Scheduling Onboard Tasks of the NIMPH Nanosatellite

Julien Rouzot[1,2], Joséphine Gobert[1], Christian Artigues[1], Romain Boyer[1], Frédéric Camps[1], Philippe Garnier[2], Emmanuel Hebrard[1] and Pierre Lopez[1]
[1]*LAAS-CNRS, Toulouse, France*
[2]*IRAP, Toulouse, France*
{*firstname.lastname*}*@laas.fr*, {*firstname.lastname*}*@irap.omp.eu*

Keywords: Scheduling, Constraint Programming, Nanosatellite, Hypervisor.

Abstract: In the context of terrestrial space missions, thanks to the recent development of micro and nanotechnologies, nanosatellites are becoming increasingly popular for their lower cost and ease of deployment. The NIMPH (Nanosatellite to Investigate Microwave Photonics Hardware) mission is an ongoing academic project aimed at developing and launching such a nanosatellite. The onboard resources on these missions are often very limited, and in our study case, a single onboard computer is responsible for orchestrating the science and avionic tasks of the nanosatellite. These tasks are subject to various constraints, such as frequency, minimum/maximum delay between the execution of the same type of task and strict precedences. This makes the scheduling of the onboard tasks a challenging problem, which is critical for the mission success. In this paper, we tackle the problem of scheduling NIMPH onboard tasks using Constraint Programming methods. Our scheduler demonstrates its performance by generating optimal or near-optimal schedules for the NIMPH nanosatellite.

1 INTRODUCTION

The NIMPH mission is part of the Nanolab-Academy project proposed by the CNES (French Space Agency) that encourages students to engage in space exploration by developing, launching and exploiting their nanosatellites of type CubeSat, through internships and academic projects (CNES, 2021). NIMPH stands for Nanosatellite to Investigate Microwave Photonics Hardware. This project is still in its development phase (C), with a launch planned for 2025. From a general perspective, the nanosatellite's mission is to evaluate the influence of the space environment on optoelectronic components (Landrea et al., 2018).

The nanosatellites developed in the context of Nanolab-Academy use the same flight software provided by CNES. The science and avionic operations are driven by dedicated and isolated partitions (Windsor and Hjortnaes, 2009), that are orchestrated by the onboard computer (OBC) hypervisor. XTratuM (Masmano et al., 2009) is used to manage the execution of these partitions. This bare-metal hypervisor is a software developed by the Spanish company FentISS and has been widely used for space missions.

The partitions represent the containers of all the different tasks that must be performed by the nanosatellite, such as orientation correction, scientific measures, uplink command management, and more. As the OBC is unable to perform any sort of multiprocessing, the partitions must be executed sequentially and cannot overlap. To simplify the scheduling of the OBC tasks for Nanolab nanosatellite missions, the team developing the flight software chose to make the schedule cyclic, with a finite time horizon of one second. This means the OBC will repeat the same partitions executions every cycle. A fixed number of tasks (i.e., partition execution of constant duration) for each partition must be performed within this time horizon. In the context of the NIMPH mission, due to the number of tasks and the various constraints on the partitions, obtaining manually the OBC schedule is an extremely challenging work. Finally, the best schedules aim to maximize the effective use of the schedule time, which corresponds to the sum of the task duration minus the number of *context switches*, that happen between the execution of two different partitions and are time-consuming. This aspect makes the cyclic scheduling of NIMPH tasks even harder.

XTratuM comes with XoncretE (Balbastre et al., 2021), a dedicated scheduler that allows the generation of a cyclic execution plan of the different partitions, that minimizes the number of context switches.

Rouzot, J., Gobert, J., Artigues, C., Boyer, R., Camps, F., Garnier, P., Hebrard, E. and Lopez, P.
Scheduling Onboard Tasks of the NIMPH Nanosatellite.
DOI: 10.5220/0012378500003639
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 13th International Conference on Operations Research and Enterprise Systems (ICORES 2024)*, pages 277-284
ISBN: 978-989-758-681-1; ISSN: 2184-4372
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

This tool has a major drawback: most constraints on the NIMPH cyclic task scheduling problem cannot be expressed within XoncretE. This makes the generation of valid schedules a laborious work, as the plans must be manually refined to match the real constraints (at least in the context of the NIMPH mission). This also often leads to poorer solutions.

