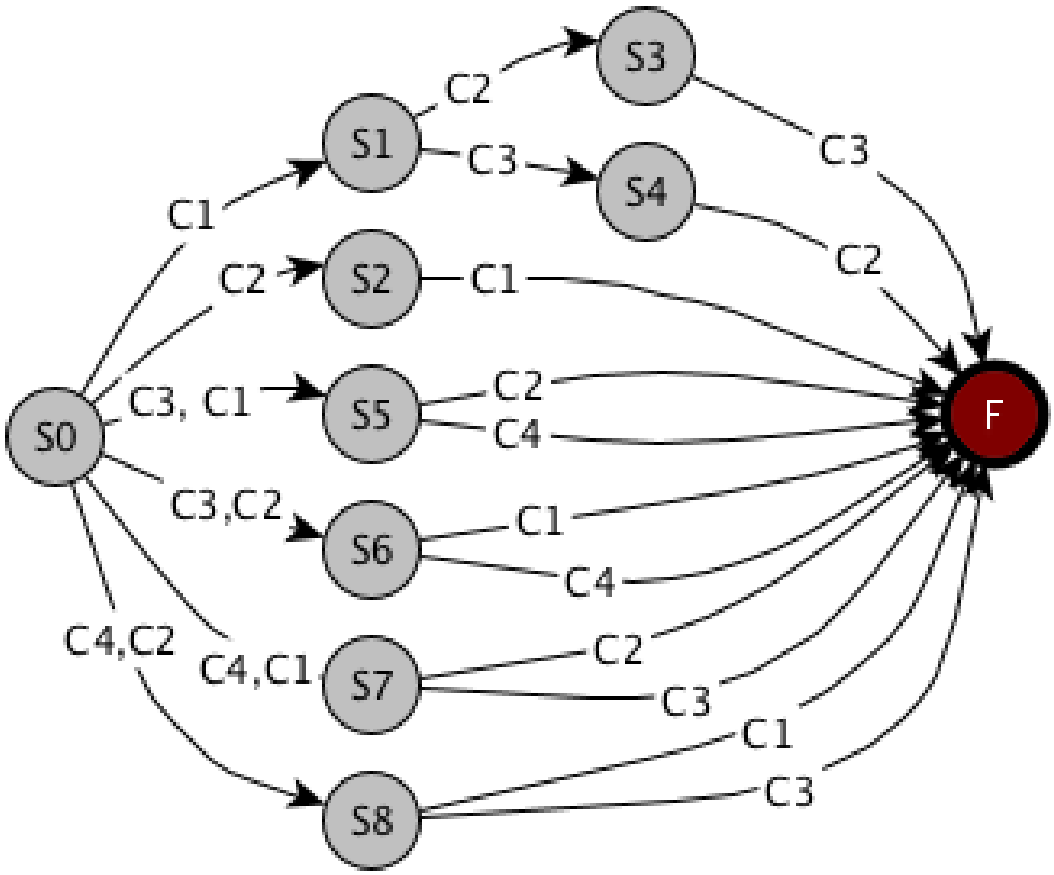


Figure 3.7 – An Example of Reconfiguration States Diagram
Source: author

which tolerates the fault. When in state S1 and component C2 fails, the system can be reconfigured according to 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 are dealt with through one reconfiguration, as stated in Chapter 1.

3.2.7 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.

3.2.8 Design Automation Algorithms

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

3.2.8.1 Mapping, Allocation, and Scheduling

The algorithms that build the implementation model and the deployed model are 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, which should satisfy the specified design constraints. This step produces an allocation matrix, containing the information to be used by the algorithm for the schedulability analysis. It also instantiates a corresponding deployment model to be used when building the reconfiguration states diagram, described in the next subsection. Table 3.3 shows the allocation matrix produced from the platform and the application models, shown in Figures 3.4 and 3.5, respectively, and platform and application properties listed in Tables 3.1 and 3.2, respectively.

Table 3.3 – 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
...	...	...	...	...	...	...	...	...	...	...

Source: author

As shown in Table 3.3, 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 and allocated on its virtual processors VP1, VP2, VP3, which are the first ones avail-

able 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 3.7 shows the Reconfigurable States Diagram that will be generated based on this allocation matrix.

3.2.8.2 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.
- In line 2, the algorithm starts from a source state (Ss) as the 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
- 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 with 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.

3.3 CASE STUDY

To evaluate the proposed approach for avionics systems development, a case study is proposed bringing examples from the avionics industry. A top-down approach is chosen as recommended by the ARP4754 (SAE, 2010), defining at first the system functions and subsystem functions:

- **Flight Control**
  - Flight Control
  - Fuel Management
- **Provide Auto Pilot**
- **Provide Navigation**
  - Guidance
  - Route Control
  - Provide Map
  - Provide Charts
- **Provide Human Machine Interaction** (HMI)
  - Screen Monitoring (display symbology)
  - System Control
  - Provide input for flight control
- **Provide System monitoring**
  - Fuel Monitoring
  - Engine Monitoring

The system functions are the highest level definitions in a system and specify its basic functionalities. From this definition, the breakdown 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 Hazard Analysis (FHA) is performed. The FHA is a systematic, comprehensive examination of the airplane and system functions to identify potential minor, major, hazardous, and catastrophic failure conditions that may arise as a result of a malfunction or a failure to function. 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. Table 3.4 shows the complete analysis performed for this case study.

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 ≠ 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 a new Deployment model from current
                        mappings;
22                     create a 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 a 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

Table 3.4 – Function Hazard Analysis (Continue on the next pages)

Function	Sub Function	Failure Condition	Flight Phase	Effect	Failure Condition Severity Classification	Function Design Assurance Level
Flight Control		Loss of Flight Control	Landing/ Take-off/ Flight	Aircraft Loss	Catastrophic	FDAL A
	Flight Control	Undetected Erroneous Flight Control	Landing/ Take-off/ Flight	Aircraft Loss	Catastrophic	FDAL A
	Fuel Management	Loss of Fuel Management	Flight	Significant increase of crew workload	Major	FDAL C
		Undetectable Erroneous Fuel Management	Flight	Mitigation: Fuel Monitoring After Mitigation: Significant increase of crew workload	Major	FDAL C
		Loss of Auto Pilot	Landing	Significant increase of crew workload	Major	FDAL C

Provide Auto Pilot	Provide Auto Pilot	Undetectable Erroneous Auto Pilot	Landing	Aircraft Loss	Catastrophic	FDAL A
		Loss of Guidance	Landing/ Take-off/ Flight	Significant increase of crew workload	Major	FDAL C
	Guidance	Undetectable Erroneous Guidance	Landing/ Take-off/ Flight	Higher workload such that the crew could not be relied upon to perform tasks accuratly	Hazardous	FDAL B
		Loss of Route Control	Landing/ Flight	Significant increase of crew workload	Major	FDAL C

Provide Navigation	Route Control	Undetectable Erroneous Route Control	Landing/ Flight	Mitigation: Guidance Functionality can provide monitoring capability for an eventual erroneous route configuration After Mitigation: Significant reduction in safety margins Significant increase of crew workload	Major	FDAL C
		Loss of Map Source	Flight	Slight increase in crew workload	Minor	FDAL D
	Provide Map	Undetectable Erroneous Map Information	Flight	Mitigation: Map not to be used as the main navigation source After mitigation: Slight increase in crew workload	Minor	FDAL D
		Loss of Charts Source	Landing/ Take-off/ Flight	Slight increase in crew workload	Minor	FDAL D

Provide Human Machine Interaction	Provide Charts	Undetectable Erroneous Provide Charts	Landing/ Take-off/ Flight	Mitigation: Chart not to be used as the main approach guide After mitigation: Slight increase in crew workload	Minor	FDAL D
	Screen Flight Critical Data Monitoring (display symbology)	Loss of Screen Flight Critical Data Monitoring (display symbology)	Landing/ Take-off	Higher workload such that the crew could not be relied upon to perform tasks accuratly	Hazardous	FDAL B
		Undetectable Erroneous Flight Critical Data Screen Monitoring (display symbology)	Landing/ Take-off	Aircraft Loss	Catastrophic	FDAL A
		Loss of Screen non-Critical Data Monitoring (display symbology)	Landing/ Take-off	Slight increase in crew workload	Minor	FDAL D

	Screen non-Critical Data Monitoring (display symbology)	Undetectable Erroneous non-Critical Data Screen Monitoring (display symbology)	Landing/ Take-off	Slight increase in crew workload	Minor	FDAL D
		Loss of System Control	Landing/ Take-off	Higher workload such that the crew could not be relied upon to perform tasks accuratly	Hazardous	FDAL B
	System Control	Undetectable Erroneous System Control	Landing/ Take-off	Aircraft Loss	Catastrophic	FDAL A
	Provide input for flight control	Loss of input for flight control	Landing/ Take-off/ Flight	Aircraft Loss	Catastrophic	FDAL A
		Undetectable Erroneous input for flight control	Landing/ Take-off/ Flight	Aircraft Loss	Catastrophic	FDAL A

	Fuel Monitoring	Loss of Fuel Monitoring	Flight	Significant reduction in safety margins Significant increase of crew workload	Major	FDAL C
		Undetectable Erroneous Fuel Monitoring	Flight	Large reduction in safety margins Higher workload such that the crew could not be relied upon to perform tasks accuratly	Hazardous	FDAL B
		Loss of Engine Monitoring	Landing/ Take-off	Large reduction in safety margins Higher workload such that the crew could not be relied upon to perform tasks accurately	Hazardous	FDAL B

Provide System monitoring	Engine Monitoring	Undetectable Erroneous Engine Monitoring	Landing/ Take-off	Large reduction in safety margins Higher workload such that the crew could not be relied upon to perform tasks accurately	Hazardous	FDAL B

Source: author

The worst failure condition for Flight Control, for instance, is aircraft loss which can be classified as catastrophic according to the ARP 4761 (SAE, 1996). Such classification implies the function design assurance level (FDAL) to be A. This enforces the group of items that realize the flight control to be developed at the highest level of assurance. While the functions related to minor events must comply at least to level D which is the second lowest on 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 traced only to the top-level system functions. Figure 3.8 illustrates what are the items dependencies which generates the HDG described in Chapter 3.2.2.2. The real dependency diagram was created in AADL using annex ARINC653 and is not included in this work due to its complexity and size. The complete model can be found in the project repository in Github[1]. Each node identified in Table 3.5 is modeled in AADL as a thread that 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 3.8 also partially shows how the system functions are realized by the software items. The *flight control* for instance is realized by *FlightStickHandler*, *Va_Control* and *Vz_Control*. The two latter components are based on an Open-Source Avionics and Control Engineering case study (PAGETTI et al., 2014). 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 IDAL as it can be seen in Table 3.5. 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 Chapter 3.2.2.3. This particular property carries an important design constraint for the allocation algorithm mentioned in Sections 3.2.4 and 3.2.8. 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. This 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 FDAL therefore a lower

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

Table 3.5 – 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

Source: author

IDAL as shown in Table 3.5 (node 01 in the first column).

