Partition Scheduling in Distributed Integrated Modular Avionics

João Gonçalves
joaomfgoncalves@tecnico.ulisboa.pt

Instituto Superior Técnico, Universidade de Lisboa, Portugal

October 2019

Abstract

The Integrated Modular Avionics architecture has replaced federated architectures in the avionic domain, allowing significant weight and power savings and enabling more competitive application development. The resource-sharing nature of this architecture requires robust temporal and spatial segregation between applications, which is achieved by statically scheduling applications on shared avionic hardware. This raises a multiprocessor scheduling problem, automation of which has seen limited progress in industry, but representing a significant challenge for system integration. We propose a mathematical model for the partition scheduling problem associated with an optimization criterion based on system scalability and flexibility, and provide both heuristic and exact methods for its solution based on existing literature.

Keywords: Arinc specification 653, Integrated Modular Avionics, Scheduling, Optimization.

1. Introduction

Avionic systems have traditionally followed a federated architecture, with each component or subsystem having its dedicated hardware and software, in what is defined as Line Replaceable Units (LRUs). Suppliers were responsible for developing both the hardware and software, and supply it as its own self-contained black-box component. This "one function – one computer" concept coupled with redundancy provided high safety and reliability. Applications have guaranteed, deterministic access to processor resources, and Input/Output (I/O) with bounded latency and jitter. Maintenance is straightforward and inexpensive, as LRUs can be readily replaced by equivalent ones. Most importantly, with loosely coupled LRUs, critical functions cannot be impaired by low-criticality functions, and the modularity increases fault containment.

However, the disadvantages of the federated architecture are evident. With a federated architecture, a function being added to the avionic system requires the addition of one of these LRUs, and this quickly escalates the mass, volume, cost and power consumption of the entire avionic system to infeasible amounts. Regarding costs, functions sharing a processor must be certified to the highest criticality level of those functions, and this encourages the usage of many processors with decreased utilization. On the other hand, modern processors have far more capability than a single critical function requires, and this constitutes an inefficient usage of resources [13].

The Integrated Modular Avionics (IMA) paradigm is the aviation industry's response to these problems, whose architecture principle relies on resource sharing between generally unrelated components, including computing resources, power, and communication media, as roughly demonstrated in figure 1. Functions which were previously implemented in isolated LRUs now coexist on shared hardware, and in order to maintain isolation between components, IMA adopts robust time and space partitioning between applications. Space partitioning protects the application's data from corruption by unrelated applications that share the same hardware, and time partitioning ensures the required access to computing resources and communication channels [10]. Due to this, loosely coupled avionic applications are designated as partitions in the context of IMA. Arinc specification 653 defines a standard interface between the Real-Time Operating System (RTOS) and partitions, being an enabler of the IMA architecture.

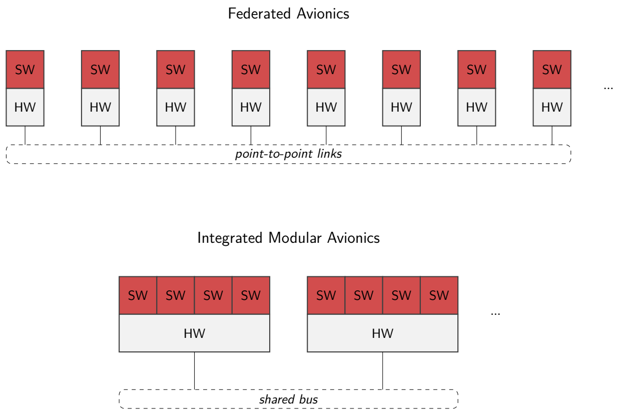


Figure 1: Overview of Federated and Integrated Modular Avionics.

Current developments on the IMA concept aim to further abstract the applications from the hardware. In classic IMA, peripherals such as sensors and actuators are connected directly to the Core Processing Input-Output Modules (CPIOMs), the equivalent of LRUs in this architecture, that often must contain specific hardware to interact with these. The introduction of Remote Data Concentrators (RDCs) allows to connect the peripherals directly to the avionic network. These are hardware devices that carry the necessary drivers for these peripherals and perform their I/O to the avionic network, hence, the CPIOMs, now simply called Core Processing Modules (CPMs), do not require specific hardware and are further standardized. Another advantage of this is the reduced cabling needed, since CPMs are usually confined to the avionics bay, potentially far from the peripherals. These systems are often called Distributed Integrated Modular Avionics (DIMA) since they distribute I/O across the aircraft [16].

Mairaj [13] compares federated and IMA architectures having surveyed 35 projects that underwent the transition. The results show weight reductions of around 50 % for all cases, volume reductions of 30 % to 40 %, power savings of 25 % to 30 %, and Mean Time Between Failures (MTBF) increasing by a factor of 1.4 to 3.8. Nowadays, all new passenger aircraft models employ a variant of this architecture.

1.1. Problem description

An IMA system is composed of several CPMs with each hosting a set of partitions, with a static, cyclic partition schedule. The schedule is static because it is configured at build time, and does not change at runtime. A partition is forcefully stopped at the end of its allocated time window, and control transferred to the next one to ensure fault containment. It is cyclic because a representative unit is continuously repeated while the system is active.

Scheduling partitions in IMA involves decisions in two domains. Firstly, IMA makes it possible that partitions are capable of running on many if not all available CPMs, but the schedule restricts that each must run on only one. Part of the scheduling problem consists in assigning partitions to different processors, verifying their real-time constraints. These can include memory, stack-size, bandwidth, as well as other constraints related to redundancy management. Also, each partition must be allocated time windows to execute while also verifying time segregation with other partitions in the same module, and being able to communicate with other elements in the avionic network.

Partitions are characterized by a period and an execution requirement, which are measured in integer units of time, noting that the highest precision for time measurements in real-time computing systems is the CPU clock period. Partitions are executed strictly periodically, which means that the time separating two consecutive execution windows (or instances, jobs) of the same partition is exactly the partition period. This strict periodicity is common in real-time systems as it is required by control loops for example, but it must be noted that the Arinc 653 standard does not enforce this. What is required is that there is a "periodic processing start", a point in a partition schedule coinciding with the beginning of a window where the internal periodic process scheduling is allowed to start [12]. The execution requirement is taken as the Worst-Case Execution Time (WCET) of the partition and can be provided by the application developer, or determined through testing since it is dependent on the hardware. The smallest unit of repetition of the schedule in one CPM is called the Major Time Frame (MTF), or in other words the hyper-period of the partitions scheduled in that CPM. This is the smallest time window that is indefinitely repeated, which must guarantee that at least one partition time window be allocated to each partition in the duration of one MTF. An example of a partition schedule for one CPM is shown in figure 2.