In this context, we propose NanoSatScheduler[1], a scheduler based on Constraint Programming methods, that tackles NIMPH's specific constraints, while being generic enough to handle OBC partition scheduling for other nanosatellite missions. We first introduce the NIMPH cyclic task scheduling problem formally and compare it to other similar scheduling problems in the literature. We then highlight two constraint programming models, NanoSat and NanoSat-Global, using the IBM CP Optimizer solver, to solve the NIMPH cyclic task scheduling problem. We also present NanoSatIterative, an iterative method to maximize the effective schedule time by adding new tasks while minimizing the number of context switches and keeping the optimality proof. These methods are evaluated on synthetic instances based on NIMPH parameters to demonstrate their performance. We finally conclude on the limitations and future work for our scheduler.

2 THE NIMPH CYCLIC TASK SCHEDULING PROBLEM

2.1 Formal Description

To handle the scientific and avionic tasks of the NIMPH nanosatellite, the main software is divided into a finite set of N partitions, denoted $P = \{P^1, P^2, \dots, P^N\}$. We can refer to a particular partition by its name (e.g., P^{IOS} for IOS partition). These partitions can be seen as containers for a piece of software dedicated to a certain task.

In the NIMPH cyclic task scheduling problem, a task corresponds to the execution of a partition for an arbitrary amount of time. Each partition P^i must be executed a fixed number of times M^i within the time horizon h. As the schedule is cyclic, these tasks will be repeated every cycle. Therefore, for each partition i, there is a set of tasks (i.e., partition execution) $(i,j)_{i=1..N, j=1..M^i}$ to schedule. Let s^i_j and e^i_j be the start and end times of the j-th occurrence of partition P^i, respectively. The tasks have a non-zero duration (1), that is fixed for each partition occurrence and is denoted by δ^i. As the tasks are equivalent within the same partition, we assume a correspondence between their lexicographical order and their temporal order (2). The tasks cannot overlap (3).

$$e^i_j = s^i_j + \delta^i \qquad i = 1..N,\ j = 1..M^i \qquad (1)$$

$$e^i_j \leq s^i_{j+1} \qquad i = 1..N,\ j = 1..M^i\text{-}1 \qquad (2)$$

$$e^i_j \leq s^{i'}_{j'} \vee s^i_j \geq e^{i'}_{j'} \qquad (i,j) \neq (i',j') \qquad (3)$$

Some partitions need to be executed regularly. For instance, the *SCAO* partition must regularly check the alignment between the solar arrays and the sun, as an incorrect angular shift could result in a dramatic loss of power. The set of partitions subject to this constraint is denoted by $C \subset P$. Such partitions are said to be *critical* and the associated tasks are called critical as well. As a consequence of this regularity requirement, a maximum delay d^i_{max} must be respected between the starting time of two tasks in a partition P^i in C (4). As the tasks are repeated in the same manner in the next cycle, the maximum delay constraint must be respected between the last task of each critical partition and the first task of the same partition in the next cycle (4').

$$s^i_{j+1} - s^i_j \leq d^i_{max} \qquad P^i \in C, j = 1..M^i\text{-}1 \qquad (4)$$

$$h + s^i_1 - s^i_{M^i} \leq d^i_{max} \qquad P^i \in C \qquad (4')$$

EDMON partition (ED) has a specific particularity: it is the only payload with a dedicated CPU. Therefore, it can run background tasks while other partitions are being executed by the OBC. The nominal behavior of *EDMON* is to wait for an uplink command transmitted by the OBC, run an experiment, wait for the OBC to get the results, and repeat this process. It means that the only worthwhile interaction with the OBC takes place when *EDMON* is waiting for an uplink command or waiting to send the results of the experiment[2]. The payload algorithm can be seen as a finite-state machine, with a **wait state** (waiting for a command), a **ready state** (ready to send the results), and several **experiment states** triggered by different uplink commands. As the execution times of the **experiment states** are known, we can impose a minimum delay d^{ED}_{min} (cf. constraints (5, 5')) between two executions of the partition that correspond to the shortest **experiment state**. This does not ensure that *EDMON* cannot be executed during an experiment (this is impossible because the longest experiments last more than the time horizon), but this tends to limit

[1]https://gitlab.laas.fr/roc/josephine-gobert/nanosatscheduler

[2]The execution of this partition during an experiment is a waste of the schedule time as *EDMON* runs the experiments with its own CPU.