Table 3.5 also presents other important item properties that need to be taken into consideration by the allocation algorithm, i.e., the 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 3.9 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 *Fuel-*

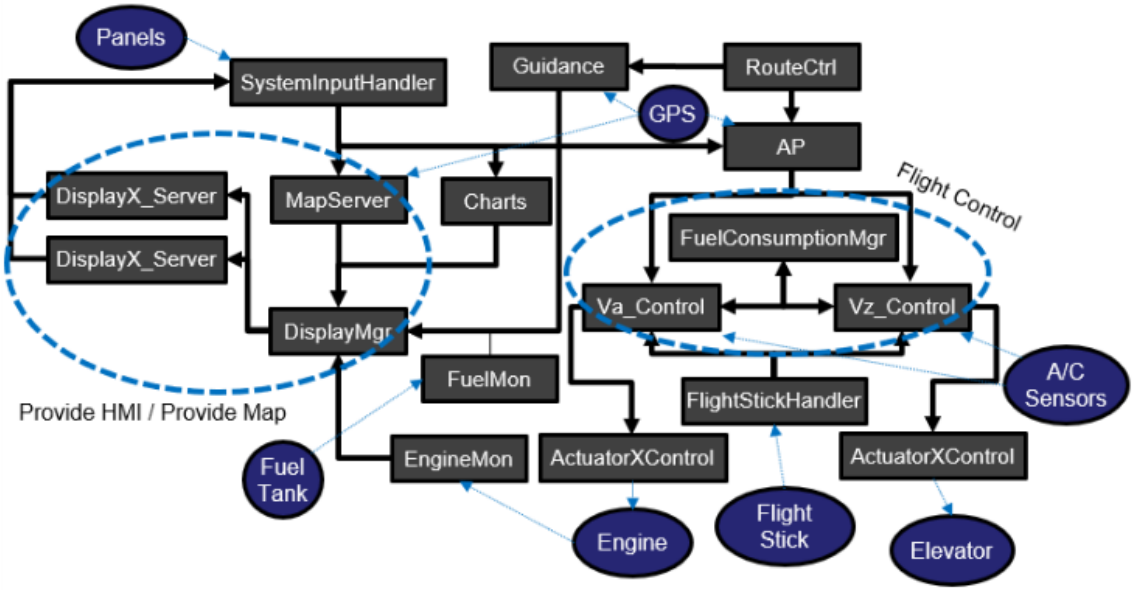


Figure 3.8 – Initial System Dependencies
Source: author

Table 3.6 – 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	4	8	3	7	3	5	4	1	4	6	8	2

ConsuptionMgr, 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 can present critical information to 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 3.6 specifies the computers properties of the platform shown in Figure 3.9. The RMU and RMCs are embedded within the computers in this platform as presented in Figure 3.1.

As previously presented in Chapter 3.2, the HDG is generated from the dependency diagram created in AADL, shown in Figure 3.10. Here, the number on each node refers to the node # in Table 3.5. The HDG is used as one of the inputs for the UPPAAL model. Figure 3.11 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 are verified by applying the UPPAAL model checker. Each node is a

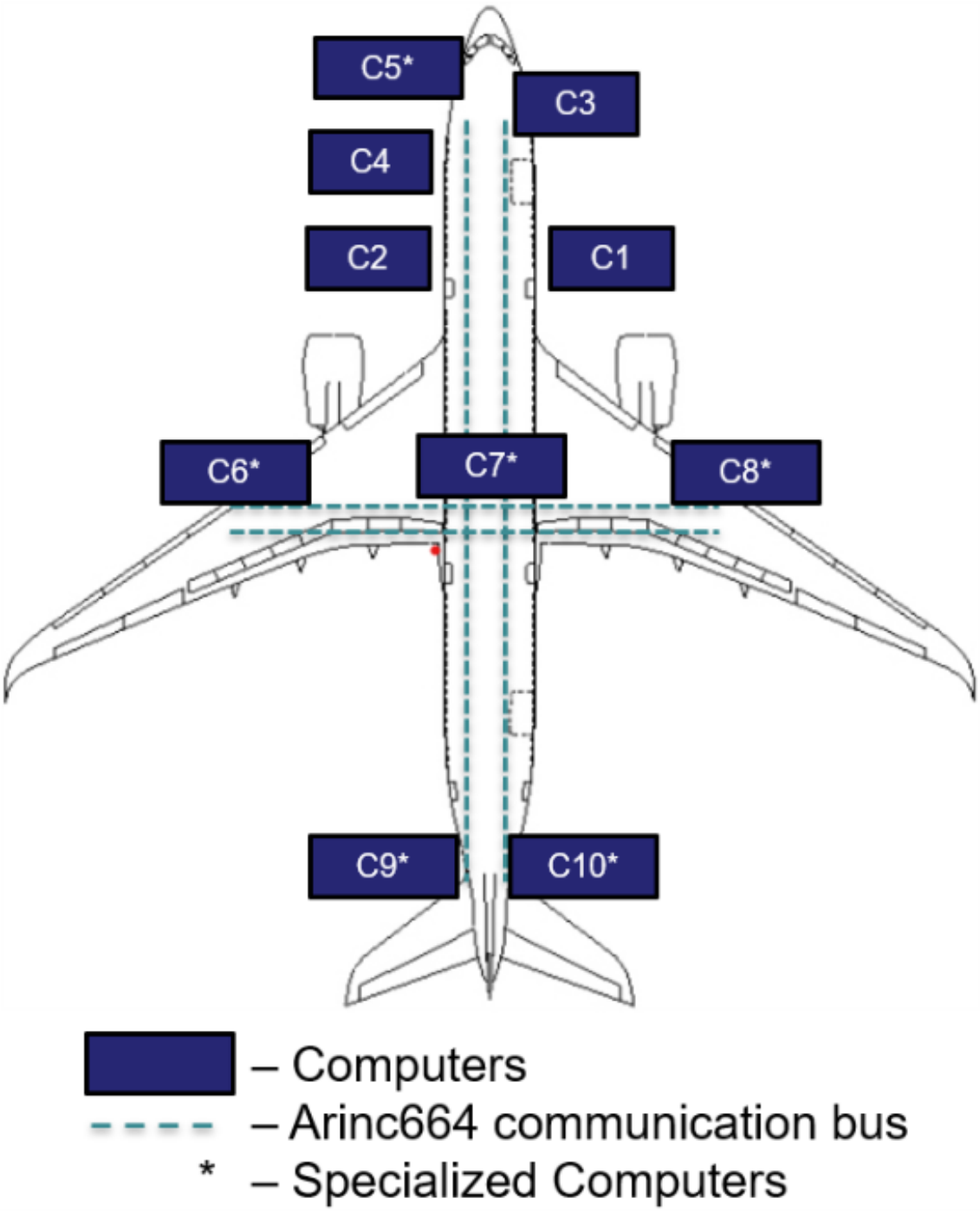


Figure 3.9 – Allocation of functions to nodes within the platform
Source: author

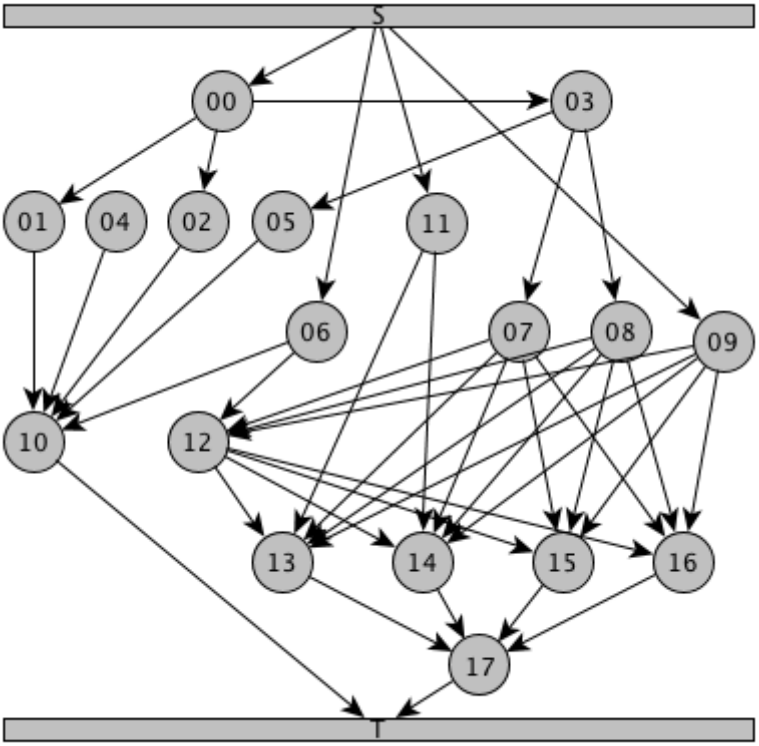


Figure 3.10 – HDG for case study
Source: author

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.

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 a Mac OS X system on an Intel i7 2,2GHz with 16Gb RAM. We used the option *over approximation*, available in UPPAAL (utilizing parameter `-A` for the command `verifyta`), which reduces the number of explored states by applying a convex-hull based approximation technique (DAVID et al., 2009). Even so, in this case, UPPAAL reached more than 62 million states.

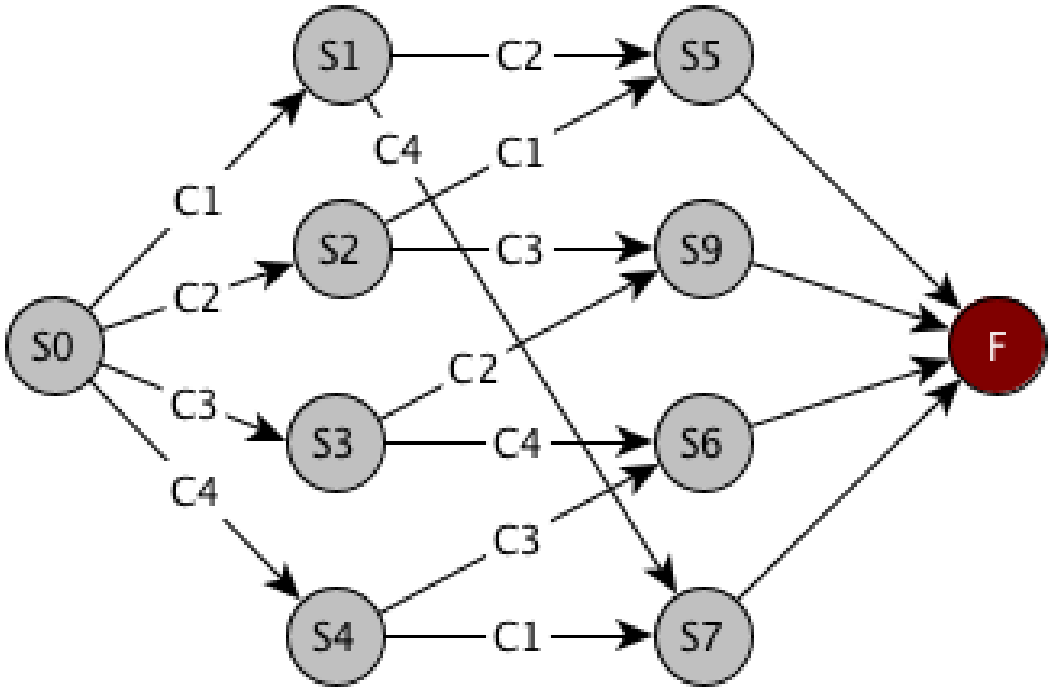


Figure 3.11 – RSD for case study
Source: author

4 RESOURCE ALLOCATION AND SYSTEM RECONFIGURATION FRAMEWORK

A crucial phase of the design flow is resource allocation. Resource allocation is the process of allocating a mixed-criticality task set to processing elements for execution respecting predefined design constraints.