Given the strict periodicity of partition executions, once an execution window is defined, all subsequent ones are implicitly defined, thus the starting time offset with respect to the MTF is sufficient to fully describe the schedule of a partition. The typical partition scheduling problem is combining this with distributing the partitions among the available CPMs, complying with some constraints. This is part of the problem we aim to solve, and the constraints considered are exclusion, inclusion (or cohabitation), domain, memory, temporal segregation and communication constraints. Furthermore, this is transformed into a Combinatorial Optimization Problem (COP), by the defining an optimization criterion based on flexibility, as introduced in [5], which intuitively consists in providing each partition with room to increase its execution window without interfering with other partitions. In addition to this, we also investigate the possibility of splitting partition execution in multiple windows.

1.2. Related work

The first efforts to automate partition scheduling for IMA is that of Lee et al. [3], which lay a framework for the deployment of this task in the system integration phase. Some approaches attempt to minimize the number of processors used, in the style of the bin-packing problem. See for example Eisenbrand et al. [6] and Eisenbrand et al. [7].

Our approach is most similar to Al Sheikh et al. [9], who aim to minimize the worst-case scalability potential of every partition, but they do not consider communication constraints. They present a Mixed Integer Linear Programming (MILP) formulation which fails to provide solutions in acceptable time, and develop a heuristic based on Game Theory. The same heuristic is improved in Pira and Artigues [15].

Other methodologies include that of Beji et al. [11] who use Satisfiability Modulo Theories (SMT) and aim to minimize integration costs, and that of Blikstad et al. [17], which uses a MILP formulation to model low-level system requirements without a particular optimization goal.

Also in the scheduling domain but not related to IMA or real-time systems, we refer to Pinedo [4]. Some methodologies which inspire the present work include scheduling with generic optimization algorithms, including Simulated Annealing (SA), Tabu-search and genetic algorithms.

1.3. Contributions

The contributions of this paper are as follows:

- A comprehensive mathematical model of the system is provided, containing distribution constraints, which restrict the assignment of partitions to modules, communication constraints, via limiting the delay in a chain of partition executions, and also a multiple window model is introduced, which allows partition execution to be divided in multiple windows, where only some of them must be scheduled strictly periodically.
- A MILP formulation describes a subset of the overall problem.
- A sequential assignment algorithm and a Constraint Satisfaction Problem (CSP) formulation are de-

veloped to produce an initial assignment of partitions to modules.
- A local optimization algorithm based on Game Theory [9] is used to improve the schedule for a single model, and it is extended to accommodate inter-partition communications and multiple windows.
- Stochastic optimization algorithms are added to complement local search and explore larger portions of the search space.

1.4. Outline

This work was developed in collaboration with GMV, a supplier of an Arinc 653-compliant RTOS named XKY. We have presented the fundamental concepts and the necessity for partition scheduling. Section 2 is dedicated to the mathematical representation of the problem, and a MILP formulation which describes part of the overall problem is included. Section 4 describes the algorithms and strategies developed to heuristically solve generic problem instances, with considerations for computational performance, and section 5 evaluates these methodologies. Section 6 is the conclusion to this work.

2. Problem definition and modelling

Consider a set of N_p partitions $\mathcal{P} = \left\{p_1, p_2, \dots p_{N_p}\right\}$ to be scheduled in N_c modules $\mathcal{C} = \left\{c_1, c_2, \dots c_{N_c}\right\}$. Partitions $p_i \in \mathcal{P}$ are characterized by:

- e_i – execution requirement in units of time, taken as its WCET.
- T_i – period, in units of time.
- s_i – memory requirement, in arbitrary units.

Modules $c_m \in \mathcal{C}$ are characterized by:

- S_m – memory capacity, in the same units as s.
- ε_m – context switching cost, in units of time.

This context switching cost is a time penalty added to the partition execution when it is divided in multiple windows, corresponding to the time taken to restore the execution state.

The assignment of partitions to modules is represented by variables f_i, $\forall p_i \in \mathcal{P}$, which denotes that partition p_i is assigned with the module with index $m = f_i$. For convenience, we also define $\mathcal{P}_m \subseteq \mathcal{P}$ as the subset of partitions scheduled in module c_m. That is,

$$\mathcal{P}_m \equiv \{p_i \in \mathcal{P},\ f_i = m\}. \tag{1}$$

The module MTF is the hyper-period of the partition periods hosted in each module:

$$H_m \equiv \operatorname{lcm}\{T_i\},\ p_i \in \mathcal{P}_m, \tag{2}$$

where lcm denotes the least common multiple operator. Under this, a partition executes $K_i = H_m/T_i$ times, and these individual executions are called jobs (notice K_i is an integer due to the definition of H_m). Let now t_i be the starting offset for partition p_i, such that this partition is scheduled to start at strict periodic instants $t_i + kT_i,\ k = 0, 1, \dots K_i - 1$.

Assuming single execution windows, a partition executes in time windows $[t_i + kT_i,\ t_i + kT_i + e_i]$, $k = 0, 1, \dots K_i$, and temporal segregation requires that these windows do not overlap for partitions in the same module. All timing variables are integers, and in particular periods and durations are strictly positive. Figure 2 shows this nomenclature on a partition schedule.

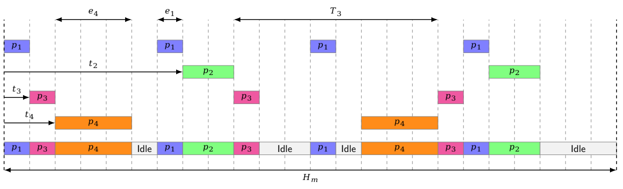


Figure 2: Schedule with annotated timing variables. Partitions have non-harmonic periods.

2.1. Distribution constraints

The assignment of partition to modules is restricted by constraints which we call distribution constraints.

Exclusion – Two partitions are said to be in exclusion if they cannot be assigned to the same module. This covers, but is not restricted to, the redundancy requirements of an avionic system, where safety-critical functionality must be replicated in different machines. An exclusion constraint between partitions p_i, p_j is denoted by $f_i \neq f_j$.

Inclusion – Two partitions are said to be in inclusion if they must be placed in the same module, which is useful for applications that are tightly coupled. Similarly to exclusion, an inclusion constraint between partitions p_i, p_j is denoted by $f_i = f_j$.