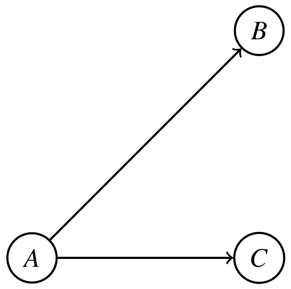

Figure 1: Simplified precedence graph for NIMPH schedule.

the waste of schedule time.

$$s^{ED}_{j+1} - s^{ED}_{j} \geq d^{ED}_{min} \qquad j = 1..M^i\text{-}1 \quad (5)$$

$$h + s^{ED}_{1} - s^{ED}_{M^{ED}} \geq d^{ED}_{min} \quad (5')$$

Certain partitions are also subject to strict precedence constraints. For instance, as the *IOS* partition handles the I/O, it must be executed just before any scientific payload to ensure that the last uplink command will be transmitted. We can describe these constraints with a directed acyclic graph $G(V,E)$, where the nodes (V) are the partitions to execute and the arcs (E) represent a direct transition between two partitions in the schedule. Figure 1 states that an execution of partition A must occur right before any execution of partition B or C. In the remainder of this paper, $A \rightarrow B$ will be used to indicate a strict precedence constraint between A and B.

In an operating system, the *context* of a process (or a thread) represents its current state (variables, instruction pointer, etc.). To ensure the good behavior of the system, the context of these pieces of software must be stored every time they are stopped and loaded when restarted. These steps are called *context switches* (Comer and Fossum, 1987). Obviously, this process can become very costly if the context switches happen too often, not only because of the direct cost of storing and loading the partition's contexts, but also because of the indirect cost of cache interference (Li et al., 2007). In our context, we will assume a constant time penalty p corresponding to the direct cost of a context switch. Context switches can be represented by a binary variable c^i_j associated with each task (i,j) that indicates whether a context switch is needed at the end of task (i,j). The only way to avoid a context switch is by merging two consecutive tasks of the same partition (6). Note that the last task of each partition will necessarily need a context switch (6')[3]:

$$c^i_j = 0 \iff e^i_j = s^i_{j+1} \quad i = 1..N,\ j = 1..M^i\text{-}1 \quad (6)$$

$$c^i_{M^i} = 1 \qquad i = 1..N \quad (6')$$

[3]A context switch is also needed to start a new cycle.

Our objective is to maximize the schedule time that will be effectively used to perform avionic or payload tasks. More precisely, this objective is the sum of the task durations, from which we subtract the sum of time penalties introduced by the context switches:

$$\max \quad \sum_{i=1}^{N} \sum_{j=1}^{M^i} \delta^i - c^i_j \qquad (7)$$

In the remainder of the paper, we will call this value the *useful schedule time*. As we assume a fixed number of tasks (i,j) and a constant time penalty p, this objective is equivalent to minimizing the number of context switches. Nevertheless, we will see in Section 3.4, how we can further improve objective (7) by adding more tasks to the NIMPH instances.

2.2 Related Work

The problem is related to many scheduling problems in the literature. First, despite its cyclic nature, since there is no overlap between two cycles, the problem is equivalent to an acyclic single-machine scheduling problem with minimum and maximum time-lag constraints: constraints (4) and (4') are maximum time lags while constraints (5) and (5') are minimum start-start time lags. More precisely, find a schedule of makespan lower than h with constraints (1)–(5') is a particular case of the decision variant of the NP-hard one-machine scheduling problem with generalized precedence constraints considered in (Wikum et al., 1994). In our case, all operations of a chain (partition) have the same duration and the minimum and maximum time lags have special values. In (Yu et al., 2004), the problem of minimizing the makespan on a single machine with two unit-time operations per job with arbitrary intermediate minimum delays is shown to be strongly NP-hard. The decision variant of this problem can be obtained by the following relaxation of our problem: we define only two unit-duration tasks per partition, ignore the EDMON partition and consider only constraints (1)–(3) and (4') by setting d^i_{max} to h minus the minimum delay of the job. The context switch between two partitions is also a variant of the sequence-independent family setup time in serial batching problems (Potts and Kovalyov, 2000).

2.3 Illustrative Example

To illustrate the NIMPH cyclic task scheduling problem, let us define a simple instance **NIMPH1** with three partitions *IOS*, *INST* and *EDMON*. We want to fit four *IOS* and *INST* executions and two *EDMON* executions of $10\mu s$ each, within the time hori-

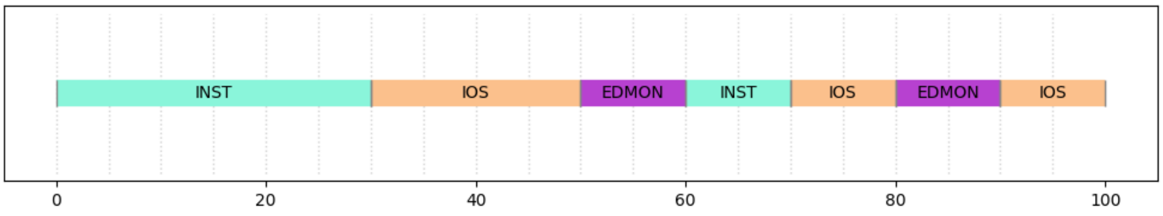

Figure 2: A valid schedule for instance NIMPH1.