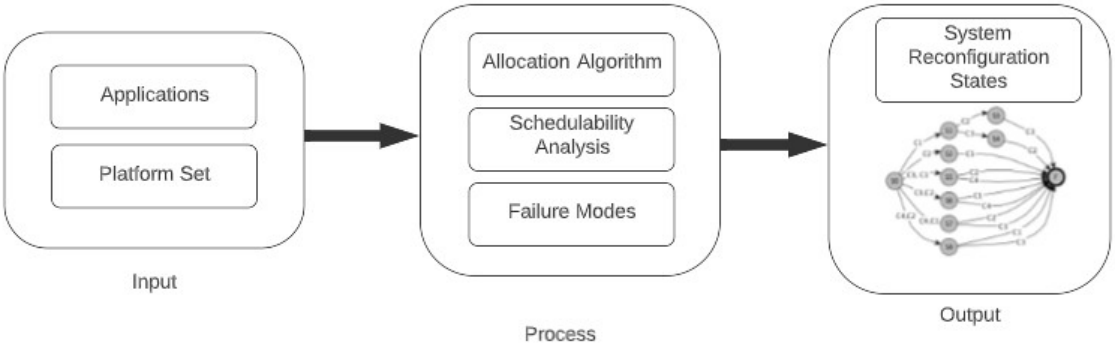
From the main failure modes of the system, several reconfiguration states are possible with a reduced number of resources. The process should be able to reallocate the tasks which were impacted by the failure itself. In the end, the reconfiguration states set and its transitions, which is the output of the process, must be in accordance with the reliability requirements of the functions under consideration.

An avionics system has a limited and expensive set of resources therefore choosing which processing resources are more suitable for a certain task set is crucial and a complex problem.

The main key performance indicators (KPI) for such a process are the number of required processing elements and the reliability figure of the overall system.

A framework is proposed and developed in this work to automate this phase of the design flow. Though out the development phase and even during the life cycle of an avionics system, requirements often change therefore a modular approach was chosen, so different allocation algorithms and schedulability analysis methods can be implemented and incorporated into the framework according to the system design decisions from system architects. A typical algorithm can easily satisfy the minimum KPIs at the beginning of a project, however with the demand for new functionalities a more sophisticated allocation might have to be applied to keep up with the strict reliability budgets.

Figure 4.1 – Framework Input Output



Source: author

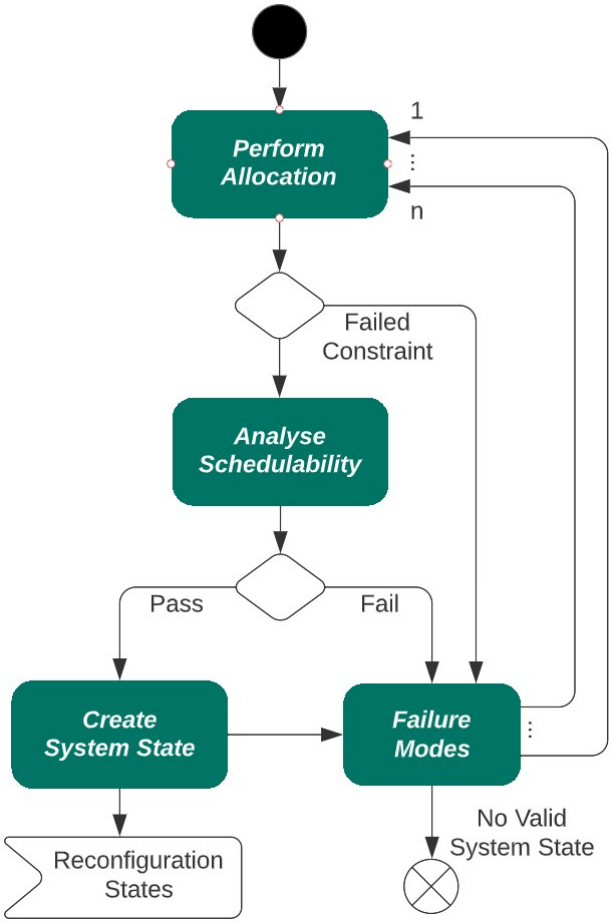
The figure 4.1 depicts at a high level the reconfiguration process with its inputs

and outputs. The applications list is provided with their proprieties set which is taken as constraints by the algorithms.

The platform set is described in the model of figure 3.3. It is important to note that a few limitations are imposed at the design level by the system or software design engineer responsible. For instance, the allocation algorithms do not decide upon the number of partitions (*VirtualProcessor* in the model) and their proprieties, those are specified within the platform model itself before the algorithm execution. Due to the intrinsic modular philosophy and automation of the framework, the platform proprieties can be modified until the best suitable solution is presented as a result.

Figure 4.2 shows an activity diagram with the high-level control flow of the framework. The first activity at the start is the *Perform Allocation* which creates an initial allocation of a set o given processes to a platform. At this stage, different algorithms can be evaluated if necessary. This analysis is demonstrated in this work in chapter 4.5 with the comparison of 3 different main algorithms.

Figure 4.2 – Framework Process



Source: author

The subsequent activity is the *Analyse Schedulability* where the proposed allocation is checked for its schedulability. Here, also, different tools can be evaluated to perform such tasks. More on the schedulability analysis tool selection in chapter 4.4. The result is an input for the decision criteria to create the proposed allocation as a system state in the final reconfiguration states.

In the following activity, the framework iterates over the next possible system failure mode according to the previously given platform. For instance, with 4 processing elements in the hardware set, the framework would trigger 4 different new allocation attempts, one for each failed computer.

4.1 Output Analysis

From the generated reconfiguration states and the related state transitions, the framework can calculate the resulting function failure rates by interpreting it as a reliability block diagram (RBD).

The first step is to identify all the failure paths in which the function under consideration, through the derived application, would result in a time constraint error. The list of failure paths can be taken as simplified series blocks on the RBD theory and each failure path can be considered a simple parallel Systems, thus can be calculated as follow:

$$Q_p = \prod_{i=1}^{n} P(X_i) \tag{4.1}$$

Being the $P(X_j)$ the probability of failure of the i^{th} computer in the failure path and Q_p the failure rate of the complete failure path.

The function failure rate is then calculated as:

$$Q_f = MAX(Q_{p1}, Q_{p2}...Q_{pn}) \tag{4.2}$$

The result from equation 4.2 can then be compared to initial safety requirements derived from the FHA. The framework output would be deemed acceptable if all calculated function failure rates are less than the required figures.

4.2 Allocation Algorithms

In addition to the algorithm presented in chapter **??**, two additional algorithms from that field were taken as a base for comparative analysis. Due to the fact the presented case study in this work is applied to a different set of constraints and infrastructure, the studied algorithms cannot be analyzed as is. Few adaptations were done to turn the results into a valid allocation for an avionics systems scenario.

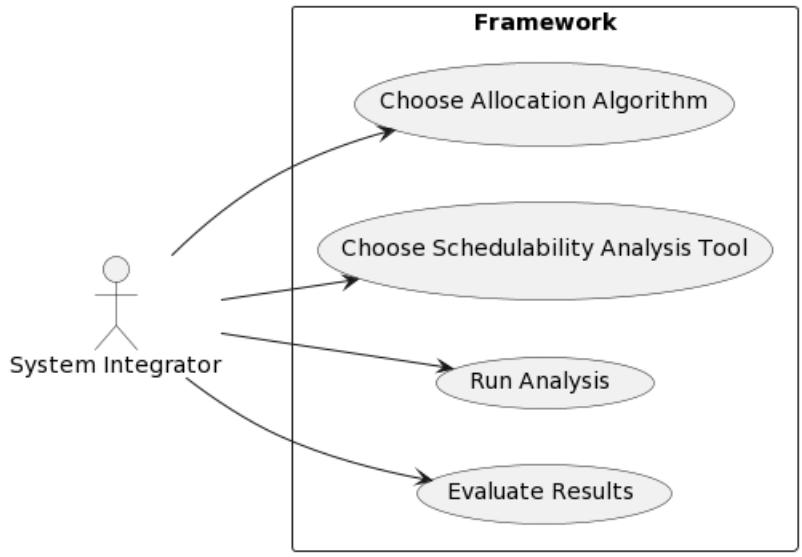
The main constraints for the resource allocation algorithm were already presented in chapter 3.3, however, there are a few more that were introduced to create a working design automation framework.

As a common practice when it is evaluating partitions WCET and computer load, a jitter is calculated according to the used Real-Time Operation System (RTOS) resources by the applications. Due to the fact, that the case study does not bring details into the application implementation, a default jitter was used for every single application allocated to a certain partition, which can be interpreted as the preemption jitter. Using this approach creates a more realistic deployment prior to the schedulability analysis. During this study, the value of 10us was chosen.

Another aspect is the partition timing proprieties adjustments in case of other empty partitions. Commercial Operating Systems already brought improvements to improve the efficiency in resource usage in an ARINC653 compliant system. For instance, the *VxWorks 653* by *Wind River* provides an option for priority preemptive scheduling (PPS) of partitions which allows designated partitions to consume what would otherwise be an idle time in the defined ARINC schedule (PARKINSON; KINNAN, 2015). However, this is not specifically defined in the standard and therefore cannot be taken as a given in all different OS implementations.

As a rule, the framework will not keep a complete idle partition in the system configuration. To optimize time and keep real-time proprieties of running processes intact, the partition's time slice initial delay is not changed. Instead, the partition duration is increased in case the subsequent one is determined as completely idle (e.g. no process allocated).

Figure 4.3 – Framework Use Case Diagram



Source: author

4.3 Software Design

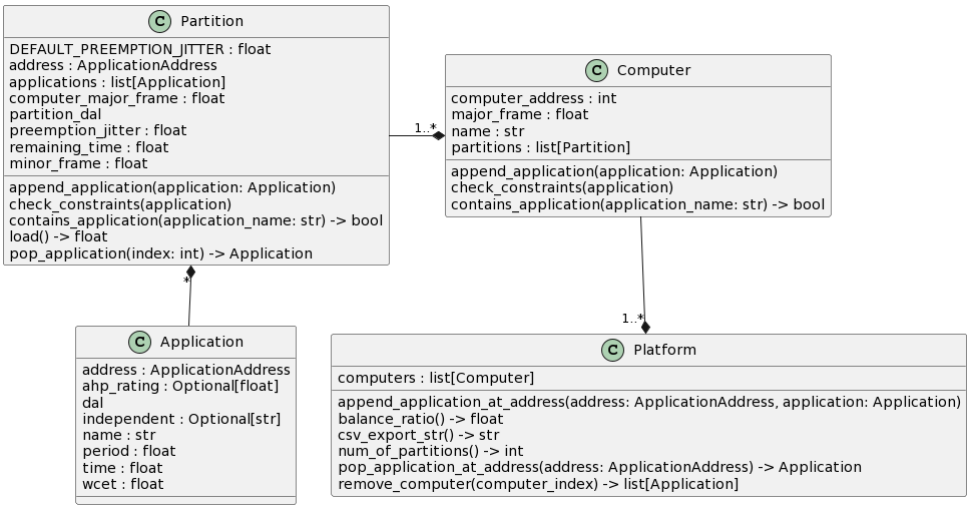
In this chapter, the software design which implements the framework according to figures 4.1 and 4.2 is described in details.