Domain – A partition can only be assigned to a subset of all modules, called the partition's domain. That is the case in some architectures where applications require specific hardware that is only installed in some modules, like peripherals. The domain D_i of a partition p_i is essentially a list of modules it can be assigned to. Formally, $D_i \subseteq \mathcal{C}$ such that:

$$c_m \notin D_i \implies f_i \neq m. \tag{3}$$

Memory – This is the only Knapsack constraint considered. For each module, the sum of the partitions' memory sizes much not exceed that module's memory capacity:

$$\sum_{p_i \in \mathcal{P}_m} s_i \leq S_m. \tag{4}$$

Uniqueness – This constraint simply imposes that each partition is assigned to exactly one module. Using f_i notation, this constraint is implicit, but with the abbreviated $\mathcal{P}_m$ notation it is expressed as:

$$\begin{cases} \mathcal{P}_1 \cup \mathcal{P}_2 \cup \dots \cup \mathcal{P}_{N_c} = \mathcal{P} \\ \forall c_m, c_n :\ \mathcal{P}_m \cap \mathcal{P}_n = \emptyset. \end{cases} \tag{5}$$

2.2. Communication constraints

Inter-partition communications are often represented as processing chains, consisting of some kind of data being treated by successive partitions. One can think of data originating from a sensor or user input, being processed by one or more partitions, then originating a certain response in its final destination [9]. For this problem, we will consider such chains, but limit them to two partitions only, such that the time taken to process the data from its origin in the sender partition to its consumption in the receiver partition is bounded.

A chain linking p_i to p_j is subject to a maximum delay

$E_{i,j}^{\max}$, so we can describe all communication constraints by a matrix $[E^{\max}]$, with entries being infinity when there is no communication between partitions. The chain processing time is denoted $E_{i,j}$, measured from the start of p_i to the end of p_j, and must verify

$$E_{i,j} \le E_{i,j}^{\max}. \tag{6}$$

This definition is agnostic to which jobs actually participate in the chain. If the two partitions have equal periods, then the delay between two consecutive jobs is constant, but if the periods are not equal but still harmonic, then $E_{i,j}$ is defined as the shortest delay, and the chain can occur with a period equal to the larger of the two partition periods. When the two partition periods are non-harmonic, then for simplicity we consider also the smallest delay, and the chain shall be repeated with a period equal to the hyper-period of these two partitions.

If the two communicating partitions are assigned to different modules, the network delay between these two modules must be considered. The network is characterized by a matrix [T] with elements $\tau_{m,n}$ of maximum End-to-End (ETE) delays, which are upper bounds to the actual communication delay between modules c_m, c_n. Communications between partitions in the same module are not subject to network delays, thus we define $\tau_{m,m} = 0,\ \forall m$. We can also assume $\tau_{m,n} \ll T_i,\ \forall m, n, i$, by at least one order of magnitude.

The situation considered here is when the modules run synchronously. Intuitively, this means the two modules' schedules are aligned, and the instant when the message arrives in the second module is known. It can be the case that messages arrive at the destination partition after the instant where it starts to process incoming messages, meaning that the message will only be processed in the next job of this partitions, we say it is delayed for one period. It is also considered that all messages are sent and received in the end and beginning, respectively, of a partition's execution window. The earlier assumption $\tau_{m,n} \ll T_i$ is meant to prevent messages from being delayed for more than one period.

2.3. Multiple window model

For the extended model, it is considered that partition jobs can be divided in more than one window of execution, hence we introduce an extended model with specific constraints such that the real-time requirements of these partitions are verified. Henceforth, this is referred to as "multiple windows".

Consider now that, for each job k of a partition $p_i \in \mathcal{P}_m$, there are $M_{i,k}$ execution windows, with lengths $(e_{i,k,1}, e_{i,k,2}, \dots, e_{i,k,M_{i,k}})$, such that:

$$\sum_{u=1}^{M_{i,k}} e_{i,k,u} = e_i + (M_{i,k} - 1)\varepsilon_m,\ \forall k. \tag{7}$$

The windows are represented as $\lambda_{i,k,u}$, and each has its own offset $t_{i,k,u}$, defined with respect to the hyper-period. Additionally, all windows that compose the partition are denoted by the set Λ_i. Since the first window at each job must be executed strictly periodically, its offset is not independent for all jobs, and is constricted by:

$$t_{i,k,1} = t_{i,1,1} + (k-1)T_i,\ \forall k. \tag{8}$$

The problem is greatly complicated because we require vector variables to completely represent the schedule, in particular because the number of jobs depends on the hyper-period, which is a function of the periods of all partitions assigned to that module, and the number and sizes of each window are also variable.

We require that the partition splitting be done only in predetermined points in order to limit the problem complexity. These set of possible points for splitting is represented for each partition p_i as $\mathcal{B}_i$. Splitting a partition at job k in a subset of preemption points $\mathbf{b} \subseteq \mathcal{B}_i$ yields the window sizes $\mathbf{e}_{i,k} = (b_1, b_2 - b_1 + \varepsilon_m, \dots, b_N - b_{N-1} + \varepsilon_m)$, where we consider without loss of generality that $\mathbf{b}$ has N sorted elements.

The response time of a task is defined as the time taken from the task activation to when the task completes, and clearly, this concept is not relevant if the execution is made in a single window. With more than one window per job, the partition finishes executing in instants $\left\{t_{i,k,M_{i,k}} + e_{i,k,M_{i,k}}\right\}$, which prompts the definition of the response time, r_i, as

$$r_i = \max_k \left\{t_{i,k,M_{i,k}} + e_{i,k,M_{i,k}} - t_{i,k,1}\right\}, \tag{9}$$

restricted by a relative deadline, d_i:

$$r_i \le d_i, \tag{10}$$

which is measured with respect to to the job start, $t_{i,k,1}$. All partitions have the implicit deadline $d_i \le T_i$ to ensure that all windows finish before the next job starts.

Figure 3 sketches the new notation introduced in this section, where the windows corresponding to a subdivision of a job are represented with rounded edges.

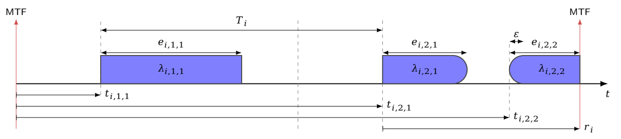


Figure 3: Multiple window execution notation.

2.4. Schedulability

From Korst et al. [2], we can derive a necessary and sufficient condition for two partitions, p_i, p_j, assigned to the same module to not overlap in time:

$$e_i \le mod\left\{t_j - t_i,\ g_{i,j}\right\} \le g_{i,j} - e_j, \tag{11}$$

with $g_{i,j} = \gcd\left\{T_i, T_j\right\}$, and *mod* denoting the modulo operator. We further define $l_{i,j} \equiv mod\{t_j - t_i,\ g_{i,j}\}$, called a latency delay, which represents the minimum delay between any two job starts of the two partitions.

