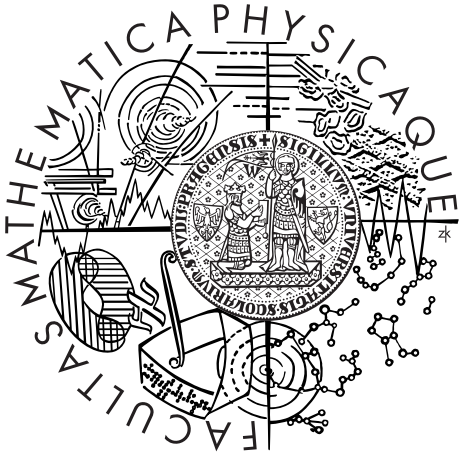
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Martin Kolombo

Planning operations of space probes

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: prof. RNDr. Roman Barták, Ph.D.

Study programme: Computer Science

Specialization: Theoretical computer science

Prague 2014

First of all I would like to thank my supervisor, Roman Barták, for the provided guidance in both the technical aspect of the thesis in developing the presented solutions and the soft aspect of the thesis in writing the actual text.

I would also like to thank my parents for supporting me during the studies and never doubting that I would finish them successfully.

Lastly, I would like to thank the ESA, in particular Simone Fratini and Nicola Policella for presenting the MEX challenge which made a great and challenging topic for a thesis and for their assistance in clarifying the problem.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Planning operations of space probes

Autor: Bc. Martin Kolombo

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: prof. RNDr. Roman Barták, Ph.D.

Abstrakt: Práce popisuje řešení komplikovaného rozvrhovacího problému z problematiky vesmírných misí. V práci je popsán problem rozvrhování operací na družici Mars Express Orbiter, který byl původně prezentován jako zadání soutěže vypsané Evropskou vesmírnou agenturou. Práce popisuje a srovnává dva různé přístupy řešení popsaného problému. Prvním řešením je speciálně pro daný problém vyvinutý rozvrhovač, který pracuje na principu přiřazování akcí do časových oken a využívá techniky lokálního prohledávání. Druhé řešení modeluje problém pomocí programování s omezujícími podmínkami (CP) a pro výpočet řešení používá SICStus Prolog. Oba přístupy jsou v závěru práce experimentálně ověřeny. U obou přístupů se podařilo vytvořit funkční řešení. Závěrem práce je, že obecnější přístup pomocí CP je i bez složité heuristiky schopný vytvořit velmi kvalitní rozvrhy, ale selhává pro malou podmnožinu vstupů. Specificky vyvinutý rozvrhovač je schopný díky lokálnímu prohledávání řešit větší velikost vstupu, ale produkované rozvrhy jsou méně kvalitní.

Klíčová slova: rozvrhování, omezující podmínky, plánování, mars express

Title: Planning operations of space probes

Author: Bc. Martin Kolombo

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D.

Abstract: The thesis addresses a complicated real-world scheduling problem from the space operations environment. The Mars Express Orbiter scheduling problem was first presented as a challenge by the European Space Agency. The thesis compares two different solutions. The first solution is an ad-hoc scheduler that is based on scheduling actions into a set of time windows and heavily utilizes local search techniques. The second solution models the problem as a constraint satisfaction problem (CSP) and uses the SICStus Prolog constraint programming solver to find a solution. Both schedulers are experimentally evaluated and the results are compared. Both approaches were able to provide a working solution. The conclusion however states that the more generic CSP based approach was capable of producing higher quality schedules even without a complicated heuristic. It however to compute the schedule for a small subset of inputs. On the other hand, the ad-hoc scheduler was capable of solving larger inputs but the produced solutions are not as good.

Keywords: scheduling, constraint programming, planning, mars express

Contents

Introduction

Space operations are always at the pinnacle of the current engineering technology. Although we are not in the moon-race era anymore, there are many active space missions at present aimed at making new scientific discoveries. Since direct control of distant space missions is often difficult because of the time it takes the signal to reach the space craft from the control center, the distant space missions require to be largely autonomous. One possibilty of achieving that is planning/scheduling of all operations offline on Earth and then uploading the plans to the distant space craft.

As space missions are very expensive and challenging to perform, there is a great pressure on the reliability and safety of the used technologies. This constraints the plans with a lot of safety procedures designed to give the mission control a chance to re-estabilish connection with a space craft if anything goes wrong and manually fix the problem. There is also a great pressure to show results from the scientific space missions and perform as many experiments as possible on the existing space craft.

All of the above makes creating a good control schedule very difficult by hand, which makes it an ideal task for computer based planning and scheduling. One of the current missions of the European Space Agency is the successfull Mars Express mission, which will be the topic of this thesis.

1. Problem introduction

1.1 The Mars Express Mission

1.1.1 Mission Background

The Mars Express (MEX) mission is a successful space exploration mission being conducted by the European Space Agency (ESA). The mission originally consisted of two vehicles (the Mars Express Orbiter, and the Beagle 2 lander) that reached Mars orbit at the end of December 2003. The Beagle 2 lander failed to land safely on Mars and was declared lost in 2004. While the lander mission failed before it could gather any data, the orbiter was extremely successful and is performing scientific experiments since early 2004, generating 2-3Gbit of scientific data per day. The orbiter mission was therefore already granted 5 mission extensions with the latest due in 2014 [1].

1.1.2 The Orbiter space craft

In this section, we will look at the orbiter space craft in more details. We will briefly cover the scientific instruments (or payloads) it carries. Then we will look at the data storages used to collect data from scientific observations. Lastly, we will cover how the space craft is controlled and how it communicates with the mission command on Earth.

1.1.2.1 Payloads

The scientific instruments equipped on the orbiter are referred to as payloads. The space craft is equipped with these 7 payloads [7]:

- ASPERA - Energetic Neural Atoms Imager
- HRSC - High Resolution Stereo Camera
- MARSIS - Mars Advanced Radar for Subsurface and Ionosphere Sounding
- OMEGA - IR Mineralogical Mapping Spectrometer
- PFS - Planetary Fourier Spectrometer
- SPICAM - UV and IR Atmospheric Spectrometer
- VMC - Visual Monitoring camera

All payloads except for VMC are still operational and experiments are actively performed on them.