The framework use cases are depicted in the figure 4.3. The System Integrator can choose an allocation algorithm from a library list. Due to the modular approach, new algorithms can be added to the list if desired. In addition, the user can choose the schedulability analysis tool from a supported list of tools. A wrapper must be implemented in case a new tool is included in the framework. Finally, the System Integrator runs the analysis and evaluates the results.

Four different class diagrams are presented to explain the main classes and their relationship within the framework: Platform, Allocation Algorithm, Schedulability Analysis, and Reconfiguration States class diagrams.

The Platform Class Diagram (figure 4.4) shows the representation of applications, partitions and computers. The structure is similar to the platform meta-model presented in chapter 3.2.2.1. For a platform to be valid, it has to have at least one computer. The same applies to the computer/partition relationship, as specified by ARINC 653. Initially, the partitions do not have any application, but that changes after the initial allocation. In the platform class, a couple of methods are worth mentioning as they are crucial throughout the framework activities. The *append_application_at_address* method performs an attempt allocating an application into the platform. It checks all the specified constraints

Figure 4.4 – Platform Class Diagram



Source: author

and it results in a fail if any of them is not met. The behavior is also dependent on the address, which gives different levels of freedom whether it specifies strictly the partition in which the application should be allocated, or if it directs only to an overall computer, giving a more simple greedy choice of the partition.

The *remove_computer* method is part of the failure mode activity, which modifies the platform to a degraded state as seen in the hardware fault tree analysis.

The allocation algorithm class diagram (figure 4.5), shows the base class and what are the overloaded methods within the inherited classes.

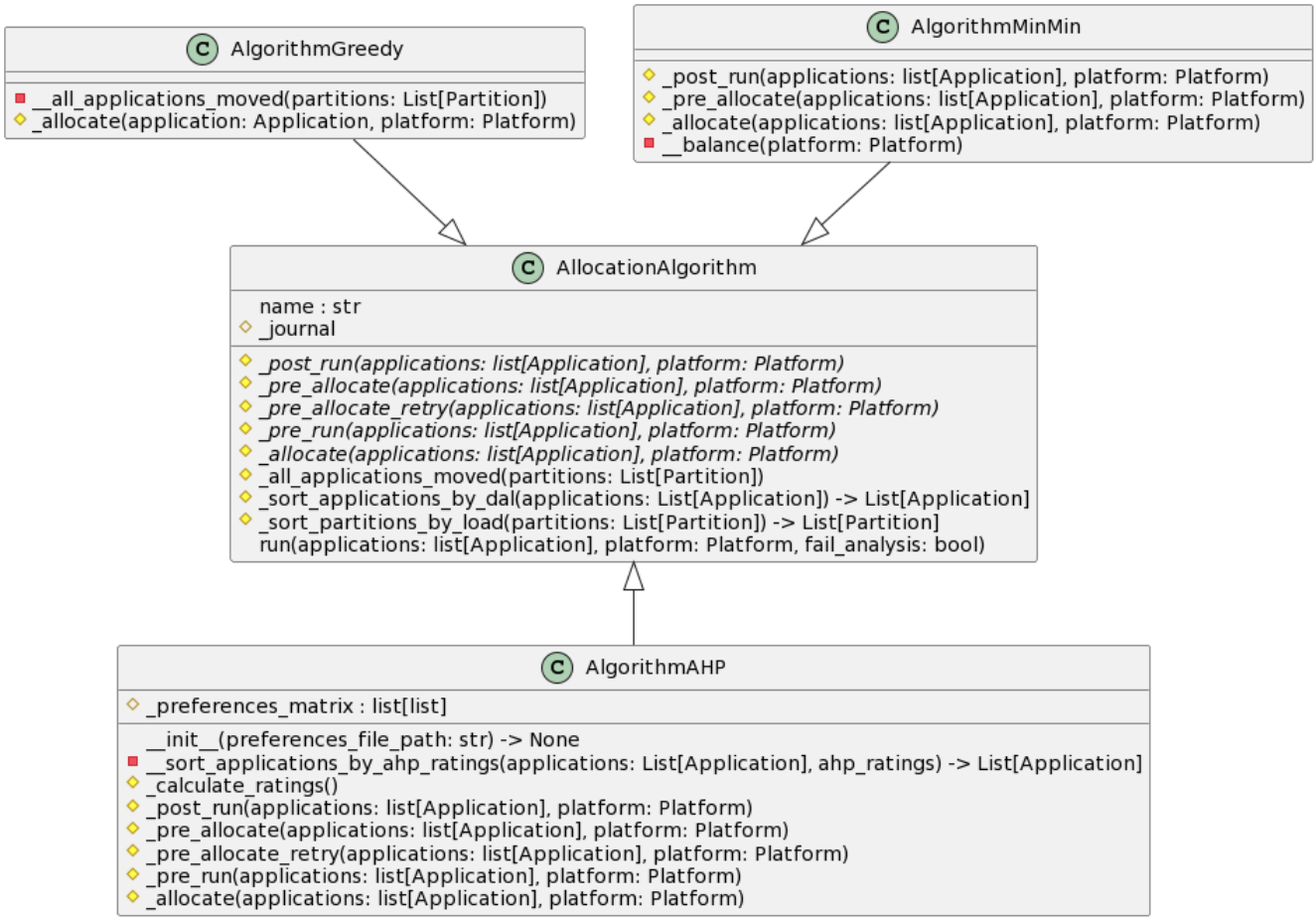
The execution of the generic allocation algorithm is shown in figure 4.6. It allows the child classes to overload only parts of the flow, which reduces the effort if a new variant of the algorithm needs to be implemented. For instance, if a greedy allocation is implemented initially, however, it is realized the need for a platform balance as the following activity. In that scenario, the software engineer can create a new child class inherited from the greedy allocation class and overload only the *_post_run*.

The base execution includes a retry mechanism which could cause the system state to be in a degraded mode, but still be schedulable. In that situation, it is ensured that all highly critical applications are allocated, however low critical ones might have been left out. The system is not in a failure state, but with reduced functionality. For instance, the chart visualization for the pilots is not crucial to a landing procedure and can be left out of the system configuration in favor of the flight stick handler process.

The child classes algorithms are detailed in chapter 4.5.

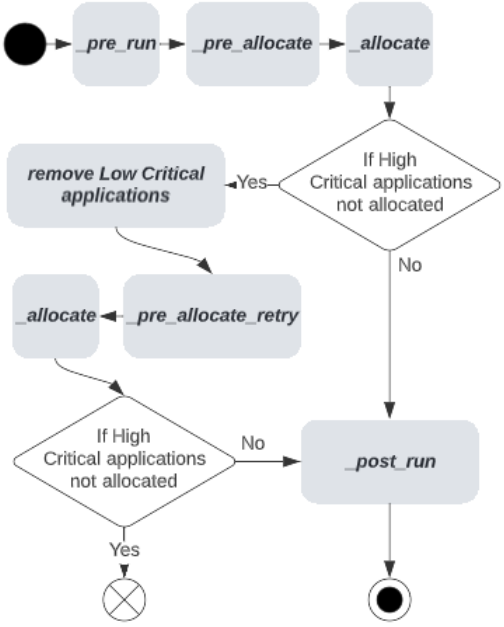
The schedulability analysis class diagram (figure 4.7) shows what's the bare minimum for a new tool wrapper to be included in the framework. The *is_schedulable* method

Figure 4.5 – Allocation Algorithms Class Diagram



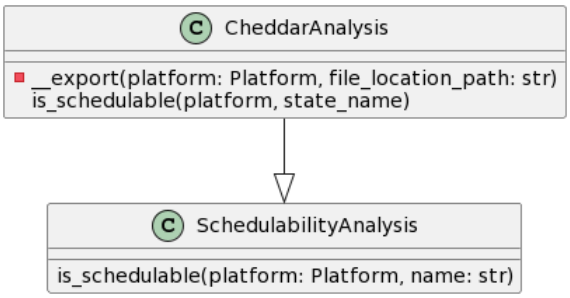
Source: author

Figure 4.6 – Base Allocation Algorithms Flow Diagram



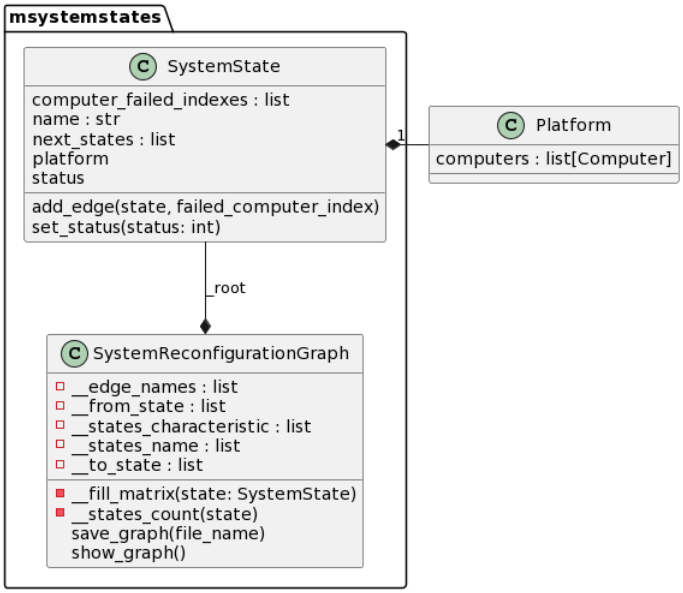
Source: author

Figure 4.7 – Schedulability Analysis Class Diagram



Source: author

Figure 4.8 – Reconfiguration (System) States Class Diagram



Source: author

will be called during the *Analyse Schedulability* activity shown in figure 4.2.