Schedulability can also be analysed with the processor usage fraction, U_m, given by:

$$U_m = \sum_{p_i \in \mathcal{P}_m} \frac{e_i}{T_i}. \tag{12}$$

This can be seen as the percentage of time that a processor is active, and for single core processors the set is clearly not schedulable when $U_m > 1$. For the complete problem, it is infeasible if $\sum_{c_m \in \mathcal{C}} U_m > N_c$.

Using the definition of the latency delay, the chain processing time is given for $p_i \in \mathcal{P}_m$, $p_j \in \mathcal{P}_n$:

$$E_{i,j} = \begin{cases} l_{i,j} + e_j, & \text{if } l_{i,j} - e_i \geq \tau_{m,n} \\ l_{i,j} + e_j + T_j, & \text{otherwise} \end{cases}. \tag{13}$$

2.5. Optimization criterion

The optimization criterion chosen is one that aims to increase flexibility, by providing each partition a potential to increase its execution time. This is accomplished by leaving some idle time after each partition execution windows, and has some benefits: upon system maintenance or modification, one can add functionality to partitions, increasing their execution requirement, without having to recompute a new schedule; it mitigates uncertainty on the determined WCETs; and possibly, it can allow for the usage of slower, cheaper hardware, where the execution requirements would be greater.

The evaluation function used is the α-parameter, which is the maximum factor that scales all partition execution requirements, such that the schedule becomes borderline valid [5]. It can be defined for each module, or for the whole system:

$$\alpha_m = \min\left\{\frac{l_{i,j}}{e_i}\right\}, \forall p_i, p_j \in \mathcal{P}_m,\ i \neq j \tag{14}$$

$$\alpha = \min_{c_m \in \mathcal{C}} \{\alpha_m\}. \tag{15}$$

This yields each partition further execution time in proportion to the original execution requirement, which is appropriate since more complex applications with longer execution requirements are more likely to need to be updated and/or expanded.

Figure 4 illustrates the optimization criterion. Note that in this figure, p_2 actually can increase its execution further, but the critical factor for the α-parameter is p_1. Also note the solution presented is sub-optimal, it is clear that shifting p_2 to the right increases α.

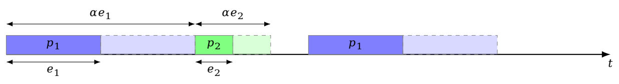


Figure 4: Effect of the α-parameter.

We also define the partition utility, α_i:

$$w(i,j) = \min\left\{\frac{l_{i,j}}{e_i}, \frac{l_{j,i}}{e_j}\right\} \tag{16}$$

$$\alpha_i(t_i) = \min_j\{w(i,j)\}, \tag{17}$$

for $p_j \in \mathcal{P}_m \setminus \{p_i\}$. It represents the module's α-parameter, as a function of the offset t_i, but considering only the partition pairs where p_i is involved, and all other offsets fixed. It is distinct from simply the execution potential for that partition, because it also accounts for the effect on the remaining partitions in that module, as seen in expression 16. For partitions with multiple execution windows we consider that the execution potential is appended only to the last window at every job.

3. MILP formulation

In this section we detail a MILP formulation that models the multiprocessor partition scheduling problem with communication constraints, but does not consider multiple execution windows. The classification as a Mixed Integer Linear Program comes from the fact that the formulation uses both integer variables (offsets and assignments) as well as real variables (the α-parameter) involved in linear constraints. The formulation follows closely that in [5], and uses the natural encoding of binary variables, 1-true and 0-false.

Let $a_{i,m}$ be a binary variable expressing that partition p_i is assigned to module c_m,

$$a_{i,m} = \begin{cases} 1, & p_i \in \mathcal{P}_m \\ 0, & \text{otherwise} \end{cases}. \tag{18}$$

With this reformulation of the partition assignment, The distribution constraints take the following form:

$$\text{Uniqueness}: \quad \forall i \sum_m a_{i,m} = 1. \tag{19a}$$

$$\text{Memory}: \quad \forall m \sum_i a_{i,m} s_i \leq S_m \tag{19b}$$

$$\text{Exclusion}: \quad f_i \neq f_j : \ \forall m\ a_{i,m} \leq 1 - a_{j,m} \tag{19c}$$

$$\text{Inclusion}: \quad f_i = f_j : \ \forall m\ a_{i,m} = a_{j,m} \tag{19d}$$

$$\text{Domains}: \quad \forall i \sum_{c_m \in \{\mathcal{C} \setminus D_i\}} a_{i,m} = 0. \tag{19e}$$

Temporal segregation is ensured by equation 11, which is non-linear due to the modulo function. Linearisation requires the introduction as free variables of the quotient from the division, $q_{i,j} \equiv \left\lfloor \frac{t_j - t_i}{g_{i,j}} \right\rfloor$, which enables rewriting condition 11 as:

$$t_j - t_i - q_{i,j} g_{i,j} \geq \alpha \cdot e_i - Z\left(2 - a_{i,k} - a_{j,k}\right) \tag{20a}$$

$$t_j - t_i - q_{i,j} g_{i,j} \leq g_{i,j} - \alpha \cdot e_j + Z\left(2 - a_{i,k} - a_{j,k}\right). \tag{20b}$$

In this step, we also introduce the α-parameter multiplying the partition executions, and these constraints should only apply to partitions in the same module, thus a "Big-M" constant Z is used to trivially satisfy these inequalities when the partitions are not assigned to the same module. Bounds for q can be found by noting that $t_i \in [0, 1, \ldots, T_i - e_i]$, which gives:

$$\frac{e_i - T_i}{g_{i,j}} - 1 < q_{i,j} \leq \frac{T_j - e_i}{g_{i,j}}. \tag{21}$$

Regarding the communications constraints, for each chain, let the new binary variable $x_{i,j}$ denote that the chain being delayed for one period. This is used to eliminate the branching in equation 13, having:

$$l_{i,j} + e_j + x_{i,j} T_j \leq E^{\max}_{i,j}. \tag{22}$$

The network delay affecting the two partitions is denoted by the auxiliary variable $\hat{\tau}_{i,j}$, defined as:

$$\hat{\tau}_{i,j} \equiv \sum_{c_m, c_n \in \mathcal{C}} a_{i,m} a_{j,n} \tau_{m,n}, \tag{23}$$

but this is non-linear in terms of our free variables. Linearisation is done by introducing new binary variables $y_{i,j,m,n} \equiv a_{i,m} \wedge a_{j,n}$, which yields:

$$\hat{\tau}_{i,j} = \sum_{c_m, c_n \in \mathcal{C}} y_{i,j,m,n} \tau_{m,n}. \tag{24}$$

Finally, when a chain is not delayed for one period, then it must verify $l_{i,j} + e_i - \hat{\tau}_{i,j} \geq 0$, so we introduce

$$l_{i,j} + e_i - \hat{\tau}_{i,j} + x_{i,j} Z \geq 0, \tag{25}$$

where again, the "Big-M" constant Z is used to ignore this constraint when $E_{i,j}^{\max}$ is respected in equation 22 with the chain being delayed one period.