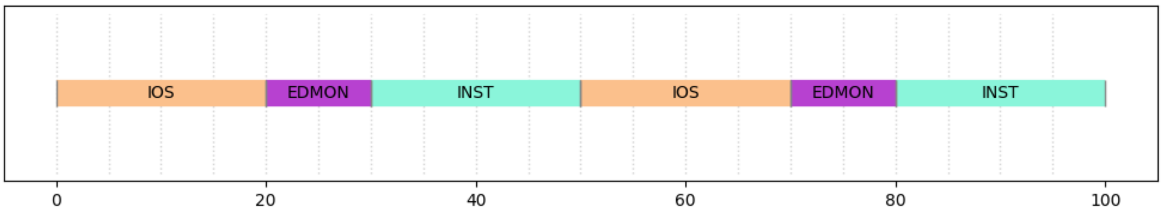

Figure 3: An optimal schedule for instance NIMPH1.

zon $h = 100\mu s$. We assume that $IOS \rightarrow EDMON$ is the only precedence constraint and that *INST* is the only *critical* partition with $d_{max}^{INST} = 40\mu s$. Finally, the minimum delay between two *EDMON* executions is $d_{min}^{ED} = 30\mu s$. Both Figures 2 and 3 represent valid schedules (i.e., a valid assignment of all tasks start) with respect to these constraints. Note that consecutive executions of the same partition are merged in the Gantt chart. The schedule represented in Figure 2 is valid but suboptimal (7 context switches), while the schedule in Figure 3 is optimal (6 context switches).

3 NanoSatScheduler

We chose to tackle the NIMPH cyclic task scheduling problem using Constraint Programming methods. We implemented NanoSatScheduler using IBM ILOG CP Optimizer commercial solver. There are two main reasons for this choice. First, CP Optimizer has shown great results on a variety of scheduling problems through the years (Laborie et al., 2018). This solver is also simple to use, with dedicated libraries such as *docplex* for Python to build CP models (IBM, 2023). CP Optimizer provides global constraints and variable types, that make the modeling of complex problems an intuitive process, especially for scheduling.

We decided to implement two different models to compare the performance of two modeling approaches for the NIMPH cyclic task scheduling problem. The first model is called NanoSat and uses only classical **integer** variables and no global constraints. Model NanoSatGlobal takes advantage of the scheduling features of CP Optimizer (particular variable structures and global constraints).

We finally developed an iterative method to insert tasks of a given partition of the base instance to maximize the use of the schedule time, while taking the context switches penalties into account.

3.1 NanoSat

Our first constraint model aims to tackle the NIMPH cyclic task scheduling problem only using simple constraints and integer variables. Hence, it is possible to implement it in any constraint solver. For NanoSat, we define the following variables:

$$s_j^i \in [0,h] \qquad i = 1..N,\ j = 1..M^i \qquad \text{Start} \qquad (8)$$

$$e_j^i \in [0,h] \qquad i = 1..N,\ j = 1..M^i \qquad \text{End} \qquad (9)$$

$$c_j^i \in [0,1] \qquad i = 1..N,\ j = 1..M^i \qquad \text{CS} \qquad (10)$$

Start variables (8) are the start times for the j-th occurrence of partition i. These are the only decision variables, and a valid assignment for all start variables is a solution. Task end times can be calculated from the task starts and the constant task durations. End variables (9) are only here for the sake of model clarity. The tasks must be scheduled between 0 and the time horizon h, therefore the domain of the start and end variables is restricted to $[0,h]$. Context switch (CS) variables (10) are binary variables that indicate, for each task, whether the end time of the current task is **not** equal to the start time of the next task of the same partition (i.e., a context switch is needed). As we want to limit the waste of the OBC schedule time, our objective is to minimize the number of these context switches under the following constraints:

$$\min \sum_{i=1}^{N} \sum_{j=1}^{M^i} c_j^i \qquad (11)$$

$$e^i_j = s^i_j + \delta^i \qquad i = 1..N,\ j = 1..M^i \tag{12}$$

$$e^i_j \le s^{i'}_{j'} \textbf{ or } s^i_j \ge e^{i'}_{j'} \qquad (i,j) \neq (i',j') \tag{13}$$