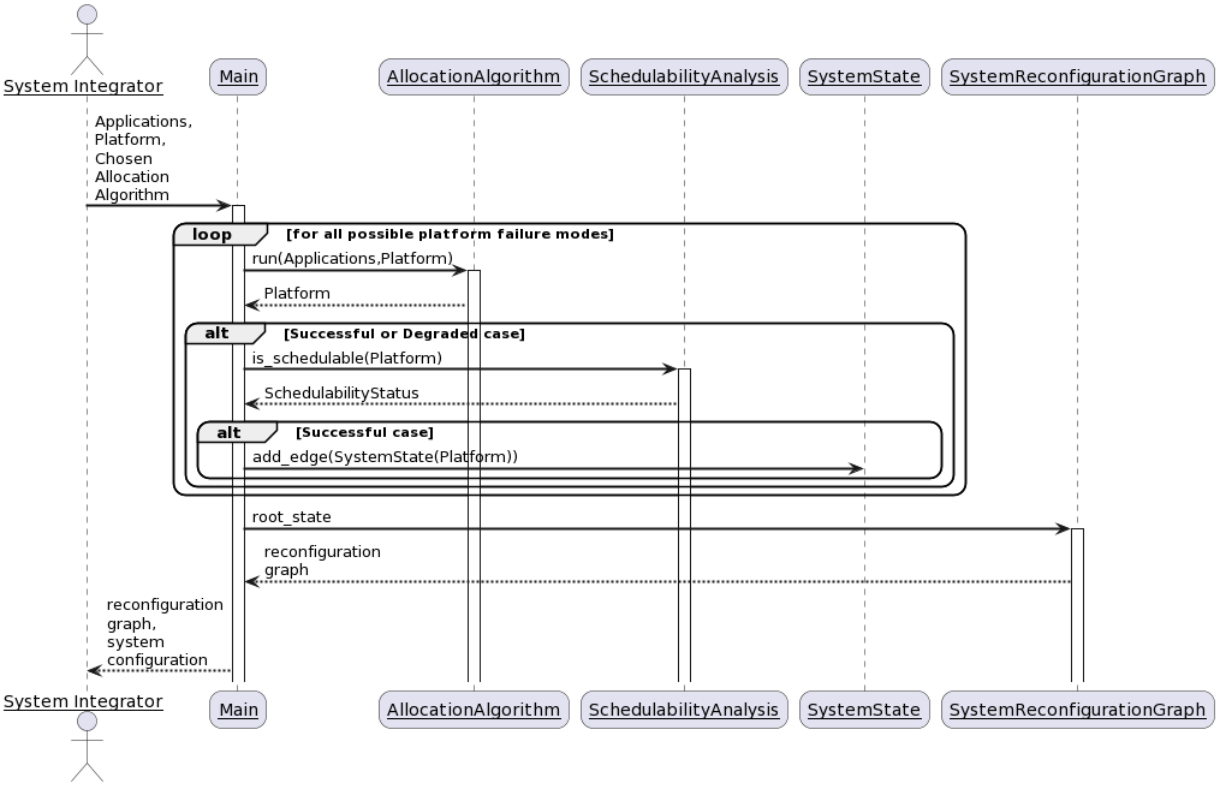
The *SchedulabilityAnalysis* child class will be further described in chapter 4.4.

The reconfiguration states class diagram (figure 4.8) represents the structure to build the framework output. A system state is composed of a schedulable platform with allocated applications which is the result of the previous two activities. It contains the status of the state (*schedulable_degraded* or *schedulable*) and the *next_states* which is represented by a list of tuples : the failed computer that triggers the new state, and the address of the actual new *SystemState*.

The *SystemReconfigurationGraph* is responsible to create a visual representation of the complete set of possible system states taking into consideration the known failure modes.

Finally, the framework sequence diagram (figure 5.8) describes the sequence calls of methods from the aforementioned classes since the interface with the system integrator.

Figure 4.9 – Framework Sequence Diagram



Source: author

The framework was developed in python and the source code is available in GitLab [1].

4.4 Schedulability Analysis Tools

The results from the initial analysis and formal verification of the case study using UPPAAL were not conclusive e extremely time-consuming especially due to the state space explosion caused by the complex model used for this exercise. In addition, the tool does not support parallel processing out-of-the-box which eliminates from the start any possibility to seek more powerful computational systems to solve the time constraint. Therefore, schedulability analysis tool alternatives were evaluated.

MAST (HARBOUR et al., 2001) is a set of software tools for schedulability analysis of real-time applications developed by the University of Cantabria, Spain. Similarly, CHEDDAR (SINGHOFF et al., 2009) also provide means for checking tasks temporal constraints of a real-time application and it was developed by the University of Brest in France.

[1] https://gitlab.com/afontoura/pampaflie

Both support distributed architecture however the currently available MAST version does not support partition scheduling and hierarchical schedulers. Even though improvements were planned and the required feature intended to be included in a second iteration, no updates were made since 2014. Cheddar, on the other hand, is still active. This can be noticed with recent publications ((DJIKA et al., 2021)) and continuous tool development as shown by activities in the tool source code repository. Moreover, it supports the basic ARINC653 specification with temporal partitioning.

However, a few limitations are imposed by CHEDDAR. The dependencies between processes through periodic messages with different criticality level and different periods are not well handled and causes several false positive for failed schedulability. The robustness of applications against those scenarios is assumed to be taken as a strict design constraint by the software designers and should not cause a critical failure. Additionally, it is also assumed the usage of ARINC664 part 7 (also referenced as AFDX), which defines the communication process between end systems where bandwidth and latency are guaranteed, besides being deterministic by definition. With that in mind, the dependencies between applications were not taken into consideration during the schedulability analysis.

The ARINC653 standard defines a hierarchical scheduling scheme as described in chapter 2.1, in which, besides a temporal partition, a second scheduler is applied within the partition level. A constraint from most OS that applies the standard is that only static scheduling for that level, which gives just a few possibilities of choice. This is also reflected in the Schedulability Analysis tools available. To give more freedom to the automation framework, the scheduler chosen for the partitions was the rate monotonic algorithm.

4.5 Allocation Algorithms

To demonstrate the analysis capability of the framework, two additional allocation algorithms were selected, making a total of three different results to be compared. A modified classical min-min approach (CHEN et al., 2013) referred to as User-priority awarded load balancing improved min-min scheduling algorithm (PA-LBIMM) and an additional one using the analytical hierarchy process (GAWALI; SHINDE, 2018) as a means to prioritize the tasks during the process. Both are from the cloud computing area of research.

4.5.1 PA-LBIMM Algorithm Adaptations

The basis of PA-LBIMM is to take the min-min algorithm and improve upon its weakness, as identified by (CHEN et al., 2013): load unbalance and user-priority demands (VIP users). Considering it is coming from a different type of target architecture, a few adaptations are necessary to be used by the framework.

At the start, the authors propose the tasks to be divided into two groups: Ordinary and VIP. Naturally, such categorization does not exist in typical avionics systems, however, we can take the criticality of processes as a VIP level measurement. In this case, up to 5 groups can be identified, one for each Design Assurance Level (DAL) level (e.g. A, B, C, D, and E).

Then, taking the groups in order of priority, the algorithm selects the task that cost the minimum execution time and assigns it to the resource that gives the minimum expected completion time. In order to choose the task, the period and the WCET are used to perform the execution time calculation (see equation 4.3). On top of that, taking the assumption all processing elements in the case study platform perform the same, there will be no difference in execution time between the resources, therefore this particular part can be simplified to assign tasks to the next available resource.

$$E_t = 1/(T \times WCET) \tag{4.3}$$

Data: $Applications, Platform$
Result: $NotAllocatedApplications, Platform$
1 $sort(Applications)$ based on E_t;
2 $sort(Applications)$ based on DAL_Level;
3 **for** *every application in Applications* **do**
4 allocate application to the next available partition;
5 **if** *allocation failed* **then**
6 populate Not Allocated Applications List;
7 **end**
8 **end**
9 **for** *every allocated application in the Platform (sorted by partition load ratio* **do**
10 reallocate application into the partition that minimizes the Platform Balance Ratio;
11 **end**

Algorithm 2: Adapted PA-LBIMM

To calculate the balance ratio, we first need to define the partition load ratio (l_p

and L_p for a full set of partitions):

$$l_p = \frac{\sum_{i=1}^{n} E_t(i) + j}{mF} \tag{4.4}$$

Being n the number of tasks to be allocated, j the preemption jitter, $E_t(n)$ the execution time of the n^{th} task assign to the partition, and mF the partition minor frame.

Then, the platform balance ratio can be defined by the coefficient of variation of all partitions load ratios:

$$B = c_v = \frac{\sigma(L_p)}{\mu(L_p)} \tag{4.5}$$

Being σ the standard deviation of all partitions load ratio and μ the average.

4.5.2 AHP Algorithm

The proposed heuristic (GAWALI; SHINDE, 2018) is based on the analytical hierarchy process (SAATY, 1978) to better prioritize the applications before assigning them to the hardware set.

In summary, the process comprises an extensive comparative analysis between a pair of tasks based on the preference table 4.1. The comparison is based on the system designer and/or system integrator's judgment, and one of the criteria during this evaluation was the task execution time. The result is a $n \times n$ matrix, being n the number of tasks. That is an activity that should be performed by the engineers themselves, and cannot be automated.

The matrix becomes the input for the AHP calculation which gives a ranking value for every task under consideration (SAATY, 1978).

The list of tasks sorted by the ranking given by the AHP calculations becomes the input for allocation.

The balancing algorithm is not well defined by the author, therefore a few common approaches were chosen and will be compared during the analysis of the results (Chapter 5): Round-Robin (algorithm 3), Resource aware (algorithm 4) and finally the same as applied on the PA-LBIMM resulting in algorithm 5.

Data: $Applications, Platform$
Result: $NotAllocatedApplications, Platform$
1 $sort(Applications)$ based on AHP rankings;
2 **for** *every application in Applications* **do**
3 allocate application to the next partition iteratively;
4 **if** *allocation failed* **then**
5 populate Not Allocated Applications List;
6 **end**
7 **end**

Algorithm 3: Adapted AHP algorithm with Roundrobin balancing mechanism

Data: $Applications, Platform$
Result: $NotAllocatedApplications, Platform$
1 $sort(Applications)$ based on AHP rankings;
2 **for** *every application in Applications* **do**
3 allocate application to the least loaded partition available;
4 **if** *allocation failed* **then**
5 populate Not Allocated Applications List;
6 **end**
7 **end**

Algorithm 4: Adapted AHP algorithm

Data: $Applications, Platform$
Result: $NotAllocatedApplications, Platform$
1 $sort(Applications)$ based on AHP rankings;
2 **for** *every application in Applications* **do**
3 allocate application to the next available partition;
4 **if** *allocation failed* **then**
5 populate Not Allocated Applications List;
6 **end**
7 **end**
8 $balance(Platform)$

Algorithm 5: Adapted AHP algorithm with post balance mechanism

Table 4.1 – Numerical saaty preferences

Numerical rating	*Judgment preference*
9	Extremely preferred
8	Very strongly to extremely preferred
7	Very strongly preferred to preferred
6	Strongly to very strongly
5	Strongly preferred
4	Moderately to strongly preferred
3	Moderately preferred
2	Equally to moderately preferred
1	Equally preferred

Source: (SAATY, 1978)

5 RESULTS

The case study system described in chapter 3.3 was used to demonstrate the framework and evaluate the chosen allocation algorithms. The table 3.6 shows the input platform, and table 3.5 the set of applications, except for *DisplayX_Server* and ActuatorX-Control, as they have their own set of specialized computers and hardware which are not part of the evaluated platform.

According to the ARP4751 standard, the reliability requirements can be depicted from the table 3.4. Function in which the severity condition was classified as *Catastrophic*, shall comply with a probability of failure per flight hour of $10^{-}9$. For *Hazardous*, the requirement is $10^{-}7$, for *Major*, $10^{-}5$ and for *Minor*, $10^{-}3$. Taking these figures into consideration, we can evaluate if the output from the framework is compliant with the requirements.