The full model is:

$$
\begin{aligned}
\max \quad & \alpha \\
\text{s.t.} \quad & 0 \le \alpha \le \min_{p_i \in \mathcal{P}} \left\{ \frac{T_i}{e_i} \right\} \\
& \sum_{c_m \in \mathcal{C}} a_{i,m} = 1; \; 0 \le t_i \le T_i - e_i \\
& \forall c_m \in \{\mathcal{C} \setminus D_i\}: \; a_{i,m} = 0; \; \sum_{p_i \in \mathcal{P}} a_{i,m} s_i \le S_m \\
& f_i \neq f_j, \; \forall m: \; a_{i,m} \le a_{j,m} \\
& f_i = f_j, \; \forall m: \; a_{i,m} = a_{j,m} \\
& j > i: \; t_j - t_i - q_{i,j} g_{i,j} \ge \alpha e_i - \\
& - Z \left(2 - a_{i,m} - a_{j,m}\right) \\
& j > i: \; t_j - t_i - q_{i,j} g_{i,j} \le -g_{i,j} \alpha e_j - \\
& - Z \left(2 - a_{i,m} - a_{j,m}\right) \\
& \frac{e_i - T_i}{g_{i,j}} \le q_{i,j} \le \frac{T_j - e_i}{g_{i,j}} \\
& 0 \le t_j - t_i - q_{i,j} g_{i,j} \le g_{i,j} \\
& t_j - t_i - q_{i,j} g_{i,j} + e_j + x_{i,j} T_j \le E_{i,j}^{\max} \\
& t_j - t_i - q_{i,j} g_{i,j} + e_i - \sum_{c_m, c_n \in \mathcal{C}} \left(y_{i,j,m,n} \tau_{m,n} \right) + \\
& + x_{i,j} Z \ge 0 \\
& y_{i,j,m,n} \ge a_{i,m} + a_{j,n} - 1 \\
& y_{i,j,m,n} \le a_{i,m}; \; y_{i,j,m,n} \le a_{j,n} \\
& a_{i,m} \in \{0,1\}, \; t_i \in \mathbb{Z}, \; q_{i,j} \in \mathbb{Z} \\
& x_{i,j} \in \{0,1\}, \; y_{i,j,m,n} \in \{0,1\}
\end{aligned}
$$

4. Methodology

This section describes the heuristic methods developed to solve the problem. We elect to divide the overall problem into three smaller subproblems, and employ specialized methods for tackling each of these. The three subproblems are: assigning partitions to modules to verify the distribution constraints, performing local optimization on a single module, and performing global optimization.

4.1. Partition assignment

The partition assignment problem aims to distribute the partitions among the available modules in a way that verifies the distribution constraints. Following this, the first step is to find one viable assignment of partitions to modules, which essentially consists in assigning values to $f_i, \; \forall p_i \in \mathcal{P}$, such that an initial solution can be created. At the same time, tackling this reduced problem allows us to quickly prove infeasibility for certain problem instances without entering the additional complexity of considering the actual schedules.

The chosen approach is to use constraint programming, which is a generic framework to solve combinatorial problems like this one, modelling as a CSP. The formulation as a CSP is straightforward from the MILP model, but here we rearrange it to use f nomenclature. $\delta(x)$ denotes the Kronecker delta.

Variables : $f_i, \; \forall p_i \in \mathcal{P}$
Domains : $\{m\}, \; \forall c_m \in D_i$
Constraints :

$$\text{memory:} \quad \forall m \sum_i \{ s_i \cdot \delta(f_i - m) \} \le S_m \tag{26}$$

$$\text{inclusion:} \quad f_i = f_j \tag{27}$$

$$\text{exclusion:} \quad f_i \neq f_j \tag{28}$$

This problem can be solved using a general purpose CSP search algorithm. One advantage of CSPs is they are able to perform search using generalized heuristics not dependant on the problem structure. Another important characteristic of using backtracking search to solve this CSP is completeness, as we are able to prove infeasibility of the whole partition scheduling problem if the partition assignment subproblem is infeasible. Therefore, the first step in the scheduling tool will be to find a valid solution to this subproblem.

Other options for solving this subproblem would be using an Integer Linear Programming (MILP), or a sequential assignment algorithm, the latter of which was verified to be effective for loosely constricted problem instances.

4.2. Local optimization

Optimizing the offsets for all partitions such that α is maximized is a complex problem. However, optimizing the offset of one partition in the schedule while taking the other partition offsets as fixed is feasible. The strategy is to iteratively update the offset of each partition to a better value, and due to the similarities with game theory, the procedure is called the best response algorithm. This solution was studied separately in [9, 15].

Consider the partitions players in a game. The game is played in turns, each player updates its strategy knowing the current strategy for the other players, and the game is played until the strategies converge. In particular, each player chooses the offset that maximizes its utility (defined in equation 17), which is the factor by which all executions can be multiplied without overlapping with its own execution window. Since partitions choose their offset independently, this game is categorically non-cooperative, and the optimal solution lies on an equilibrium point, which in game theory is known as a Nash Equilibrium Point, from Nash [1]. However, a partition's utility maximizes not only the partition's window of execution, but also other partitions' interactions with its own, therefore, it has a cooperative trait. This is an important aspect that guarantees that this procedure converges to one of these equilibrium points, and additionally, this point will be at a local optimum with respect to the α-parameter. The converse is also true, any local optimum solution will be an equilibrium point.

In general, problem instances have many equilibrium points, which are all locally optimal solutions to the scheduling problem. Finding the optimal solution consists in finding the best of these equilibrium points, and is achieved by providing different starting points to the best response algorithm.

The introduction of chains restricts which offsets are valid, which has the effect of speeding up convergence since it restricts the problem further, however, it also increases the number of equilibrium points, making the procedure more dependant on the initial state. Using multiple windows, we must consider each individual window as a separate player, with the exception of all windows with $u = 1$, which are not independent.

Overall, the best response algorithm has complexity $\mathcal{O}(NA^{N-1})$, where N is the number of players and A is the number of strategies per player [14].

4.2.1. Linear search

The "best value" procedure consists in finding t_i which maximizes utility α_i of a given partition. We call it linear search because linear programming can be used to compute this value. This is trivially solved by computing α_i for all possible values of t_i and choosing the best one, however, this procedure will be repeated many times inside the local and global search procedures, therefore a faster method is paramount.