Packet Store	Payload	Capacity (Bits)
SI	SPICAM	312 475 468
AS	ASPERA	464 388 096
MI	MARSIS	800 014 336
PS	PFS	890 109 952
OM	OMEGA	2 129 395 712
HR	HRSC	4 262 772 736
AX	HRSC	25 378 816

Table 1.1: Payloads data stores

The space craft is also equipped with a Solid State Mass Memory (SSMM) device to store data on. The SSMM is further subdivided into packet stores of finite capacity as displayed in the table 1.1 The HRSC payload is the only one with two accompanying data stores. This is because the former VMC data store was reassigned to HRSC after the VMC payload was decommissioned. On top of these packet stores which are tied to the payloads, there are several packet stores used to store house-keeping (maintenance) data. We will list these in the following section.

1.1.2.2 Space craft control

The spacecraft is remotely controlled from the Earth by uploading Tele Commands (TCs) to the space craft. The TCs are either stored in a specific data store on the SSMM or they are stored in the Mission Timeline (MTL). The MTL can only hold 150 TCs at any given time and the TCs have to be ordered in sequence of execution. The space craft then performs the TCs directly from the MTL. A specific TC is used to trigger the transfer of a set of TCs from the SSMM into the MTL. To our understanding, this transfer of TCs can be performed in parallel with any other space craft operations.

The spacecraft also produces an amount of maintenance (or house-keeping) data during its regular maintenance operation. This data is then collected in the data stores shown in table 1.2 on the SSMM (these are disjunctive to the payload data stores described earlier) [7].

Packet Store	Capacity (bits)
AC	900 000
EV	1 050 000
DM	30 000 000
DI	25 000 000
AO	60 000 000
PH	20 000 000

Table 1.2: House-keeping data stores

1.1.2.3 Communication

We recognize two types of communication between the space craft and the command center. Uplink (UL) is a term for uploading data from the command center to the space craft. Downlink (DL) on the other hand refers to the command center downloading stored data from the craft. Both uplink and downlink can be conducted in parallel to each other. Uplinks consist of TCs being uploaded on the space craft in the form of Master timeline Detailed Agenda File (MDAF) files. Downlinks can either mean downloading scientific data from the payload packet stores, or downloading the maintenance data from the house-keeping data stores. The housekeeping data are produced in packets that cannot be further subdivided when downlinking. It is thus always required for a full data packet to be transmitted before the transmission can be terminated.

For the space craft to establish and maintain a connection, two conditions must be met. Firstly, the craft needs to be pointed at Earth to align its communication equipment. Secondly, a communication ground station needs to be available on Earth in that time. The Ground Station Availability (GSA) windows depend on the relative alignment of Earth and Mars and thus can be known in advance. Each ground station also has a finite bitrate by which it can communicate. Every communication operation requires the exclusive use of a single ground station. A communication window cannot be shorter than 2 hours. Switching groundstations to communicate with takes 5 minutes.

2. Problem definition

The previous chapter introduced the Mars Express mission and described the real-life operations of the Mars Express Orbiter. In this chapter, we will define the problem that we will later discuss in the thesis. The definition of the problem is based on the ICKEPS challenge described in [7] and was further adjusted to better fit our understanding of the real Mars Express operations. There were some areas, however, where the definition of the problem in [7] was too vague. In these cases we had to define the problem ourselves based on our current knowledge or in some cases we were able to obtain additional information from the European Space Agency (ESA).

2.1 The Problem

We are trying to solve a complete Mars Express (MEX) scheduling problem. This means that we are automating the whole process from taking the experiments that are to be scheduled to having a complete and detailed schedule of all MEX operations. All of the current solutions of the MEX problem only deal with parts of the scheduling process. We will look at these partial solutions in chapter 3.

Our solution is motivated by eliminating the need for any manual planning other than selecting the set of experiments that the researchers want to perform. This obviously adds to the complexity and challenge of the problem.

2.2 The Inputs

There are three different input files for the problem: Payload Operations Request (POR), EVTM[1] and Ground Station Availability (GSA). While these mimic the the input files for the real problem, the real file format is very complicated and ESA did not permit us to make the contents public. Therefore we are using files of the format specified in attachment A.

The POR file is essentially just a list of experiments to perform. Each POR is characterized by the payload on which the POR should be performed and then a list of data productions. A data production specifies a packet of data produced by the experiment at a specific time offset. The offsets are defined in ms and are relative to the space craft's passage through the orbit pericenter.

The experiment starts with the earliest offset and ends with the latest offset. The experiment also cannot be paused and then resumed later, it must always be performed whole at once. Similarly, each data packet can only be downloaded whole at once. The data packets can however be downloaded in any order.

[1]The full meaning of the EVTM acronym was never explained in the challenge description. We are therefore just using the acronym by itself in the thesis.

The EVTM file format describes the individual orbits of the space craft by stating times of passage through the orbit apocenter and pericenter. The passage through either apocenter or pericenter is called an orbit event. Each orbit event also has a defined height above Mars surface in the EVTM file. The height is relevant as a soft constraint for some of the experiments. We are not considering orbit height in our solution.

The last input is the GSA file. The file simply lists all available communication times in the form of ground station windows. Every such window consists of the ground station identifier (simply a string), a fixed bitrate and the window start and end times in ms.

While we obtained the real data-sets from ESA, all our testing data is self-generated and was just based on the ESA data. This is because we were requested by the ESA not to distribute the data further. We have attempted for similar distribution of experiments, events and ground station events in our data, but it must still be kept in mind that the data is not exactly the same as real data.

2.3 The Constraints

In this section we will look at the various constraints of the problem. We will at first briefly cover the actions that the orbiter can perform. Then we will describe our model of the Space Craft (S/C) control and explain the S/C pointing. Then, we will look at each action type in turn and describe the constraints tied to each action. Lastly, we will shortly cover the soft constraints originally defined in the ICKEPS problem.

2.3.1 Orbiter control

2.3.1.1 Actions

For the purpose of this thesis, we differentiate between four different types of actions that the orbiter can perform : science, maintenance, communication and pointing transition. By science action, we understand performing any planned POR on any instrument. Maintenance actions are regular self-checks performed by the probe during which maintenance data is generated. Communication actions are either downlinks of data produced during the S/C operations or uplinks of Tele Commands (TCs) from Earth. Pointing transition actions will be described in detail in the next section. For now it suffices to say that they change the S/C orientation relative to Mars/Earth. No action can be interrupted once it has been started.

2.3.1.2 Space-Craft Control

As we said in Chapter 1, all S/C actions need to be pre-planned and uploaded on the craft in the from of TCs. We also mentioned that the TCs are uploaded in the form of an Master timeline Detailed Agenda File (MDAF) file. We also covered the Mission Timeline (MTL), which sequentially orders the uploaded TCs.