The table 5.1 shows one of the outcomes of the framework, which is the calculated failure rates for every function within the system. As presented in chapter 3.3, the requirements for failure rates per flight hour for *HMI*, *System Monitoring*, *Flight Control*, *Reconfiguration*, *Navigation* and *Auto Pilot* are respectively 10^{-8}, 10^{-8}, 10^{-9}, 10^{-9}, 10^{-5} and 10^{-5}.

For the HMI function, the initial algorithm and the adapted PA-LBIMM failed to produce a compliant system. The output failure rate is 2 orders of magnitude behind the actual desired value. For System Monitoring, the initial algorithm and the AHP with a resource-aware balance mechanism failed to create a valid system. For the Flight Control and Reconfiguration functions the result, the initial algorithm, PA-LBIMM, and the AHP Resource Aware failed to find a possible solution. The algorithms didn't have problems complying with the Navigation and Auto Pilot function requirements.

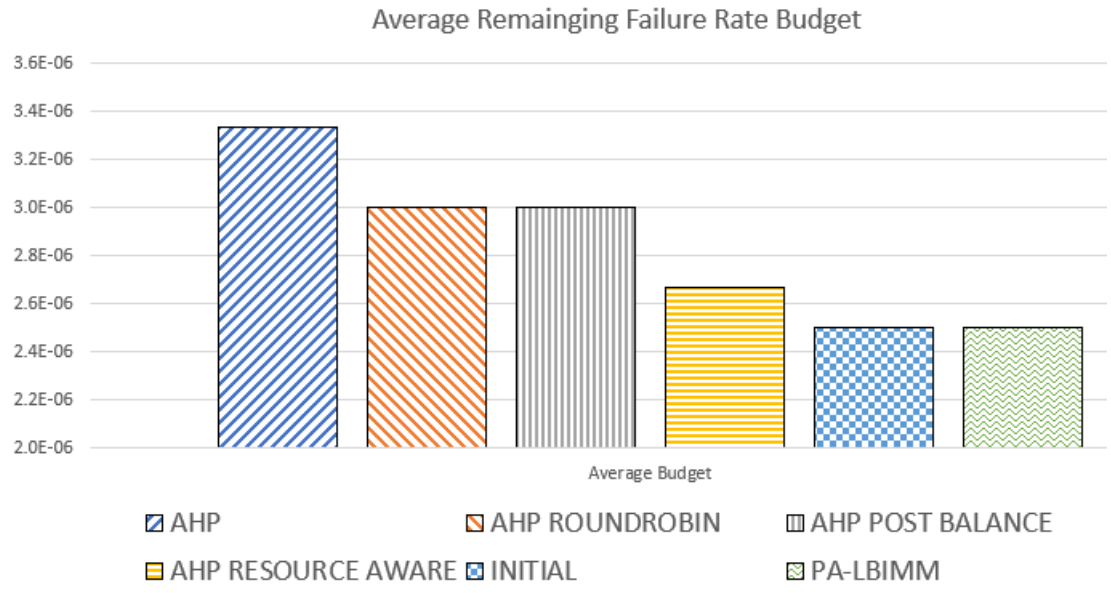
Figure 5.1 brings the aggregated results for each evaluated algorithm. The y-axis represents the average remaining failure rate budget between all functions. This data is important due to the fact the current calculations take into consideration only the failures related to the computer platform, as it is the main subject of analysis of this work. Any other failure event (i.e loss of airspeed sensor for the flight control), would be added to the next steps of the system design, for instance when the system fault tree is being constructed. Therefore, a result with more remaining budget brings more flexibility to the overall architecture, and as a consequence better chances to get to a fully compliant system.

Table 5.1 – Output System - Functions Failure Rate

	Functions					
Algorithm	HMI	System Monitoring	Flight Control	Reconfiguration	Navigation	Auto Pilot
Initial Algorithm	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-12}
PA-LBIMM	10^{-6}	10^{-12}	10^{-6}	10^{-6}	10^{-6}	10^{-6}
AHP (Not Balanced)	10^{-12}	10^{-12}	10^{-12}	10^{-12}	10^{-12}	10^{-12}
AHP Roundrobin	10^{-12}	10^{-12}	10^{-12}	10^{-12}	10^{-6}	10^{-6}
AHP Resource Aware	10^{-12}	10^{-6}	10^{-6}	10^{-6}	10^{-6}	10^{-12}
AHP Post Balance	10^{-12}	10^{-12}	10^{-12}	10^{-12}	10^{-6}	10^{-6}

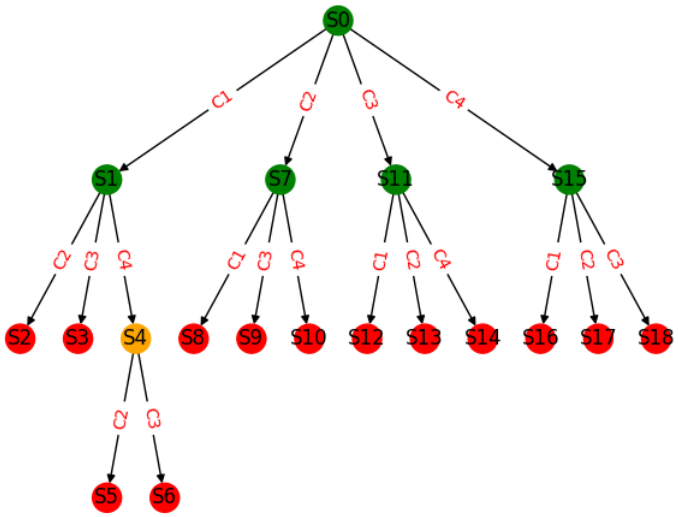
Source: author

Figure 5.1 – Average Remaining Failure Rate Budget



Source: author

Figure 5.2 – Reconfiguration Graph - AHP with no balance mechanism



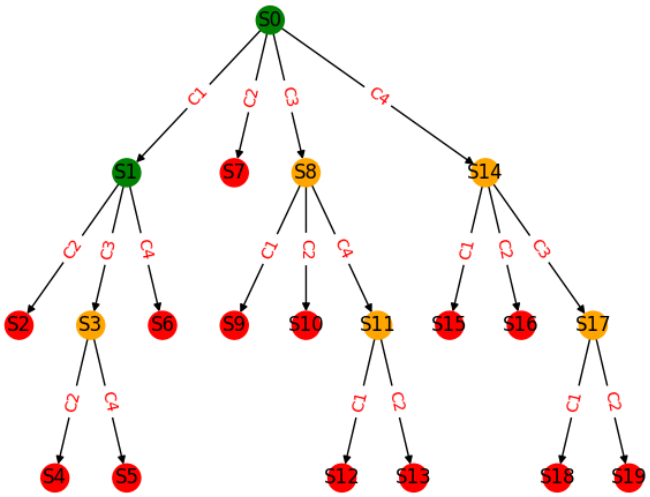
Source: author

The AHP with no balancing mechanism is the best performer among all evaluated algorithms. Figure 5.2 shows its RSD generated by the framework. Green circles mean fully schedulable states, yellow circles mean degraded system states, and red circles mean not schedulable states. The degraded status is assigned when some of the non-critical applications were not possible to be allocated to the platform, however, all the critical ones are still part of the system. One particular aspect of this diagram that shows partially the reasons for a good result for the simpler version of the AHP, is the fact it finds fully schedulable scenarios for all states generated after every single computer failure, while all the other algorithms do not achieve a such solution.

The adapted PA-LBIMM is the worst performer of all, together with the initial algorithm. The generated RSD (figure 5.3) shows that if C2 fails after the initial state, the system would directly transition to a complete failure state, with critical components not part of the platform allocation.

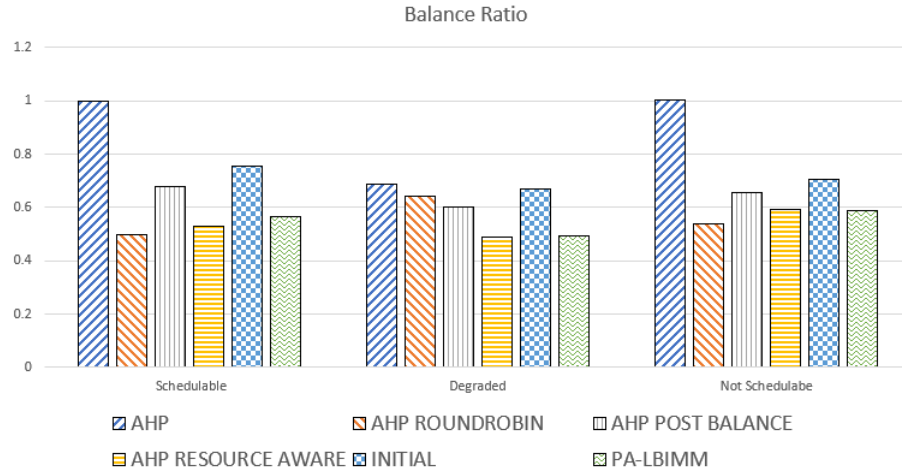
Another particular detail stands out from the remaining budget chart: the balance mechanism. Especially the resource-aware (load dependent) variants, produce the worst results. That can be explained by the design constraints that must be followed during a system reconfiguration which is described in chapter 3.2.8. A critical application is not allowed to be reallocated due to a computer failure if it was not already part of the failed computer. That means a balanced system gives less flexibility to the algorithm to allocate the remaining applications to the platform. Figure 5.4 shows the average balance ratio of all states produced by each algorithm, where the bigger the value, the less the platform is balanced. Especially in the degraded states, the results show that the more balanced the

Figure 5.3 – Reconfiguration Graph - PA-LBIMM



Source: author

Figure 5.4 – Balance Ratio



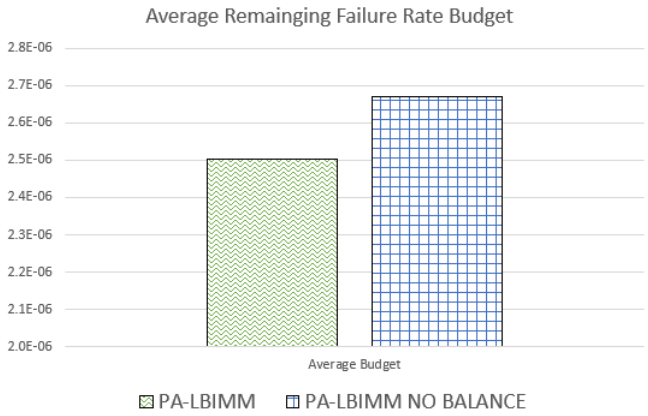
Source: author

platform is, the worst the final result.

In addition, a follow-up analysis was performed to get more details about this effect. A modified PA-LBIMM with no balancing mechanism was introduced to the framework by simply building an inherited class from *AlgorithmMinMin* class (see class diagram in figure 4.5) which overloads the *_post_run* method, removing the balance mechanism. The result of the failure rate budget is in the figure 5.5 (where the bigger the value, the less the platform is balanced), and it shows that the removal of the platform balancing mechanism improves the overall result of the algorithm. However, in this particular case, it did not change the outcome of the PA-LBIMM, which was still not compliant.