The solution to this problem, as investigated in Al Sheikh et al. [9] and Pira and Artigues [15], is to determine intersection points of the partition utilities based on knowledge from the solution set, and compute the partition utility only in these intersection points. The solution set is composed of adjacent polyhedra, with the maximum value on each polyhedra being located at an intersection of an ascending and descending line of the partition utility.

The actual calculation of the utility is done in $\mathcal{O}(N_p)$ time (see equation 17), therefore the exhaustive search runs in $\mathcal{O}(N_p T_i)$. For the method based on intersection points, we first note that for N constraining partitions, there will be N^2 intersection points in each polyhedron, with the number of polyhedra being trivially bounded by T_i, so actually this algorithm runs in $\mathcal{O}(T_i {N_p}^3)$. Despite this asymptotic time complexity being clearly worse, for usual problem instances with limited N_p, we verify that this version is superior to exhaustive search, as will be seen in section 5.

The effects of chains in liner search is constricting the partition offsets to valid regions that verify the maximum processing time. Multiple windows, however, degenerate the solution set and thus linear search cannot be applied; for these we must use exhaustive search.

4.2.2. Parallel best response

Parallel best response is what we define by applying local optimization to several modules simultaneously, motivated by the fact that when there are chains spanning two modules, their respective schedules are not independent. The approach in this case is to take all partitions assigned to a certain group of modules as the set of players, and proceed with the best response algorithm as normal, with the new nuance being that for determining the best value, only the partitions in the same module should be considered.

Preceding this, an extra step is needed, which is to determine which modules need to be optimized in parallel and which can be optimized independently. If we form an undirected graph where the modules are vertices and any chain creates an edge between the modules where the partitions are assigned to, then this problem consists in enumerating all unconnected subgraphs, solvable in linear time with depth-first search, for example.

4.3. Global optimization

Local optimization allows us to efficiently generate schedules for a single module after having defined the partitions that are assigned to this module as well as their configuration with respect to windows. The followed strategy is based on stochastic optimization algorithms, in particular we implement SA, Tabu-search and a genetic (or evolutionary) algorithm. These are of course classified as local search algorithms, but the fact that we are applying them only to a subtask in our problem, namely exploring starting points for a dedicated local optimization procedure, makes it adequate to use them for global search. These algorithms operate on a complete solution and gradually improve it, and this allows for the usage of the modularity of the local optimization procedure to improve only the needed modules. Another reason for this choice is we lack a proper way to evaluate partial solutions, that is, not all problem variables being assigned a value, which means a constructive algorithm is not appropriate.

4.3.1. Operators

Operators allow us to move from a state, S, which is a not necessarily valid solution, represented by a complete assignment of our problem variables, to another state differing by only some values of these variables. The operators apply to states and produce a new state, as $op(S_1) \rightarrow S_2$. Six operators are defined and detailed below: the move operator, $Mov(S, \mathcal{X}, m, n)$, the swap operator, $Sw(S, \mathcal{X}_1, \mathcal{X}_2, m, n)$, the shuffle operator, $Sh(S, m)$, the local optimization operator, $Lop(S, \mathcal{M})$, the slice operator, $Sl(S, i)$, and the crossover operator, $Cr(S_1, S_2)$, which is the only binary operator and combines two states to produce a new one.

The move operator changes the assignment of a group of partitions, $\mathcal{X} \subseteq \mathcal{P}$, essentially moving from one module to the other. When applying it, we can use a few heuristics. Most importantly, we can attempt to move a partition away from the most constrained module, but also, in order to comply with the distribution constraints, partitions subject to $f_i = f_j$ should be moved in the same group.

The swap operator swaps the assignment of two partitions, or two groups of partitions, virtually chaining two move operators. This operator is useful for reaching certain states without passing through worse intermediary states, having either low α or just invalid distribution constraints.

The shuffle operator provides a new start point for the local optimization procedure, by assigning random offsets to all partitions and all partition windows assigned to a certain module.

The local optimization operator is essentially the local optimization described before, applicable to a group of modules in parallel, or a single module as needed.

The slice operator changes the window configuration of a partition, essentially selecting different preemption points from $\mathcal{B}_i$. This is done randomly, but heuristically we can prefer the single window configuration more often when the execution is already sliced, because even though this configuration does not necessarily lead to an optimal result, it is important to decrease complexity. We can also preferentially slice partitions with large executions.

The crossover operator is a binary operator specific to the genetic algorithm that combines two states into one. For this specific problem, the gene representation considers tuples (f_i, t_i) for each partition, representing the module and the offset, respectively. Essentially, we combine a subset of partitions from one state with the remaining from the other state: In order to use the strengths of genetic algorithms, the genes should represent good solution characteristics with some modularity, such that they can be transmitted to new solutions, and be gradually improved. In this regard, the chosen representation is flawed, as the optimization criteria fundamentally evaluates groups of partitions. To gain some value from the crossover operator, we favour that partition pairs involved in a chain originate from the same state.

4.3.2. Operator selection and strategy

The main idea is to apply one of the operators Mov, Sw, Sh, Sl, Cr at each iteration, followed by Lop only on the modules which were affected. The overlying meta-heuristic algorithm is responsible for choosing which operators are used, to which variables, and also whether or not to accept the resulting solution. The operator selection is done mostly randomly as is characteristic of the class of algorithms used, but the heuristics described for each operator affect the selection. Essentially, the operator is chosen according to a fixed probability vector, determined empirically.

Upon description of these operators, one should intuitively notice that these operators are sufficient for reaching any possible distribution of partitions among modules. In particular, just applying Mov at random visits every possible state with respect to the partition assignment to modules, given enough time. Furthermore, with Lop we can reach every equilibrium point corresponding to local maxima.

The implemented algorithms are identical in most aspects. A copy of the best state so far is kept at all times, and is returned if the stopping condition is verified or the algorithm is stopped early. Additionally, a current state (or a population of states) is kept, and the algorithm traverses the search space by applying the operators to generate new states. The new states can be better (in the sense of the optimization criterion) or worse than the current state, and the policy that decides whether to accept or reject a new state is the main distinguishing factor between the meta-heuristic algorithms.

5. Results

In this section, we evaluate the computational performance of the methodology detailed in the previous section, evaluating both the scheduling tool as a whole, and some individual algorithms. The tests are run in a machine equipped with an Intel® i7 CPU rated @ 3.6 GHz with 8 MB cache and 16 GB RAM memory, running Linux. The tool and all algorithms are implemented in Python 3.6 and the MILP-based solver used is the open source solver CBC [8]. All time measurements are taken as the sum of processor time in user model and kernel mode on behalf of the program, and every execution exceeding 24 h is aborted.

5.1. Local optimization algorithms

As a first analysis we evaluate the performance of some key algorithms that solve subproblems rather than the complete scheduling problem, namely the best response algorithm and its important component, the best value algorithm.