$$e^i_j \le s^i_{j+1} \qquad i = 1..N,\ j = 1..M^i\text{-}1 \tag{14}$$

$$s^i_{j+1} - s^i_j \le d^i_{max} \qquad i : P^i \in C, j = 1..M^i\text{-}1 \tag{15}$$

$$h + s^i_1 - s^i_{M^i} \le d^i_{max} \qquad i : P^i \in C \tag{16}$$

$$s^{ED}_{j+1} - s^{ED}_j \ge d^{ED}_{min} \qquad j = 1..M^i\text{-}1 \tag{17}$$

$$h + s^{ED}_1 - s^{ED}_{M^{ED}} \ge d^{ED}_{min} \tag{18}$$

$$c^i_j = 1 \textbf{ or } e^i_j = s^i_{j+1} \qquad i = 1..N,\ j = 1..M^i\text{-}1 \tag{19}$$

$$c^i_{M^i} = 1 \qquad i = 1..N \tag{20}$$

$$e^A_k \le s^B_k \qquad (A,B) : A \to B = E' \tag{21}$$

$$e^i_j \le s^A_k \textbf{ or } e^B_k \le s^i_j \qquad \begin{array}{r} i = 1..N,\ j = 1..M^i \\ k = 1..M^A \\ (i,j) \neq (A,k) \\ (i,j) \neq (B,k) \\ (A,B) : A \to B = E' \end{array} \tag{22}$$

Constraint (12) just defines the end of the tasks to their start plus their duration. Constraint (13) forbids the overlapping of any pair of tasks, as the OBC can only execute one partition at the same time. The occurrences of the same partition are time-ordered (14) for two reasons. First, the next constraints (15–22) are easier to express if the tasks are ordered, which improves model clarity. But more importantly, it eliminates a lot of symmetric solutions that could impact the solving performance.

Constraint (15) states that the difference between the start times of two consecutive critical tasks should not exceed the maximum delay allowed for this partition. Constraint (16) handles the side effect of the transition between two cycles, as the schedule will be restarted at the time horizon. In the same manner, the minimum delay between two calls of *EDMON* partition is ensured with constraints (17–18).

We need to force the context switch variable c^i_j to 0, if and only if task $(i, j+1)$ starts exactly at the end time of the current task (i, j) (i.e., we can merge tasks (i, j) and $(i, j+1)$ in the schedule). Constraint (19) ensures that either c^i_j is equal to 1 or the start time of the occurrence $j+1$ of partition i is equal to the end time of the previous occurrence. As our objective is to minimize $\sum_{i=1}^{N} \sum_{j=1}^{M^i} c^i_j$, the solver will set c^i_j to 0 if the other condition is true, and c^i_j will be forced to 1 otherwise. The last task of every partition will necessarily induces a context switch, so we set the last context switch variable to 1 (20).

Our precedence graph $G(V,E)$ states that for each arc $A \to B$ in the precedence graph, only a task of partition A can be the direct predecessor of a task of partition B. In other words, no task of partition $C \neq A$ can be inserted in the middle of a sequence AB. To satisfy a precedence constraint $A \to B$, we must have at least as many tasks of type A as tasks of type B. More generally, if we have n precedence constraints for the same predecessor, e.g., $A \to B$, $A \to C$, ... the number of predecessor tasks must be greater than or equal to the number of successor tasks. As the tasks within the same partition are interchangeable, we can assign any task of the predecessor partition to be just before any task of any successor partition, without impacting the solution quality. Hence, for each partition that is a predecessor of at least one other partition, we can split the predecessor partition tasks set into n distinct sets A^B,A^C,... for each successor, plus one for the extra tasks of the predecessor partition. We can now express the precedences with a bipartite directed graph $G(V', E')$ with only distinct pairs of partitions. We then force the k-th task occurrence of type A^P to be the direct predecessor of the k-th task occurrence of type P with constraints (21) and (22). The first constraint is a classical precedence constraint: all tasks in a predecessor node must end before the start of the corresponding task is the successor node. The last constraint sets all other tasks to either end before the predecessor task or to start after the successor task for each precedence (i.e., each pair in our new precedence graph $G(E', V')$). As all partition sets are time-ordered, we ensure strict precedences, while maintaining the solution quality and breaking symmetries.

$$\begin{tikzcd} A^B \arrow[r] & B \\ A^C \arrow[r] & C \end{tikzcd}$$

Figure 4: New simplified precedence graph with predecessor partition split.