In our model, we simulate the upload of each TC as a separate uplink action. We however do not simulate the transfer to MTL or even the MTL itself. The reasoning behind that is simple. Since the transfer to MTL can be done at any time and in parallel to any other S/C operations, the problem to schedule such transfers is trivial.

The only real problem remaining is to uplink the TCs to trigger the transfer to MTL. The communication within a single GSA window can be however performed in both directions in parallel. This means the orbiter can perform both uplink and downlink actions at the same time. The bitrate is also symmetrical (same uplink speed as downlink speed). Since we do not know the real size of the TCs we assume them to be 1000 bits each.

We also only consider a single TC per action, no matter the action type. Lastly we do not simulate any maximum capacity of TCs in the S/C's Solid State Mass Memory (SSMM). We were specifically told by the ESA that the capacity is 'sufficient'.

In the real problem, the orbiter is never in a 'clear' state. There are always operations being performed and any plan needs to take the previous orbiter state into account. In our solutions, we always assume that all SSMM data stores are completely empty and the orbiter is in the Earth pointing. While this is a simplification of the problem, we could easily account for non-empty SSMM data stores by having the amount of data as one of the inputs. We could then generate downlink actions to download this data during the schedule. The different pointing could be addressed in a similar manner. We will explain the pointing in the next subsection.

Also one of the constraints in our problem is that we will download all produced data before the end of the schedule. Therefore the schedule following a schedule made by our method will always start in a 'clear' state.

2.3.1.3 Pointing concept

The orbiter state can be described by two variables. The amount of data currently stored in each data store on the SSMM and the S/C pointing. We differentiate between three pointings : Earth, Inertial (NAD) and Fixed (FIX). The latter two pointings are both different kinds of alignment of the S/C towards Mars.

The NAD and FIX pointings both have a maximum duration for which the S/C can remain in the pointing. The durations are specified in the table 2.1.

Pointing	Duration
NAD	68min
FIX	90min

Table 2.1: Pointing durations

The duration for the Earth pointing is not limited.

To change pointing, the S/C has to perform a Slew or Pointing Transition Action (PTA). The PTA takes 30 minutes to perform and is otherwise unconstrained. This makes the PTAs not known in advance and means that PTAs have to be planned dynamically.

2.3.1.4 Orbiter actions

The most important action of the orbiter is a Payload Operations Request (POR). A POR is always performed on one of the 6 operational instruments. PORs on different instruments can also be performed in parallel. Every POR needs to be performed at a specific time offset from the S/C orbit's pericenter. This offset can differ from POR to POR even on the same instrument. We also model the data produced by the POR in a 'pessimistic' way - all data is produced at the beginning of the experiment. Every POR also requires the S/C to be in a specific pointing (either NAD or FIX, depending on the payload). PORs are never performed in the Earth pointing.

The next important actions to be scheduled are communications. There are two types of communication actions : downlink (DL) and uplink (UL). Every communication action is characterized by the amount of data it transfers (in bits). As was mentioned earlier, DLs and ULs can be performed in parallel. In order for any communication action to be performed, the S/C needs to be in the Earth pointing and a GSA window needs to be available. There is also a soft-constraint that we try to keep in mind in our solutions. The data produced by the S/C operations should be downloaded as soon as possible without unnecessary delays. Also the TCs to perform the planned actions should be uploaded with sufficient time reserve. Lastly there is a requirement for a four hour reserve communication window every 24 hours. This window can be split into two windows 12 hours apart if necessary.

The S/C also needs to perform regular maintenance actions. A maintenance action's duration is 90 minutes and it should be scheduled close to the orbit apocenter. The maintenance actions should also be scheduled 2 to 5 orbits apart. The maintenance action needs to be performed in the Earth pointing. Our understanding is that these maintenance actions produce the house-keeping data mentioned in chapter 1.

Non-mandatory constraints There was also a number of non-mandatory constraints defined in the problem:

1. Backup uplink windows
2. Keeping a safety margin in SSMMs agains accidental overwriting
3. Downloading housekeeping data generated during maintenance
4. Minimizing time between PORs and downlinks of the produced data
5. Scheduling experiments with respect to the lighting (day/night) requirements of the instruments

6. Scheduling experiments with respect to altitude required by the instruments

2.4 Problem definition summary

As we can see from this chapter, the problem has many various constraints to consider which makes developing a solution challenging. It is also almost impossible to verify or create solutions by hand as it is very easy to miss one of the constraints. We can also see that the problem does not entirely fit the classic planning problem description or the classic scheduling problem. We have to consider both time and resources (of several types), which rules out classical planning. We however also do not have the list of all actions in the problem input (the PTAs are dynamically inserted when required), which rules out classical scheduling.

The next chapter will look at the related works about this problem and at the existing solutions used by the ESA

3. Related works

The Mars Express (MEX) planning problem was at first being solved manually by the MEX engineers in charge of the project. Later on, two tools were developed to automate the planning process (at least in part). These tools are MEXAR and RAXEM. In this section, we will look at both and describe the relevant planning process as well as the algorithms used by the tools.

3.1 RAXEM Uplink planner

3.1.1 Uplink problem

The uplink problem, as defined in RAXEM [10] is in some respects more complicated than the problem presented in this thesis. On the other hand, the RAXEM tool is then only used to schedule uplinks of Tele Commands (TCs) on the MEX orbiter while this thesis is attempting to create an integrated planner for all mission operations.

The main concern when uplinking TCs to the spacecraft is safety. The MEX always needs to be left in a safe status (Earth pointing) with all unnecessary instruments switched off after every batch of uplinked TCs. For this reason, the TCs are generated in Master timeline Detailed Agenda File (MDAF). An MDAF is a self-contained set of TCs that leaves the Space Craft (S/C) in a safe state after all commands have been performed. Such an MDAF file can contain several thousand TCs, but experience has shown that an average file contains between 250 and 300 TCs. The number of TCs that can be uplinked at one given time is then limited by the following constraints:

- The size of available uplink window (determined by ground station availability)
- Available Mission Timeline (MTL) capacity at the given time (The MTL can hold 3000TC in total).
- MDAFs with earlier execution times need to be uplinked sooner.
- Uplinked TCs need to be confirmed as stored to MTL by the S/C.
- A backup uplink window needs to exist in case the uplink fails for any reason.

These constraints result in the MDAF file usually being split into several smaller files by an automated process. These several files are then scheduled to be uplinked individually (in order).