We evaluate two different algorithms for determining the best value. Here, `best_value_a` is the exhaustive version which checks the partition utility at each valid offset, having time complexity $\mathcal{O}(T_i N_p)$, and `best_value_b` is the algorithm that computes the utility only in a set of interest points, which has time complexity $\mathcal{O}(T_i {N_p}^3)$. Measurements are taken for several instances with 2 to 12 partitions, presented in figure 5.

Figure 5 presents these results. The results corroborate the asymptotic time complexity for these algorithms, and we verify that version 'b', despite having worse behaviour for large numbers of partitions, performs substantially better for the number of partitions per module typically found in these problems We further advance that the same is verified when we adjust the number of chains affecting the partitions, and the respective periods.

Finally, we analyse only cases with up to 12 partitions because in real-life instances the number of partitions per CPM is consistently around this value. Even if future improvements to IMA motivate this number to increase, we present an algorithm that has linear time complexity, so this methodology is not invalidated in this case.

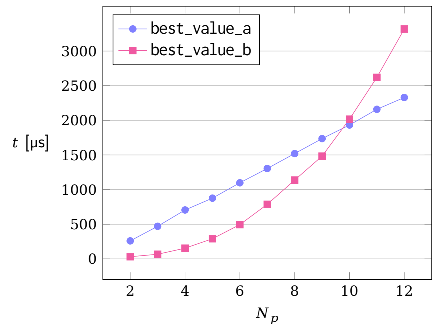


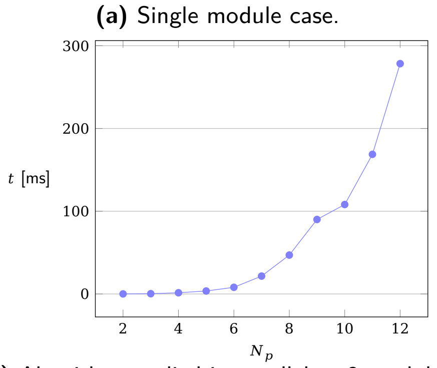
Figure 5: Performance of two methods for computing the best value for partition offsets.

Evaluation of the performance of the best response algorithm is done similarly for the single module case, the results being shown in figure 6a, and for the parallel approach, we consider similar numbers of partitions per module in a case with 3 modules, showing the results in figure 6b. We verify the predicted exponential complexity with respect to the number of partitions, but we note that for the parallel approach adding a partition does not always increase the time to convergence, as it can further restrict the problem.

Although this algorithm only reaches the optimal solution from a fraction of starting points that decreases as the dimension of the problem increases, the time to convergence remains overwhelmingly smaller when compared to the alternative MILP approach, which for these cases ranges from seconds to 2.2 h in the case with $N_p = 12$.

5.2. Scheduling tool

We define three test cases for the problem without multiple windows, which are identified by the number of modules and partitions, and one test case of moderate complexity for a problem where multiple windows are allowed. The test cases are generated randomly, choosing periods from the set $\{100, 200, 500, 1000\}$ which allows



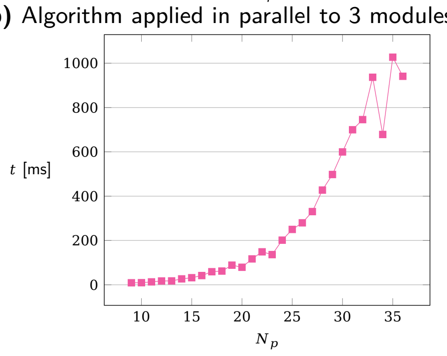


Figure 6: Performance of the best response algorithm as function of the number of partitions.

non-harmonic periods, and durations up to 15 % of the respective period. With respect to the distribution constraints, the memory requirements of the partitions are set to about 40 % of the modules' capacity, thus imposing some restriction. About 20 % and 5 % of partitions are subject to an exclusion and inclusion constraints, respectively, and the domains are not restricted. Regarding communications, chains are defined with maximum delays in the order of magnitude of the partition periods, but always in a way that some restriction is imposed. The summary of these test cases is presented in table 1. We

Table 1: Test cases definition.

Designation	N_c	N_p	Number of chains	α_{best}
$2M6P$	2	6	0	5.5
$3M15P\text{-}S$	3	15	3	1.26
$20M100P$	20	100	40	2.325

are not able to anticipate the optimal α-parameter for these test cases given that the MILP model was only able to provide an optimal solution for the easiest case, $2M6P$. Hence we present the best value found in the course of gathering results, α_{best}. The test case $3M15P\text{-}S$ allows multiple windows on some partitions, and was manually adjusted so that it is infeasible without these multiple windows.

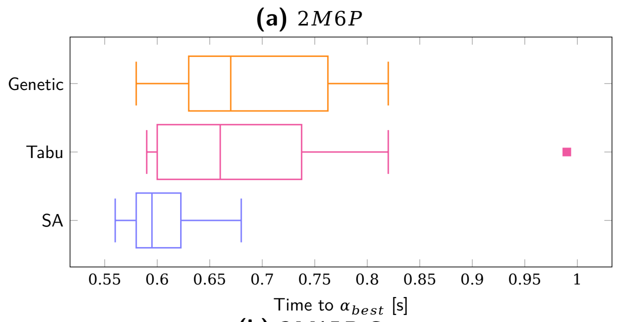
We evaluate the performance of the scheduler to finding a valid solution, without attempting to optimize the α-parameter, as listed in table 2. Results are shown in table 2. Both exact and heuristic methods perform well for the easier cases, but the MILP formulation does not converge in acceptable time for the largest case considered. Also, since our MILP model does not consider multiple windows, it is unable to handle $3M15P\text{-}S$, but ignoring the possibility of multiple windows, it proves infeasibility in 1.19 s.

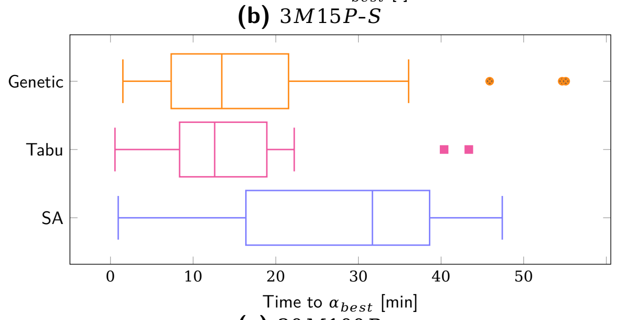
Table 2: Scheduler performance finding the first valid solution.