3.2 NanoSatGlobal

Model NanoSatGlobal uses the **interval** variables of CP Optimizer, a structure dedicated to task modeling. These variables have a start time, an end time and a duration (that is fixed in our problem). The main advantage of this kinds of variables is the synergy with the **sequence** global constraint that allows us to reason about the tasks as an ordered sequence rather than a set of start times. We will use the notation t^i_j for the interval variable representing the task (i, j). Our variables are defined as follows:

$$t^i_j \in [0,h] \qquad i=1..N,\ j=1..M^i \qquad \text{Task} \tag{23}$$

$$c^i_j \in [0,1] \qquad i=1..N,\ j=1..M^i \qquad \text{CS} \tag{24}$$

Our **interval** variables start and end domains are constrained within the time window $[0,h]$ (23). We model the context switches exactly as in the previous model, and the objective function is the same:

$$\min \sum_{i=1}^{N} \sum_{j=1}^{M^i} c^i_j \tag{25}$$

However, our constraints are fundamentally different. Rather than working with the start dates of the tasks, we use CP Optimizer scheduling global constraints on the **sequence**:

$$seq = \textbf{sequence}\left(\left(t^i_j, i\right)_{i=1..N, j=1..M^i}\right) \tag{26}$$

$$\textbf{no\_overlap}\,(seq) \tag{27}$$

$$\textbf{start\_of}\left(t^i_{j+1}\right) - \textbf{start\_of}\left(t^i_j\right) \leq d^i_{max}$$
$$\forall\, i : P^i \in C,\ j = 1..M^i\text{-}1 \tag{28}$$

$$h + \textbf{start\_of}\left(t^i_1\right) - \textbf{start\_of}\left(t^i_{M^i}\right) \leq d^i_{max}$$
$$\forall\, i : P^i \in C \tag{29}$$

$$\textbf{start\_of}\left(t^{ED}_{j+1}\right) - \textbf{start\_of}\left(t^{ED}_j\right) \geq d^{ED}_{min}$$
$$\forall\, j = 1..M^i\text{-}1 \tag{30}$$

$$h + \textbf{start\_of}\left(t^{ED}_1\right) - \textbf{start\_of}\left(t^{ED}_{M^{ED}}\right) \geq d^{ED}_{min} \tag{31}$$

$$c^i_j = 1 \textbf{ or } \left[\textbf{end\_of}\left(t^i_j\right) = \textbf{start\_of}\left(t^i_{j+1}\right)\right]$$
$$\forall\, i = 1..N,\ j = 1..M^i\text{-}1 \tag{32}$$

$$c^i_{M^i} = 1 \quad \forall\, i = 1..N \tag{33}$$

$$\textbf{type\_of\_prev}\left(t^B_j, A\right)$$
$$\forall\, j = 1..M^i,\ (A,B) : A \rightarrow B \in E' \tag{34}$$

In CP Optimizer, the **sequence** variable is a structure dedicated to task scheduling. We create such a variable with all our tasks and we assign a different **type** equal to the partition index for each (26). This allows the use of global constraints on the **interval** variables within the sequence, such as **no_overlap** constraint (27), that ensures that the tasks cannot run at the same time. Just like in the previous section, constraints (28, 29) ensure that the delay between two tasks of the same critical partition is below d^i_{max} and constraints (30, 31) impose a minimum delay between the start of two *EDMON* executions. The keyword **start_of** and **end_of** respectively refers to the start and the end of the **interval** variables. We constrain the context switch variables (32, 33) as in the last model (see Section 3.1). The global constraint **type_of_prev** (34) is exactly what we need to express our strict precedence constraints. This constraint forces the predecessor of the current task to be of a specified type. Note that we perform the same pre-processing step described in Section 3.1 to express our precedence graph $G(V,E)$ as a perfect matching $G(E',V')$. Therefore, for each arc $A \rightarrow B$ in $G(E',V')$, we assign a predecessor of type A (i.e., from partition A) to each task of partition B.

3.3 Lower Bound

To decrease the optimality proof computation time, we calculate a lower bound *lb* on the number of context switches for each instance. Indeed, our instances have the following properties: A context switch is needed for the last task of every partition. As the tasks are ordered, the task starting just at the end of the last task of a partition cannot be from the same partition; A context switch is needed for every task of *EDMON* partition as $d^{ED}_{min} > \delta^{ED}$; Every task subject to a strict precedence constraint will need a context switch (for both predecessor and successor); Tasks subject to a max delay constraint will need a minimum number of context switches.