3.1.2 Manual planning

The planning process was at first achieved in a fully manual fashion. The planning team used a Gantt chart-style diagram, manually drawn on paper. The chart linked uplink times with the execution times of each MDAF with a backup window being considered as well. The output from this process was then manually transferred to the Spacecraft Controller, specifying MDAFs to be uplinked with their primary and secondary uplink windows. The result was re-calculated for confirmation. This process was used for 3 years even though it was extremely time consuming and did not result in fully optimal plans.

3.1.3 RAXEM

The RAXEM tool automates the scheduling of uplinks process albeit it still requires human interaction to produce the final schedule. The tool takes the following input:

- MDAFs from the previous planning period (to determine the initial status of the MTL buffer)
- MDAFs for the current planning period
- a file listing the useable uplink windows, which is automatically generated from a medium-term planning process.

The user can also set constraint flags required for the solution:

- the use of secondary uplink windows
- the employment of full confirmation of the uplinked telecommands.

The application then attempts to generate a solution based on these constraint flags using constraint resolution techniques. If a solution cannot be found, the application will attempt to change the flags : a) without a secondary window, and b)without full confirmation. If a solution still cannot be found, the user is alerted and can decide to split the files into smaller segments which will fit into either the MTL capacity or the uplink windows depending on which is the deciding constraint. A final rollback option is to exclude some operations from the S/C schedule thus reducing the number of MDAFs to be uplinked.

A graphical representation of the resulting plans is provided for the user as well as the option of exporting the data in a human readable format for the on-duty S/C controllers to execute.

3.2 MEXAR2 Downlink Planner

The MEXAR2 is an automated tool [5] for scheduling downlinks (or data dumping activities) for the MEX. This tool is capable of creating a downlink schedule from a pre-scheduled set of Payload Operations Requests (PORs) and communications windows. The problem is represented as an optimal network flow problem for which a known algorithm is used to retrieve the solution.

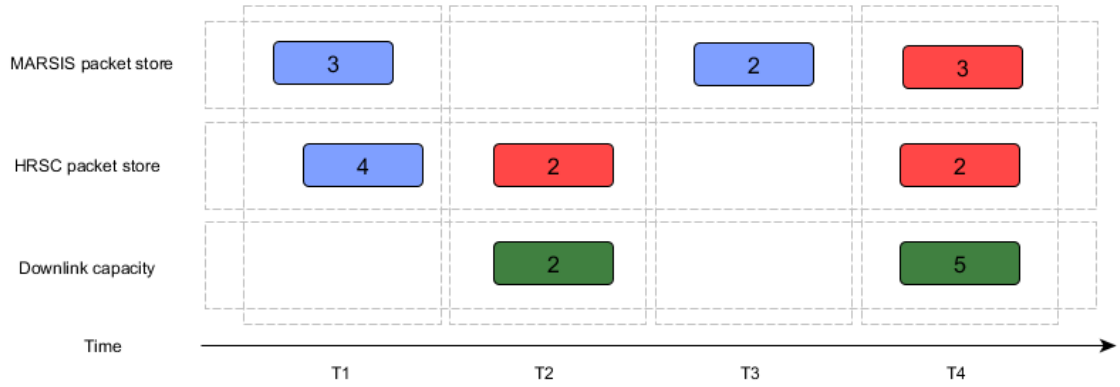


Figure 3.1: Example data downlink schedule

The basic input for creating the flow network are the intervals in which the MEX can communicate (ON intervals) and in which it produces scientific data (OFF intervals. The on and off intervals are presumed to be alternating. Each interval is then represented with a node in the flow network. Two arbitrary nodes (source and sink) are then added to the network.

The interval nodes will then have two input edges each (generated data and data already present in memory) and two output edges (downloaded data and data kept in memory to be downloaded later). The capacity for the input (data production) edges is simply the amount of data produced by the experiment. The capacity for the intermittent edges (keeping data until next interval) and the output edges from the "OFF" intervals is the packet store's capacity. The output edges from the "ON" intervals to the sink (downlinks) then use the amount of data that can be downloaded in their respective interval as capacity.

Figures 3.1 and 3.2 illustrate the principle. The figure 3.1 shows a very simple schedule where blue rectangles represent units of data generated by experiments, red rectangles represent units of data downloaded from the packet store and green rectangles represent total units of data downloaded over the specified time interval. In the example, we can see that after all downloads are completed, there are still 2 units of data left in the MARSIS packet store to be downloaded at a later date. Figure 3.2 then shows a flow network illustrating the same problem. Blue edges represent data generated on the packet stores, red edges represent download of data from the specific packet stores and green edges represent total download of data over time. Each edge is labeled by its flow and capacity (in parenthesis). Black edges then represent overflow of packet stores from one time interval to another.

This representation of the problem allows, as the authors say, to use state of the art maximum flow algorithms to obtain the solution. It should be noted however, that the tool itself does not attempt to perform all optimization regarding other constraints of the problem (overconstrained inputs, prioritization of data, etc...). This means human interaction is still required to obtain the final plan. The engineer is able to tune various parameters of the algorithm to obtain several solutions to the basic problem, out of which he can pick one based on further