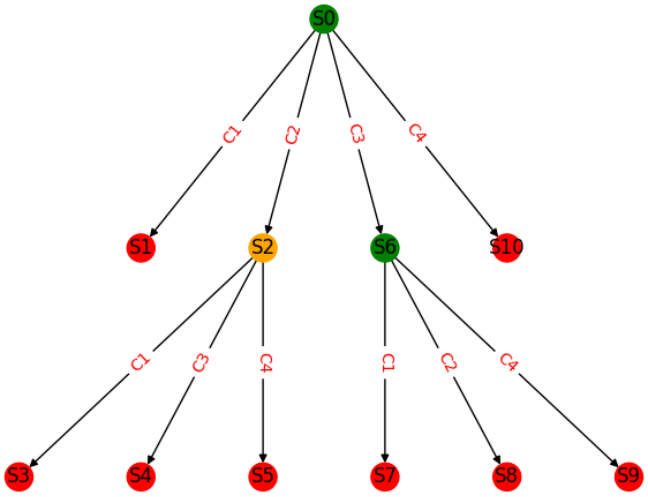
One characteristic of the AHP-based algorithm is the preference judgment of the engineer responsible for the overall system. For this particular exercise, the criticality

Figure 5.5 – Failure rate budget comparison for PA-LBIMM balanced and unbalanced



Source: author

Figure 5.6 – Reconfiguration Graph - AHP Based With Resource Aware Balance Strategy



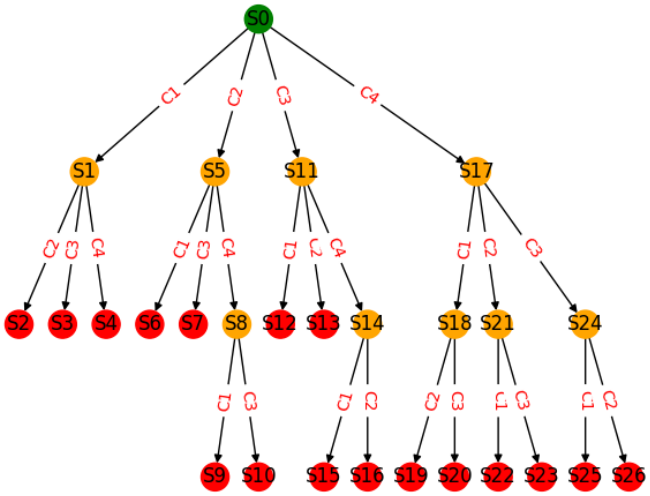
Source: author

was not the main source for the ranking, but a higher WCET of an application had a bigger weight during the comparison scores evaluation. That led to an inverse behavior if compared to the PA-LBIMM, where the processes with the least time cost would have priority during the allocation. According to the results presented in the figure 5.1, it is possible to conclude that algorithms that shift the priority to time costly applications and do not perform any balance attempt tend to perform better when building a reconfigurable system with the previously presented design constraints.

To present the entire solution space, the RSD for the initial algorithm, for the AHP based algorithm with a resource aware balance mechanism and with round robin balance strategy are shown in figure 5.8, 5.6 and 5.7

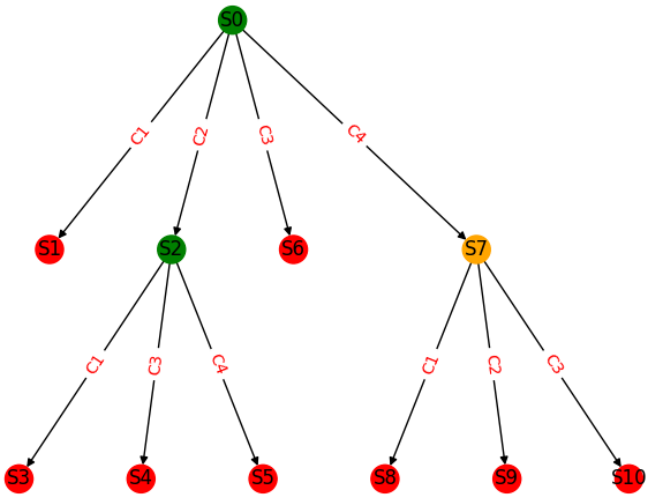
This exercise demonstrates the flexibility that the framework brings to the systems integrator and responsible engineers. New algorithm variants were easily incorporated

Figure 5.7 – Reconfiguration Graph - AHP Based With Round Robin Balance Strategy



Source: author

Figure 5.8 – Reconfiguration Graph - Initial Algorithm



Source: author

to evaluate specific aspects of the initial set of results, and quick conclusions could be drawn. A typical analysis performed during this work took around 8 minutes to give a result. Most of that time is spent running the schedulability analysis, and it can vary depending on the complexity of the system under development. For instance, using the PA-LBIMM algorithm and CHEDDAR, the framework built an RSD with 7 schedulable system states. To check the schedulability of the initial state (*S0*), which has 21 different applications, 4 computers, and 13 partitions, the tool takes 62 seconds.

A hypothetical scenario was created to stress the framework's capabilities and evaluate its performance when run against a larger input. The input platform was increased to 12 computers, 32 partitions, and 69 applications. The time needed by CHEDDAR to evaluate the initial deployment (worst case) was 196 seconds. The number of states generated by the design automation is close to 1000, which results in 50 hours of execution in a 2018 laptop with 8th generation core i7 (i7-8550U @ 1.80GHz, 16GB of ram).

Obviously, this is would not be acceptable in an agile environment, however, several optimizations can be done and were not the scope of this work. Besides running in a proper modern server machine, which would reduce the time significantly already, the usage of parallel processing can be explored by running several instances of cheddar for different possible system state candidates. In addition, the depth of the RSD was not limited, giving an unrealistic tree structure of 9 levels.

6 CONCLUSION

In this work, a reconfiguration approach to deal with fault management and its associated schedulability analysis is presented. While the model checking guarantees that all foreseen situations are evaluated in determining that the design time timing constraints are effectively satisfied, it proved itself a non-viable solution due to the rapid state exploration leading to non-conclusive results more than often. On the other hand, scheduling simulation is a more suitable tool to evaluate real-timing requirements in a dynamic design exploration environment compared to formal verification, as it can be easily iterated over different inputs.

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. It shows that the proposed methodology provides a feasible design flow for avionics systems to be further evaluated in industrial settings.

Moreover, the framework proof of concept for design automation created an agile foundation for system integrators and designers to evaluate different algorithms and approaches and their results towards the initial requirements.

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 to 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.

From the scheduling simulation point of view, some constraints brought by CHEDDAR limited the correct evaluation of feasible states. However, the tool is actively being improved and the limitations which were evident during this work were presented to the developers which took the suggestion as an interesting feature to be added and are evaluating the impacts of such improvement in the tool. In case of a future release, the framework could be changed, eliminating partially the limitations for application dependencies.

Another area that can be explored is the use of multi-core systems. Modern and more powerful processing units rely on multi-core technology, and studies are showing how to apply current standards to this scenario and adapt the IMA architecture. Those aspects can be introduced in the framework as design constraints, creating another ab-

straction for system designers.

Several aspects must be further evaluated when implementing the proposed reconfiguration architecture. One of the key functions is the health monitoring mechanism that is necessary for the RMU and RMC to run as expected. The technical details of such functionality can be explored further and experiments could bring valuable insights of the reconfiguration system.

REFERENCES

Airforce Technology. **Embraer completes first batch deliveries of F-5EM fighter to Brazil**. 2009. Accessed 2018-12-02. Disponível em: <https://www.airforce-technology.com/news/newsembraer-upgraded-f-5em-tiger-fighter-brazil/>.

ANNIGHOEFER, B.; REIF, C.; THIELECK, F. Network topology optimization for distributed integrated modular avionics. **AIAA/IEEE Digital Avionics Systems Conference - Proceedings**, IEEE, p. 4A11–4A112, 2014.

ANNIGHOFER, B.; THIELECKE, F. Multi-objective mapping optimization for distributed Integrated Modular Avionics. **AIAA/IEEE Digital Avionics Systems Conference - Proceedings**, p. 1–13, 2012.

ATITALLAH, R. B. et al. FPGA-Centric design process for avionic simulation and test. **IEEE Transactions on Aerospace and Electronic Systems**, v. 54, n. 3, p. 1047–1065, June 2018. ISSN 0018-9251.

AVSI. **Motivation for Advancing the SAVI Program**. 2009. Accessed 2022-05-17. Disponível em: <https://savi.avsi.aero/about-savi/savi-motivation/>.

AVSI. **Summary Final Report - Produced under the System Architecture Virtual Integration (SAVI) Proof-Of-Concept Program AFE 58**. [S.l.], 2009. v. 2009.

BAIER, C.; KATOEN, J. P. **Principles of model checking**. Cambridge, MA, USA: MIT Press, 2008. ISBN 9780262026499.

BALIS, C.; BERENSON, D.; JOVOVIC, A. Top 5 trends in military aviation. **The European Security and Defence Union**, June 2013.

BIEBER, P. et al. Preliminary design of future reconfigurable IMA platforms. **SIGBED Rev.**, ACM, New York, NY, USA, v. 6, n. 3, p. 7:1–7:5, October 2009. ISSN 1551-3688.

CHEN, H. et al. User-priority guided min-min scheduling algorithm for load balancing in cloud computing. In: **2013 National Conference on Parallel Computing Technologies (PARCOMPTECH)**. [S.l.: s.n.], 2013. p. 1–8.

CUI, Y.; SHI, J.; WANG, Z. Backward reconfiguration management for modular avionic reconfigurable systems. **IEEE Systems Journal**, v. 12, n. 1, p. 137–148, March 2018. ISSN 1932-8184.

DAVID, A. et al. Model-based framework for schedulability analysis using UPPAAL 4.1. p. 1–32, November 2009.

DJIKA, B. et al. Work-in-progress: Models and tools to detect real-time scheduling anomalies. In: **2021 IEEE Real-Time Systems Symposium (RTSS)**. [S.l.: s.n.], 2021. p. 540–543.

Federal Aviation Administration. **AC - SYSTEM SAFETY ANALYSIS AND ASSESSMENT FOR PART 23 AIRPLANES**. [S.l.], 2011.

FEILER, P. H.; GLUCH, D. P. **Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language**. New York, NY, USA: Addison-Wesley Professional, 2012. ISBN 9780321888945.

FOHLER, G. et al. Evaluation of dreams resource management solutions on a mixed-critical demonstrator. **9th European Congress on Embedded Real Time Software and Systems (ERTS)**, v. 12, n. 1, p. 1–10, January 2018.

GAWALI, M.; SHINDE, S. Task scheduling and resource allocation in cloud computing using a heuristic approach. **Journal of Cloud Computing**, v. 7, 02 2018.

GIGANTE, G.; PASCARELLA, D. Formal methods in avionic software certification: The DO-178C perspective. In: MARGARIA, T.; STEFFEN, B. (Ed.). **Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 205–215. ISBN 978-3-642-34032-1.

GONG, S. et al. Adaptive multivariable control for multiple resource allocation of service-based systems in cloud computing. **IEEE Access**, v. 7, p. 13817–13831, 2019.

HARBOUR, M. G. et al. Mast: Modeling and analysis suite for real time applications. In: IEEE. **Proceedings 13th Euromicro Conference on Real-Time Systems**. [S.l.], 2001. p. 125–134.