This last property is trickier. It is not obvious how to compute the minimum number of context switches for this last constraint, as it depends on the number of tasks, the tasks' duration, the time horizon and the maximum delay allowed. To get this minimum number of context switches, we solve a simplified version of the NIMPH cyclic task scheduling problem, with only one partition, for each of these *critical* partitions. This simpler problem is solved to optimality very quickly, so we can compute this minimum number of context switches for each critical partition as a preprocessing step of NanoSatScheduler. The notation *lb* refer to the sum of all of these mandatory context switches, so we can add the following constraint to both NanoSat and NanoSatGlobal models:

$$\sum_{i=1}^{N} \sum_{j=1}^{M^i} c^i_j \geq lb \tag{35}$$

3.4 NanoSatIterative

Due to the various constraints on the NIMPH cyclic task scheduling problem, it is very hard to manually build instances that yield a valid solution that maximizes the useful schedule time. Moreover, it is not useful to maximize the number of tasks for all kinds of partitions. For instance, the number and duration of the avionic partitions tasks are designed to ensure the good behavior of the nanosatellite with respect to a security margin, so maximizing these types of tasks is unnecessary. On the other hand, the time allocated to the scientific payloads should be maximized to improve the scientific feedback.

To address this, we decided to relax the strict number of task constraints for **one** partition l, selected by the user. We thus propose NanoSatIterative, a method to find the optimal useful schedule time, by jointly selecting the number of tasks of partition l and scheduling all the tasks. To maximize the useful schedule time and keep the optimality guarantees, we perform a dichotomic search on our objective: $\sum_{i=1}^{N}\sum_{j=1}^{M^i} \delta^i - c_j^i$. It is possible to compute a lower and an upper bound on the number of tasks of type l. The lower bound is the minimum number of tasks that are necessary to reach the current objective, assuming an optimistic number of context switches[4], while the upper bound is the number of tasks of type l before the problem is trivially unsatisfiable (i.e., the sum of the task durations is greater than the time horizon). The lower bound is updated at each objective step, and then we iteratively add new tasks of partition l and try to solve the NIMPH cyclic task scheduling problem with a fixed number of tasks using NanoSatScheduler or NanoSatGlobal, until we reach either the upper bound or we find a feasible solution. According to the satisfiability of the problem with the current objective, we update either the lower bound or the upper bound of the useful schedule time. If a solution is found, we also update the lower bound on the number of tasks of type l with the current number of tasks. Note that we cannot perform a dichotomic search on the number of tasks as well, because a solution with more tasks is not necessarily better than another solution with fewer tasks but fewer context switches. If, at each step, we can find a valid solution or prove the unsatisfiability of the objective, the solution is optimal.

4 EXPERIMENTAL RESULTS

4.1 Instances

As the NIMPH mission is still in its development phase, some of the real constraints for the NIMPH cyclic task scheduling problem are still unknown (e.g., tasks' duration, exact number of occurrences, minimum delay for critical partitions, etc.). To perform realistic and relevant experiments, the NIMPH development team helped us create a base instance with the expected values for each partition. As this instance is subject to uncertainties and to deeply analyze the performances of our approach, we generated 100 random instances based on the NIMPH nominal instance (71 tasks from 10 distinct partitions, for a time horizon $h = 1s$). We created these synthetic instances by disturbing the original instance with a Gaussian law ($\mu = 1$, $\sigma^2 = 0.3$) for each uncertain parameter. After deleting trivially unsatisfiable instances $\left(\sum_{1=1}^{N} \delta^i \times M^i > h\right)$, we have a total of 98 instances.

[4]We assume we can reach the lower bound presented in Section 3.3.

4.2 Results

We compare the two implementations NanoSat and NanoSatGlobal of our model in CP Optimizer, and the iterative method NanoSatIterative using both NanoSat and NanoSatGlobal to solve the subproblems (with a fixed number of partitions). We use the following setup:

- **Hardware:** 13th Gen Intel® Core™ i7-1365U × 12, 32 GB RAM.
- **Solver:** CP Optimizer 12.7.0 with *docplex* library for Python.
- **Time limit:** 100 seconds (NanoSat, NanoSatGlobal), 1000 seconds (NanoSatIterative).

We can see in Table 1 that both NanoSat and NanoSatGlobal are efficient to solve NIMPH instances, as feasible solutions are found very often and a majority of instances can be solved to optimality before 100 seconds. However, the use of CP Optimizer *interval* variables and *sequence* global constraints within NanoSatGlobal increases the resolution time and decreases the number of optimal solutions on our instances, but this model is able to prove the infeasibility of the few instances that are not solved by NanoSat.