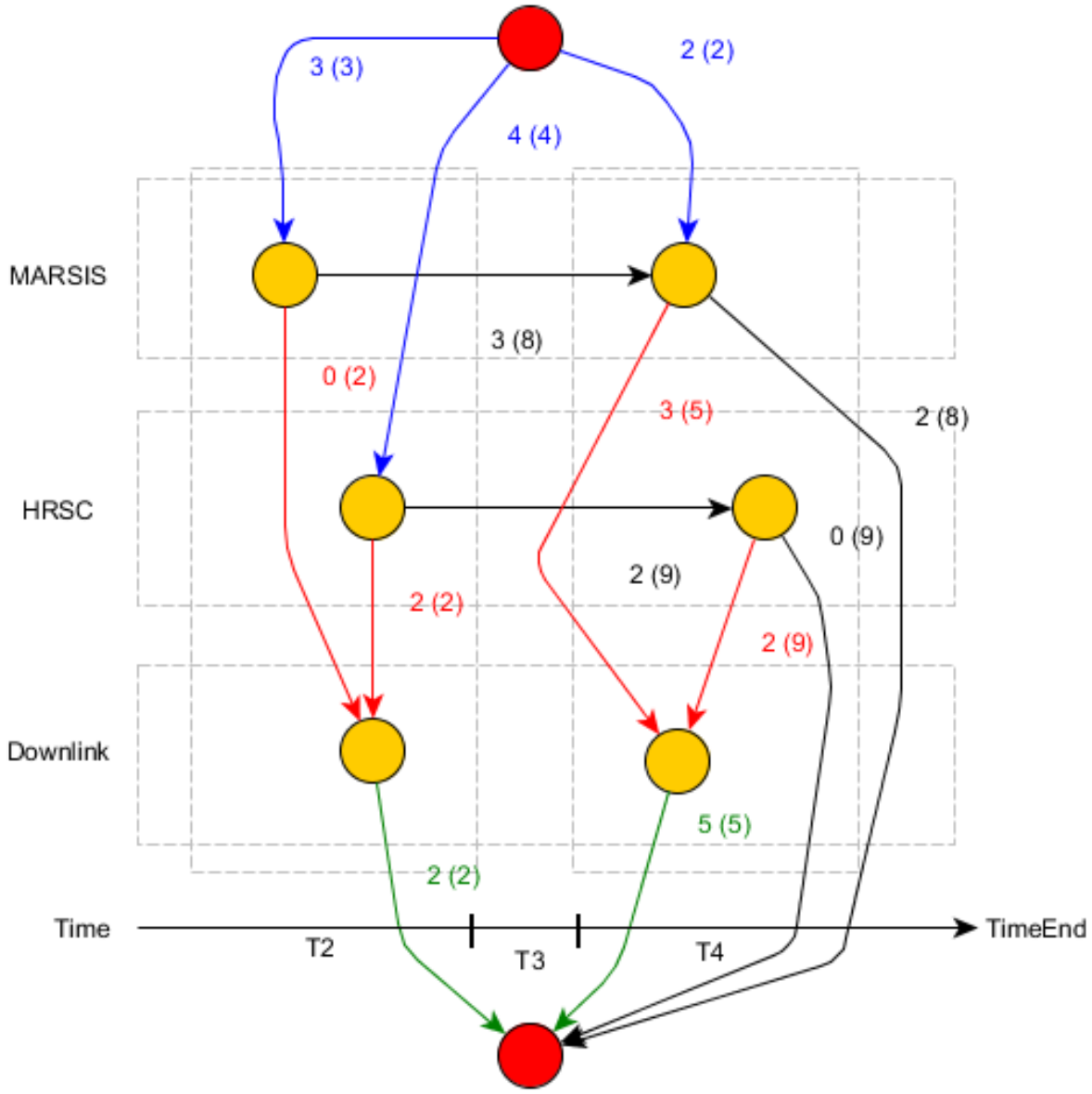


Figure 3.2: Example data flow network representing schedule in figure 3.1

criteria. The authors do not describe what are the exact parameters, but we can safely assume that they will entail prioritization of some data over other, maximizing certain downlink intervals or removing experiments to be performed.

4. Ad-hoc planner approach

Citation notice Note that this whole chapter cites heavily from a previously written article describing this solution [8] for the original ICKEPS challenge. The article was presented as a solution to the challenge, but was never fully published beyond that.

The first attempted solution for the Mars Express (MEX) problem as described in chapter 2 was to develop an entirely ad-hoc planner. This decision was based on the fact that the problem does not fit neatly into the planning or scheduling category but is somewhere in between and adapting any of the existing schedulers/planners to fit the MEX problem was deemed too complicated with dubious possible results. A decision was therefore made to create an ad-hoc solution to the problem.

4.1 Time-Line concept

The core solving concept to the entire solution is what we define as a Time-Line. A Time-Line is a set of linearly ordered windows where each window has a non-zero duration. The Time-Line itself has start and end times and spans the entire period for which we are planning the operations. The windows then always span the entire length of a Time-Line without any gaps. Note that the number of windows making up a Time-Line can change without affecting the total duration of it. This is also a key principle used in the algorithm.

The windows in a Time-Line contain the actions themselves. Each window can contain either no actions or any number of actions of the same type. Each window is then assigned a window type based on the actions it contains. The window and action types are:

- science
- pointing
- maintenance
- communication

To define the Time-Line more formally:

T1 A Time-Line consists of a non-empty, linearly ordered sequence of windows.

T2 For each window and the window immediately following it, there is no time gap between them, i.e. the start of the following window equals the end of the previous one.

T3 A Time-Line covers the entire planning period. The duration of the Time-Line may not change during the course of the algorithm.

T4 A window is either empty or contains actions that can be performed under identical conditions.

T5 If a window contains actions, it wraps tightly around the actions contained by it.

T6 No action can span more than one window.

T7 Only science and communication windows can contain more than one action. Science windows can even contain several actions for a single payload.

We call a Time-Line consistent if the following conditions are met:

C1 An empty window has no empty neighbors.

C2 All actions present in the Time-Line satisfy all domain constraints. This means the Time-Line forms a valid MEX plan.

The algorithm always starts with an empty Time-Line (a Time-Line containing just a single empty window) which it then modifies using the three possible window operations: splitting, merging and adding actions. The actions are illustrated in figure 4.1.

Window splitting Any empty window can be split into two empty windows at any time. This creates two empty windows adjacent to each other thus violating the condition C1 and making the Time-Line inconsistent. A window already containing actions cannot be split.

Window merging Any two adjacent empty windows can be merged into a single empty window spanning the duration of both. This action is used to make the Time-Line consistent again in case the condition C1 is violated. Windows containing actions cannot be merged.

Adding actions The only requirement for adding an action into a window is that it can fit into the timeframe represented by that window or the current window can be extended into time occupied by adjacent empty windows (shortening the empty windows). The window is always stretched just enough so that the condition T5 is satisfied.