Instance	t_{MILP}	$t_{heuristic}$ (median)
$2M6P$	1.00 s	0.593 s
$3M15P\text{-}S$	NA	74.53 s
$20M100P$	> 24 h	23.32 s

The performance of the scheduler is appropriate for the industry setting, where we can expect it to be able to verify feasibility in seconds even for complex problem instances. This is certainly useful because system integration depends on many interactions with the different suppliers to define all the parameters and requirements, which are then translated into constraints accepted by this model. In this stage, the problem changes quickly and having a tool to check feasibility and build a simple solution is of great benefit to the system integrator.

For the optimization problem, we are interested in finding the optimal solution, however, since these optimal value is unknown, we aim to evaluate the performance in finding a *good* solution. This solution is characterized by an evaluation function of α_{best}, but for convenience we settle for a value $\alpha \geq \alpha_{best}$ for $20M100P$. The meta-heuristic algorithms used are stochastic, thus the time taken can vary greatly between separate runs. For this reason, we measure the solution time 20 times for each meta-heuristic, and present the results as boxplots in figure 7. We verify from these results that SA is superior





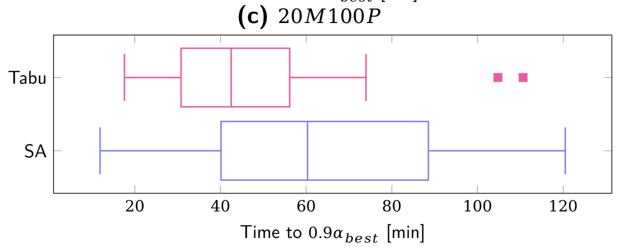


Figure 7: Performance of the scheduler with different meta-heuristics.

in smaller cases, and Tabu-search more consistent for larger cases, while the genetic algorithm scales poorly and ultimately does not produce the targeted result in under 24 h.

As for the consequences to the industrial setting, based on the results for $20M100P$, we can expect to have a fairly optimized solution for the partition scheduling problem consistently in under an hour of processor time (over 75 % of attempts), which is an excellent result. We further observe that allowing multiple windows on $3M15P$-S induces an expensive increase in the solution time, especially considering the relatively small number of modules and partitions for this case.

5.2.1. Comparison to related work

We compare results to Al Sheikh et al. [9] by generating multiple instances of a case with 4 modules and 40 partitions in the same way as described in this reference, which only considers memory, exclusion, and temporal segregation constraints, thus consisting in a subset of the problem described here. Our scheduler reached its stopping condition in 5.17 min to 45.85 min with the average being 15.25 min, compared to this reference's 5 min to 50 min and average of 27.4 min. However, we were unable to replicate the used stopping condition, so we elect to stop our scheduler on $0.95\alpha_{best}$. In spite of this, we can conclude that this approach is an improvement on similar work.

6. Conclusions

The research work presented focused on formally describing an avionics partition scheduling problem, and providing methods for efficiently solving it. The final product is integrated in GMV's tool suite for system configuration.

Firstly, the described model remains compatible with most similar approaches to the problem, while supporting more kinds of constraints, which allow the user (the system integrator) to adequately specify platform requirements, with respect to resource usage as well as the redundancy architecture.

The different methods implemented also allow the scheduling tool to be used for multiple purposes. The heuristic methods are generally able to quickly provide a solution that verifies all constraints, and this is useful for determining feasibility of certain instances in the early stages of integration. Additionally, in later phases of system integration, the heuristic methods are able to create optimized solutions in a moderate amounts of time. If on the other hand optimality is the goal and time is not an issue, then any external solver can take our MILP model and solve the problem to optimality.

In addition, this work poses a contribution to academic research due to the novel changes imposed in our model, namely the synchronous communications model and the possibility for splitting a partition's execution in multiple windows.

6.1. Future work

Given the recent additions to the Arinc specification 653 and the lacklustre performance observed in our scheduler with the addition of multiple partition execution windows, we consider that the most relevant future work on partition scheduling should be dedicated to multicore IMA systems.

Other smaller modifications to the described model would also be interesting to consider, namely on the optimization criterion or the communications model, driven by the IMA platform's specific requirements.

References

[1] John Nash. “Non-cooperative games” (1951). DOI: 10.2307/1969529.

[2] Jan Korst et al. “Scheduling periodic tasks” (1996). DOI: 10.1287/ijoc.8.4.428.

[3] Y-H Lee et al. “Scheduling tool and algorithm for integrated modular avionics systems”. IEEE. 2000. DOI: 10.1109/DASC.2000.886885.

[4] Michael Pinedo. *Scheduling. Theory, Algorithms, and Systems*. 3rd ed. Springer, 2008. DOI: 10.1007/978-0-387-78935-4.

[5] Ahmad Al Sheikh et al. “Partition scheduling on an IMA platform with strict periodicity and communication delays”. 2010.

[6] Friedrich Eisenbrand et al. “Scheduling periodic tasks in a hard real-time environment”. Springer. 2010. DOI: 10.1007/978-3-642-14165-2_26.

[7] Friedrich Eisenbrand et al. “Solving an avionics real-time scheduling problem by advanced IP-methods”. Springer. 2010. DOI: 10.1007/978-3-642-15775-2_2.

[8] COIN-OR Foundation. *CBC – Coin-or branch and cut*. Version 2.9.0. 2010. (Visited on 07/08/2019).

[9] Ahmad Al Sheikh et al. “Strictly periodic scheduling in IMA-based architectures” (2012). DOI: 10.1007/s11241-012-9148-y.

[10] CM Ananda et al. “ARINC 653 API and its application–An insight into Avionics System Case Study” (2013). DOI: 10.14429/dsj.63.4268.

[11] Sofiene Beji et al. “SMT-based cost optimization approach for the integration of avionic functions in IMA and TTEthernet architectures”. IEEE Computer Society. 2014. DOI: 10.1109/DS-RT.2014.28.

[12] *Arinc Specification 653: Avionics Application Software Standard Interface. Part1: Required Services*. Aeronautical Radio, Incorporated. 2015.

[13] Aamir Mairaj. “Preferred choice for resource efficiency: Integrated Modular Avionics versus federated avionics”. IEEE. 2015. DOI: 10.1109/AERO.2015.7119127.

[14] Stéphane Durand and Bruno Gaujal. “Complexity and optimality of the best response algorithm in random potential games”. Springer. 2016. DOI: 10.1007/978-3-662-53354-3_4.

[15] Clément Pira and Christian Artigues. “Line search method for solving a non-preemptive strictly periodic scheduling problem” (2016). DOI: 10.1007/s10951-014-0389-6.

[16] Miguel Barros et al. “Distributed Integrated Modular Avionics”. NATO. 2018.

[17] Mathias Blikstad et al. “An optimisation approach for pre-runtime scheduling of tasks and communication in an integrated modular avionic system” (2018). DOI: 10.1007/s11081-018-9385-6.