There is a higher contrast between these two approaches when they are integrated into NanoSatIterative. Table 2 highlights the differences in terms of resolution time and proportion of optimal solutions between NanoSat and NanoSatGlobal within NanoSatIterative. We can see that both methods improve the useful schedule time, with a mean increase of 6% for NanoSatIterative + NanoSat.

If the performances of both NanoSat and NanoSatGlobal are globally satisfactory for the NIMPH instances, such a performance gap between those two models is hard to explain. A deeper analysis of the models performance on bigger and more diverse instances is our main focus for the future work.

Table 1: Comparison of methods NanoSat and NanoSatGlobal: Mean resolution time; Number of instances solved; Number of instances solved to optimality; Number of instances proved infeasible.

Method	Mean time	Nb. feasible	Nb. optimal	Nb. Infeasible
NanoSat	6.4s	94	94	0
NanoSatGlobal	33.1s	75	74	4

Table 2: Comparison of methods NanoSat and NanoSatGlobal within NanoSatIterative: Mean resolution time; Mean instances solved to optimality; Mean useful schedule time; Mean upper bound on the useful time for the original instances (without adding tasks).

Method	Mean time	Mean optimal	Mean sched. use	Mean original sched. use
NanoSatIterative+ NanoSat	58.1s	89.8%	88.5%	82.5%
NanoSatIterative+ NanoSatGlobal	786.5s	11.2%	83.1%	82.5%

5 CONCLUSION

We presented NanoSatScheduler, a tool suited for on-board task scheduling in the context of nanosatellite missions. Two versions of our software are compared, and we highlighted that the use of CP Optimizer global constraints is less effective than a classical model for the NIMPH cyclic task scheduling problem, while demonstrating the efficiency of both methods to find optimal schedules for NIMPH instances. We also presented NanoSatIterative, a method to enhance solution quality by iteratively inserting new tasks into the instance and maximizing the useful schedule time.

Apart from improving experimental analysis, there are still some interesting research directions related to the OBC task scheduling, in synergy with NIMPH team needs. First, the iterative method could be improved to handle multiple partitions, solving a multi-objective problem of maximizing the useful schedule time of each partition. NanoSatScheduler could benefit from a Graphical User Interface, with the possibility of dynamically modifying the instances, to allow manual refining from the NIMPH team, while ensuring the feasibility or the optimality of each new solution. A variant of the NIMPH cyclic task scheduling problem with a variable time horizon would be very interesting to investigate, as the useful schedule time could be improved by reducing the time horizon instead of adding new tasks.

REFERENCES

Balbastre, P., Masmano, M., and Morales, V. (June, 2021). User manual. Technical Report fnts-xe-um-17b, Fent Innovative Software Solutions.

CNES (2021). CNES project libraries – Nanolab Academy. https://nanolab-academy.cnes.fr/en/janus.

Comer, D. and Fossum, T. V. (1987). *Operating System Design: Internetworking with Xinu*. Prentice Hall.

IBM (2023). IBM Docplex documentation. https://ibmdecisionoptimization.github.io/docplex-doc/cp/index.html. Accessed on October 9th 2023.

Laborie, P., Rogerie, J., Shaw, P., and Vilím, P. (2018). IBM ILOG CP Optimizer for scheduling: 20+ years of scheduling with constraints at IBM/ILOG. *Constraints*, 23:210–250.

Landrea, T., Maignan, M., Risson, A., and Roux, G. (2018). Spécification mission NIMPH. *ISAE-SUPAERO and CSUT*.

Li, C., Ding, C., and Shen, K. (2007). Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, pages 2–es, San Diego, CA (USA).

Masmano, M., Ripoll, I., Crespo, A., and Metge, J. (2009). XtratuM: A hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, volume 9, Dresden (Germany).

Potts, C. N. and Kovalyov, M. Y. (2000). Scheduling with batching: A review. *European Journal of Operational Research*, 120(2):228–249.

Wikum, E. D., Llewellyn, D. C., and Nemhauser, G. L. (1994). One-machine generalized precedence constrained scheduling problems. *Operations Research Letters*, 16(2):87–99.

Windsor, J. and Hjortnaes, K. (2009). Time and space partitioning in spacecraft avionics. In *2009 Third IEEE International Conference on Space Mission Challenges for Information Technology*, pages 13–20. IEEE.

Yu, W., Hoogeveen, H., and Lenstra, J. K. (2004). Minimizing makespan in a two-machine flow shop with delays and unit-time operations is NP-hard. *Journal of Scheduling*, 7:333–348.