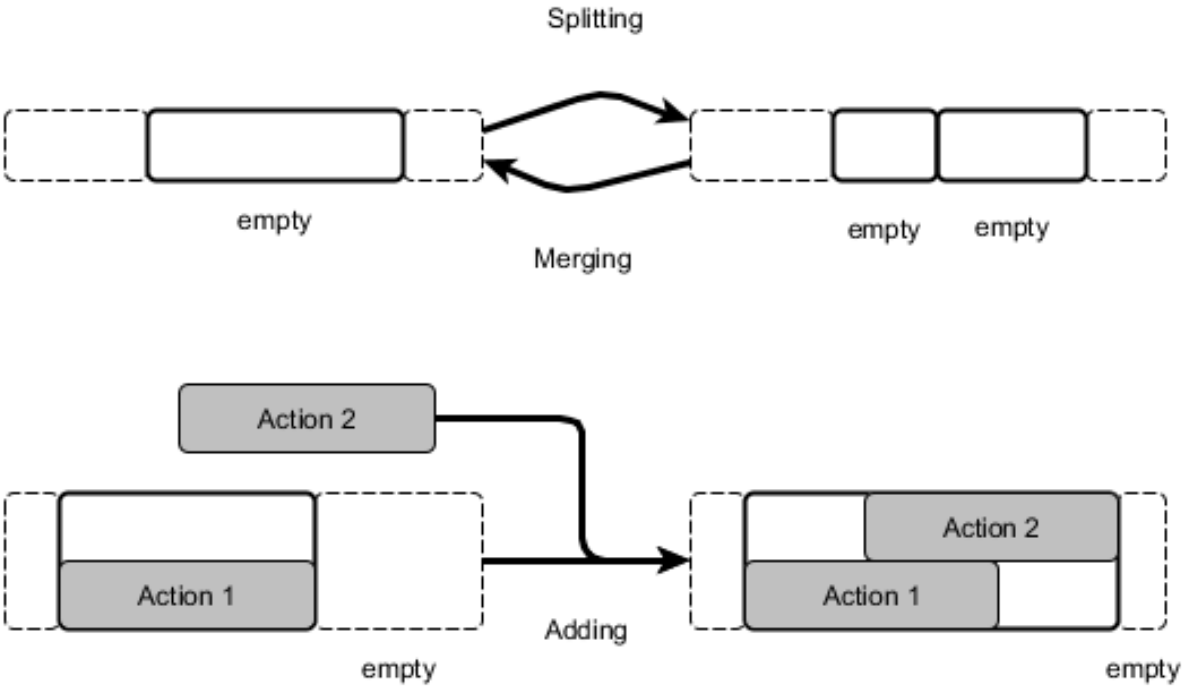


Figure 4.1: Window actions diagram. The action is always applied to the window highlighted with the thick border line.

4.2 The scheduling algorithm

Tasks The main goal of the algorithm is to schedule PORs, Maintenance and reserved communication actions into the timeline. All of these actions will be refered to as *tasks* in this chapter.

Related actions Each task requires a number of supportive actions for all MEX constraints to be satisfied. These may include a pointing transition, uplink of TCs and downlink of data. We will refer to these actions as *related actions*.

The algorithm can be divided into four main stages: preprocessing of PORs, scheduling tasks using depth-first search (DFS), scheduling related actions using local search and a postprocessing phase, which cleans up the schedule and adds trivial actions.

4.2.1 Preprocessing

The preprocessing phase:

- generates an appropriate number of maintenance tasks
- generates an appropriate number of reserved communications tasks
- generates related actions for each Payload Operations Request (POR) and each maintenance task

Since the only constraint on scheduling maintenance tasks is that the time distance between two neighboring maintenance tasks is between 3 and 5 orbits, we simply generate m maintenance tasks where $m = \lceil \frac{timelineLength}{4 * orbitLength} \rceil$.

The generated related actions are stored for use during the later stages of the scheduling process.

4.2.2 Task scheduling

The goal of the main scheduling stage is to allocate all tasks to time windows. The algorithm works in two phases.

1. Pre-schedule maintenance and reserved communications tasks.
2. Use DFS to schedule POR tasks.

4.2.2.1 Pre-scheduling

In the pre-scheduling phase, we roughly schedule all maintenance and reserved communications actions.

The pre-scheduling phase is very straightforward. Since we start with an empty timeline and the tasks scheduled in this phase have been pre-generated to fit into this timeline, we can simply schedule them from the earliest starting time to the latest and assume that they always fit. This stage is called pre-scheduling because the times of actions scheduled here are not final and can change during the later stages.

Algorithm 1 PreScheduleMaintenance(M,C,W)

- M is the list of all maintenance actions.
- C is the list of all reserved communication windows.
- W is the list of all timeline windows.

1: $w \leftarrow getFirst(W)$
2: $t_0 \leftarrow 0$
3: **for** $m \epsilon M$ **do**
4: $\quad t \leftarrow min\{t | canFit(m, w, t) \wedge (t - t_0) \geq 21h\}$
5: $\quad add(m, w, t)$
6: $\quad w \leftarrow next(w)$
7: **end for**

The pre-scheduling of maintenance actions is described in algorithm 1. The algorithm places maintanance actions not closer than 28h (4 orbits) apart. This is the average time difference between maintenance actions which need to be 3-5 orbits apart. The algorithm doesn't do any checking against already present actions as the timeline is assumed empty. The window w is always the last empty window in the timeline over the run of the algorithm. This is because the maintenance actions are scheduled left to right.

Scheduling of reserved comms actions works analogically except for a different time interval (15-24h apart). The maximum 24h spacing was specified in the problem definition. Since no lower bound was specified, we have chosen 17h (up to one orbit earlier). This gives the scheduler some flexibility to fit the actions into available Ground Station Availability (GSA) windows. We again do not expect the scheduling to fail as if a maintenance is occupying the slot, we can shift it by up to 7h (because of the 28h spacing).

The timeline at the end of this stage is consisting only of empty, maintenance and communications windows. The resulting timeline is then passed over to the next stage.

4.2.2.2 POR scheduling

The POR tasks are scheduled using a simple backtracking algorithm. The important fact is that with each POR, we attempt to schedule all of its related actions as well. If we later remove the POR during backtracking, we also remove all its related actions. This ensures that the timeline is kept in a consistent state after each DFS step. The ordering of PORs for the backtracking search queue is simply the ordering of the input data, no other heuristic is used.

The backtracking steps are as follows:

1. Select and remove the first unscheduled POR p from the queue.

2. Attempt to schedule p using the $SchedulePor(p, w)$ function (algorithm 5, where w is any window from the timeline, sorted by their start time in ascending order. Schedule p into first w where $SchedulePor(p, w)$ returns true.

3. If p was scheduled successfully, continue with step 1.

4. If p was not scheduled, backtrack to the previous POR and schedule it to a different time.

5. If there are no more times available, fail.

Consistency Note that because *SchedulePor* schedules the POR and its related actions and either adds all or none, the C2 timeline-consistency property is always satisfied at the end of iteration. C1 can easily be achieved locally when adding/removing an action. Thus the timeline is always consistent at the end of each iteration.

The next section will describe in detail the mechanism for scheduling both the POR and the related actions.