HOLLOW, P.; MCDERMID, J.; NICHOLSON, M. Approaches to certification of reconfigurable IMA systems. **the International Council on Systems**, p. 1–8, 2000.

HOUSSEYNI, W. et al. Multiagent architecture for distributed adaptive scheduling of reconfigurable real-time tasks with energy harvesting constraints. **IEEE Access**, v. 6, p. 2068–2084, 2018.

JIANG, H. Key findings on airplane economic life. **Boing white paper**, March 2013. Disponível em: <https://aviation.report/view-resource.aspx?id=11>.

JÖZWIAK, L.; NEDJAH, N. Modern architectures for embedded reconfigurable system - a survey. **Journal of Circuits, Systems, and Computers**, World Scientific Publishing, v. 18, n. 2, p. 209–254, 2009.

KLEINBERG, J.; TARDOS, E. **Algorithm Design**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN 0321295358.

LARSEN, K. G.; PETTERSSON, P.; YI, W. UPPAAL in a nutshell. **International Journal on Software Tools for Technology Transfer**, Springer Berlin / Heidelberg, v. 1, n. 1-2, p. 1, March 1997. ISSN 1551-3793.

LIU, C.; LAYLAND, J. **Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment**. 1973.

LÖFWENMARK, A.; NADJM-TEHRANI, S. Fault and timing analysis in critical multi-core systems: A survey with an avionics perspective. **Journal of Systems Architecture**, v. 87, p. 1 – 11, 2018. ISSN 1383-7621.

MCMILLAN, K. L. **Symbolic Model Checking**. [S.l.]: Springer New York, NY, 1993. ISBN 9781461363996.

MONTANO, G.; MCDERMID, J. Human involvement in dynamic reconfiguration of Integrated Modular Avionics. **AIAA/IEEE Digital Avionics Systems Conference**, n. 978, p. 1–13, 2008.

MORGAN, M. Integrated modular avionics for next generation commercial airplanes. **IEEE Aerospace and Electronic Systems Magazine**, v. 6, n. 8, p. 9–12, 1991.

OSATE. **OSATE 2: Open Source AADL Tool Environment**. Accessed 2018-12-02. Disponível em: <https://wiki.sei.cmu.edu/aadl/>.

PAGETTI, C. et al. The ROSACE case study: From simulink specification to multi/many-core execution. In: **IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)**. [S.l.: s.n.], 2014. p. 309–318. ISSN 1545-3421.

PARKINSON, P.; KINNAN, L. Safety-critical software development for integrated modular avionics. 05 2015.

PORCARELLI, S. et al. A Framework for Reconfiguration-Based Fault-Tolerance in Distributed Systems. In: LEMOS, R. D.; GACEK, C.; ROMANOVSKY, A. (Ed.). **Architecting Dependable Systems**. [S.l.]: Springer-Verlag, 2004. p. 167–190.

PRISAZNUK, P. Integrated modular avionics. In: **Proceedings of the IEEE 1992 National Aerospace and Electronics Conference@m**$_N AECON 1992.[S.l. : s.n.], 1992.p.39 - -45vol.1.$

PRISAZNUK, P. J. Arinc 653 role in integrated modular avionics (ima). In: **2008 IEEE/AIAA 27th Digital Avionics Systems Conference**. [S.l.: s.n.], 2008. p. 1.E.5–1–1.E.5–10.

RTCA. **DO-178C - Software Considerations in Airborne Systems and Equipment Certification**. [S.l.], 2012. v. 2012.

SAATY, T. L. Modeling unstructured decision problems — the theory of analytical hierarchies. **Mathematics and Computers in Simulation**, v. 20, n. 3, p. 147–158, 1978. ISSN 0378-4754. Disponível em: <https://www.sciencedirect.com/science/article/pii/0378475478900642>.

SAE. **ARP4761 - Guideline and Method for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment**. [S.l.], 1996. v. 1996.

SAE. **ARP4754A - Guideline for Development of Civil Aircraft and Systems**. [S.l.], 2010. v. 2010.

SINGHOFF, F. et al. Investigating the usability of real-time scheduling theory with the cheddar project. **Real-Time Systems**, v. 43, p. 259–295, 11 2009.

ZHANG, Q.; WANG, S.; LIU, B. Approach for integrated modular avionics reconfiguration modelling and reliability analysis based on AADL. **IET Software**, v. 10, n. 1, p. 18–25, 2016. ISSN 1751-8806.

ZHOU, Q. et al. An AADL-based design for dynamic reconfiguration of DIMA. In: **IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC)**. [S.l.: s.n.], 2013. p. 4C1–1–4C1–8. ISSN 2155-7195.

APPENDIX A — RESUMO EXPANDIDO

A reconfiguração de sistemas embarcados distribuídos em tempo real consiste em alterar ou modificar subsistemas e/ou configurações de subsistemas para melhor servir um determinado propósito (JÖZWIAK; NEDJAH, 2009). Em um sistema de aviônicos, as mudanças de modo do sistema são usadas para realizar adaptações às mudanças das condições operacionais de voo. Enquanto os modos são predeterminados, sua realização pode ser por meio de reconfigurações. A reconfiguração pode ser aplicada para tolerar falhas que podem causar a perda de uma determinada função crítica em resposta a uma mudança externa ou a pedido de um usuário do sistema ou mesmo a um evento temporizado em uma aplicação. A pesquisa de Löfwenmark et al. (LÖFWENMARK; NADJM-TEHRANI, 2018) mostra que arquiteturas tolerantes a falhas continuam sendo uma importante área de pesquisa, e a combinação de tolerância a falhas com garantias de tempo ainda não foi resolvida, por exemplo, na presença de arquiteturas multicore.

Quando um componente do sistema falha, uma plataforma de aviônicos reconfigurável move as funcionalidades, que foram alocadas anteriormente no componente com falha, para outro componente do sistema disponível. Tal esquema de reconfiguração, além de aumentar a confiabilidade, também pode ser benéfico em termos de capacidade de evolução ao longo do ciclo de vida da aeronave.

A vida útil das aeronaves comerciais vem aumentando desde o final do século 20 até o presente século 21 (JIANG, 2013) e agora atingiu uma certa estabilidade. Além disso, espera-se que o mercado de Manutenção, Reparo e Revisão (MRO) produza uma forte demanda futura, pois as Forças Aéreas militares em todo o mundo recorrentemente decidem atualizar aeronaves legadas em vez de adquirir novas plataformas (BALIS; BERENSON; JOVOVIC, 2013), o que dá às frotas militares um aumento de vida útil. No Brasil, por exemplo, uma revisão recente trouxe a uma frota de aeronaves dos anos 70 a capacidade de estender sua vida útil além de 2020 (Airforce Technology, 2009).

Projetos de aeronaves, sejam plataformas novas ou reformuladas, aumentaram o tempo de desenvolvimento e, portanto, os custos substancialmente nos últimos anos. A obsolescência da tecnologia aviônica que ocorre antes da vida útil da estrutura principal da aeronave também é uma causa da tendência do mercado de MRO. A flexibilidade de reconfiguração pode aliviar parcialmente tais problemas. Entregas antecipadas com recursos básicos podem ser realizadas e funcionalidades mais avançadas podem ser incorporadas ao sistema alterando a configuração.

Dado o cenário descrito acima, este trabalho propõe uma arquitetura reconfigurável distribuída na qual um agente global e agentes locais cooperam para supervisionar a transição de aplicações de módulos com falha para módulos funcionais. As reconfigurações viáveis determinadas em tempo de projeto são armazenadas no sistema para serem utilizadas pelos agentes, que então mantêm os computadores em uma configuração previamente definida em uma situação prevista.

A reconfiguração de um subsistema não afeta o resto do sistema de forma alguma. Em outras palavras, as restrições de tempo real especificadas originalmente ainda seriam satisfeitas. A sequência de passos necessários para a conclusão de uma reconfiguração deve ser atômica, no sentido de que devem ser inteiramente bem-sucedidas ou descartadas. No caso de uma reconfiguração ser abortada, a operação do sistema aviônico não deve ser afetada de forma alguma. Em ambos os casos é importante destacar que assumi-se que as falhas ocorrem uma de cada vez.

Ter uma transição bem-sucedida para o estado reconfigurado devido a uma falha não garante a viabilidade de tal estado. Portanto, fica clara a necessidade de uma verificação final para avaliar a correção dos sistemas em tempo real configurados para desempenhar a função desejada. Para isso, a análise de escalonabilidade é utilizada para verificar se todas as tarefas do sistema atenderão às suas restrições de tempo.

Diferentes abordagens podem ser usadas para realizar essa verificação. A verificação de modelo (*model checking*) (BAIER; KATOEN, 2008) pode ser usada para determinar as reconfigurações viáveis, levando em consideração todas as possíveis sequências de etapas necessárias. A partir de uma especificação, fornecida por uma linguagem de modelos voltada para análise de arquitetura, Architecture Analysis and Design Language (AADL) (FEILER; GLUCH, 2012), uma das abordagens propostas inclui a criação de uma rede de autômatos (LARSEN; PETTERSSON; YI, 1997), representando os aspectos de temporização de um sistema aviônico , a fim de realizar a análise de escalonabilidade de cada reconfiguração possível. Isso é feito avaliando propriedades lógicas temporais específicas no rastreamento temporizado das tarefas do sistema aviônico e observando seus prazos. Alternativamente, a abordagem de simulação de escalonamento também é usada aplicando os algoritmos de escalonamento durante um período de tempo para calcular o escalonamento do sistema (SINGHOFF et al., 2009).

A análise de escalonabilidade garante a previsibilidade do sistema após cada reconfiguração e facilita a aprovação da aeronavegabilidade pelas autoridades certificadoras.

Levando em consideração os sistemas estão ficando exponencialmente mais com-

plexos a cada ano (AVSI, 2009b). Durante a década de 1980, um projeto de avião típico tinha menos de 1 milhão de linhas de código de software (SLOC), enquanto a partir de 2000 esse número explodiu. Por exemplo, o SLOC relatado para o programa F35 foi de cerca de 24 milhões (AVSI, 2009a). Esse cenário exige mais abstrações e automação de design durante o desenvolvimento. Os engenheiros de software devem se concentrar em seu próprio escopo, pois não são capazes de lidar com a complexidade sempre crescente dos sistemas de computador subjacentes e suas restrições de projeto. Durante este trabalho a automação de projeto proposta é implementada na forma de um framework de prova de conceito.

A abordagem proposta foi ilustrada usando um exemplo sintético para explicar seus algoritmos. Em seguida, foi aplicado em um estudo de caso de sistema aviônico para mostrar que tamanhos de problemas razoáveis em termos de número de nós, dependências, restrições de criticidade e mapeamentos de software/hardware podem ser tratados. Tambem mostra que a metodologia proposta fornece um process de desenvolvimento de projeto viável para sistemas aviônicos a serem posteriormente avaliados em ambientes industriais.

Além disso, a prova de conceito do framework para automação de projeto criou uma base ágil para integradores e designers de sistemas avaliarem diferentes algoritmos e abordagens e seus resultados em relação aos requisitos iniciais.