4.2.2.3 Local search

In this section, we will describe the actual algorithms used for selecting final times for all actions. The local search uses two main principles for scheduling actions into the plan: fitting an action into a selected time window and shifting actions already present in the plan to different times. Especially the shifting is important as it allows us to change times of previously scheduled actions to make room for current actions without the need to backtrack.

Algorithm 2 CanFit(a,w,t)

- a is the action to fit
- w is window to fit a into
- t is the expected start time of a

Require: $t \geq$ StartTime(w)

```
1: if t + duration(a) > EndTime(w) then
2:    if Next(w) is empty then
3:       return t + duration(a) ≤ EndTime(Next(w))
4:    else
5:       shiftRight ← GetWindowShift(Next(w),1)
6:       return t + duration(a) ≤ EndTime(w) + shiftRight
7:    end if
8: else
9:    return true
10: end if
```

Algorithm 3 GetWindowShift(w,dir)

- w is the window to shift
- dir is the direction of the shift: -1 left, 1 right

Require: w is not empty
1: $shift \leftarrow min\{GetActionShift(a, dir)|a \in$Actions($w$) $\}$
2: **if** Adjacent(w,dir) is empty **then**
3: **return** min($shift$,Duration(Adjacent(w,dir)))
4: **else**
5: **return** min($shift$,GetWindowShift(Adjacent(w,dir),dir)
6: **end if**

Fitting To see if an action a with a potential start time t fits into a window w, the $CanFit(a, w, t)$ function is used. The function is described in algorithm 2.

In short, a fits into w either when w already encompasses a or when w can be extended so that it encompasses a. We can extend w either by shrinking adjacent empty windows or by shifting adjacent windows containing action. Both of these are handled by the $GetWindowShift(w, dir)$ function described in algorithm 3.

Shifting To shift a window, all actions contained in the window have to be shifted as well. In our algorithm, we always shift all of the actions in the same window by the same $offset$ which is also the same offset by which the entire window is shifted. This ensures that the action ordering (relative to the window) remains the same and also that the window duration remains unchanged. This comes from the assumption that actions already present in the window are optimally scheduled in respect to the window, but the window itself may not be optimally scheduled in respect to the entire timeline.

The shifting itself is handled by functions $GetWindowShift(w, dir)$ (algorithm 3) and $GetActionShift(w, dir)$ (algorithm 4). Both functions expect dir (direction) to be either -1 (left = sooner) or +1 (right = later) which is a convention used everywhere in the planner.

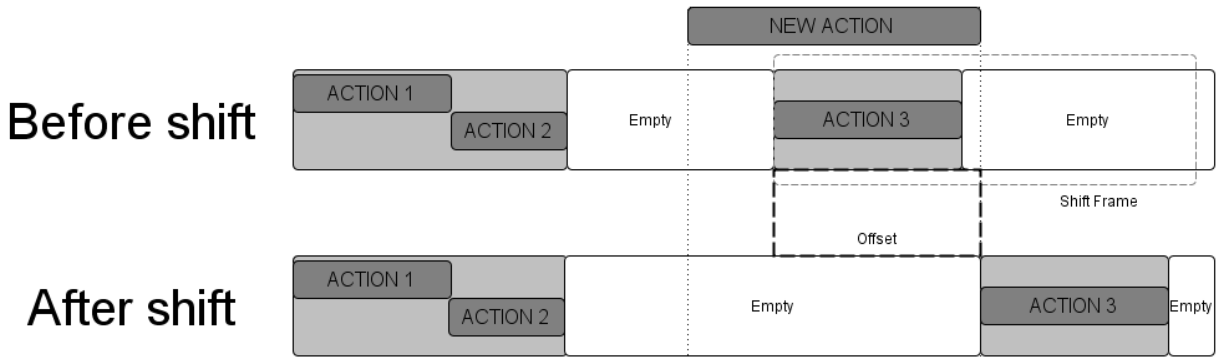


Figure 4.2: A diagram illustrating window shift

Algorithm 4 GetActionShift(a,dir)

- a is the action to shift
- dir is the direction of the shift: -1 left, 1 right

Require: $dir \in \{-1, 1\}$

$w \leftarrow$ GetWindow(a)
$t \leftarrow$ Start(a)
$p_t \leftarrow t$
continue $\leftarrow$ true
while *continue* **do**
  $p_t \leftarrow t_0 : (t_0 \epsilon w,\ t_0$ maximizes $(dir * (t_0 - t))$ and ConstraintsSatisfied(a,$[t,t_0]$))
  if $dir = -1$ **then**
    continue $\leftarrow (p_t =$ StartTime(w))
  else
    continue $\leftarrow (p_t =$ EndTime(w))
  end if
  continue $\leftarrow$ *continue* and not isEmpty(w)
  $w \leftarrow$ Adjacent(w,dir)
end while
return $|p_t - t|$

The basic principle of shifting is that a window can be shifted as far as the most-constrained action allows, which is precisely what $GetWindowShift$ is calculating. More so, a window can only be shifted into a space occupied by an empty window. If the adjacent window is not empty, it has to be shifted as well, which is catered to by the recursion in $GetWindowShift$. It should be noted that the recursion ends at the first empty window and thus the resulting offset for shifting is always at most the duration of the first empty window encountered in the direction we're shifting to. While this seems an unnecessary limiting factor, it works well in practice and reduces the number of checks required for each shift significantly. The principle of shifting is illustrated by the diagram 4.2.

Whenever we refer to shifting later in this chapter, we will always have this process in mind. If we say that we will shift a window (an action resp.), we mean running the function $GetWindowShift$ ($GetActionShift$ resp.) and then shifting the window (action resp.) by an offset not greater than the returned value.

Scheduling PORs As we stated before, each POR is scheduled with its related actions. The related actions for each POR are:

- downlinks
- uplinks
- Pointing Transition Actions (PTAs)

The problem definition states that uplinks can run in parallel to downlinks and that they are significantly smaller than downlinks. Since we do not have an upper constraint on the number of Tele Commands (TCs) in the internal orbiter memory, we can assume that we will be able to fit most uplinks in parallel to existing downlink windows. The only problem are uplinks that happen before the first downlink occurs. We solve this by reserving some time for them at the beginning of the plan. This makes the whole problem of scheduling uplinks trivial for us. For this reason, the uplinks are scheduled in the post-processing phase. This phase thus only deals with scheduling the related downlinks and PTAs.

Algorithm 5 SchedulePOR(*por*,*w*)

- *por* is the POR action to schedule
- *w* is the window to schedule *por* into

```
1: if w does not cover correct orbit stage for por then
2:    return false
3: end if
4: t ← time of desired orbit stage in w
5: added ← false
6: pta ← NULL
7: if CanFit(por,w,t) and Compatible(por,w,t) then
8:    if Pointing(w) = Pointing(por) then
9:       add(w, por, t)
10:      added ← true
11:   else
12:      pta ← new PTA(Pointing(por))
13:      if AddPointing(w,pta,t−PTAduration) then
14:         add(w, por, t)
15:         added ← true
16:      end if
17:   end if
18: end if
19: if added then
20:   if ScheduleRelatedActions(por) then
21:      return true
22:   else
23:      remove(w,por)
24:      if pta! = NULL then
25:         remove(window(pta),pta)
26:      end if
27:   end if
28: end if
29: return false
```

The algorithm for scheduling a POR and it's related actions is done in function *SchedulePOR* which is described in algorithm 5. Remember that this is the main algorithm representing one backtracking step or iteration.

The function $Compatible(por, w, t)$ is used when window w is not empty and contains only scientific actions. These actions cannot be shifted (as they need a fixed time offset from the pericenter), but actions can run in parallel on multiple instruments. The function *Compatible* checks if we can add the *por* into w without conflicting with another experiment on the same instrument. The function *Compatible* returns false for any other action types as PORs cannot be run in parallel with anything else than other PORs.

The logic of the function *AddPointing* will be explained several paragraphs later. PTAduration is the duration of any PTA and is a constant to the problem (30 minutes).

The function $ScheduleRelatedActions(por)$ used at the end of the algorithm takes each related action in turn (downlinks) and attempts to schedule it into the TimeLine either before or after the *por* as appropriate. The ordering in which the actions are scheduled is imposed by the temporal constraints of these actions. If any such related action cannot be scheduled, the timeline is returned to the previous state (all of these related actions are removed from the) and the function returns false. If all related actions are scheduled successfully, the function returns true.

Local search - Downlink actions The local search for scheduling downlink actions can be described in two simple steps.

- Find the first communication window following the window of the main action that the action can fit into (using the $CanFit$ function).
- Put the action in the communication window so that it has the minimum possible duration (this is done by selecting the GSA event with the highest bitrate that is intersecting the window).
- If the previous step fails (no such communication window is found), place the action into the first empty window that can fit the action and either has a correct pointing already or a PTA can be added to change it.

Local search - Pointing actions The pointing actions are special because they are not directly generated in the preprocessing phase but have to be generated on-demand whenever we need to add an action to a window with incompatible pointing. Their scheduling is further complicated by the fact that adding a pointing action to a window changes the MEX pointing for all succeeding windows until another pointing action is performed. On top of that, the orbiter can only maintain the NAD and FIX pointing for a limited amount of time. There are two cases, where this needs to be considered (assume that we are scheduling a pointing action *pa* into window w at time t and that *pa* sets pointing to p):

- A window w_1 exists (successive to w) that contains an action requiring a pointing p_1 different from p and there are no other pointing windows between w and w_1. Assume w_1 is the closest such window to w. In that case we will call w_1 a *conflict window* of w.

- p has a maximum duration shorter than the time to the next pointing window.

In either case, we address the problem by scheduling another pointing action pa_f using the algorithm described in algorithm 6.

The algorithm may add an *empty* pointing action. This happens when there are no constraints on the pointing of windows between w_f and w_1. The planner can later instantiate such empty pointing with a concrete value. This is a similar strategy to *lifting* used in classical backward planning [9].

The full algorithm for adding a pointing action p_a to window w before time t is described in Algorithm 7.

The result is that we are always adding the pointing transition just before the action that requires them (the t parameter). The algorithm also ensures, that any pointing conflicts are always resolved and the maximum duration constraint is maintained (by calling the function *PointingFix* when required). If either of these cannot be solved, the pointing is not added and the whole action needs to be scheduled elsewhere.

Algorithm 6 PointingFix(w,w_1,t,p,p_1)

- w is the window containing the original PTA
- w_1 is the window containing an action that requires a different pointing
- t is the end time of the original PTA
- p is the pointing set by the original PTA in w
- p_1 is the new desired pointing required by w_1

```
1: w_f ← w
2: while endTime(next(w_f)) ≤ t + maxDuration(p) and next(w_f) != w_1 do
3:    w_f ←next(w_f)
4:    if p_1 != ∅ then
5:       pa_f ← emptyPointingAction()
6:    else
7:       pa_f ← pointingAction(p_1)
8:    end if
9:    t_0 ← start(w_f)
10:   if CanFit(pa_f,w_f,t_0) then
11:      add(pa_f,w_f,t_0)
12:      return true
13:   end if
14: end while
15: return false
```

Let us demonstrate the above process in an example. See Figure 4.3 for reference. In the example, we are trying to schedule a new POR for the SPICAM instrument, which requires the Fixed (FIX) pointing. As we can see, the MEX is in the earth pointing in the timeframe, where we want to add the POR action.

Algorithm 7 AddPointing(w,p_a,t)

- w is a window to add the PTA to
- p_a is the PTA to add
- t is the wanted start time for the PTA

```
1: w ← split(w,t) {left portion of w before t}
2: OK ← false
3: if CanFit(p_a,w) then
4:    add(p_a,w)
5:    p ← TargetPointing(p_a)
6:    t ← StartTime(p_a)
7:    if ∃w_1 : w_1 is conflict window of w then
8:       p_1 ← RequiredPointing(w_1)
9:       OK ← PointingFix(w, w_1, t, p, p_1)
10:   else if maxDuration(p) ≤ ∞ then
11:      OK ← PointingFix(w, EndWindow(timeLine), t, p, ∅)
12:   else
13:      OK ← true
14:   end if
15: else
16:   merge(w)
17:   return false
18: end if
19: if OK then
20:   return true
21: else
22:   remove(p_a,w)
23:   merge(w)
24:   return false
25: end if
```

The arrows in the diagram represent the duration of the pointing state induced by the originating action.

To schedule the POR, we have to add a new PTA to change MEX pointing to FIX before the POR. The FIX pointing however has a limited maximum duration of 90 minutes. Moreover, adding the FIX pointing would break the required pointing constraints for the downlink action that is scheduled later. For this reason, we use the PointingFix algorithm described above to place the second earth PTA. As we can see in the resulting schedule, all pointing-related constraints are now satisfied. This process is very similar to how a general temporal and resource planner Filuta handles new events in timelines [6].

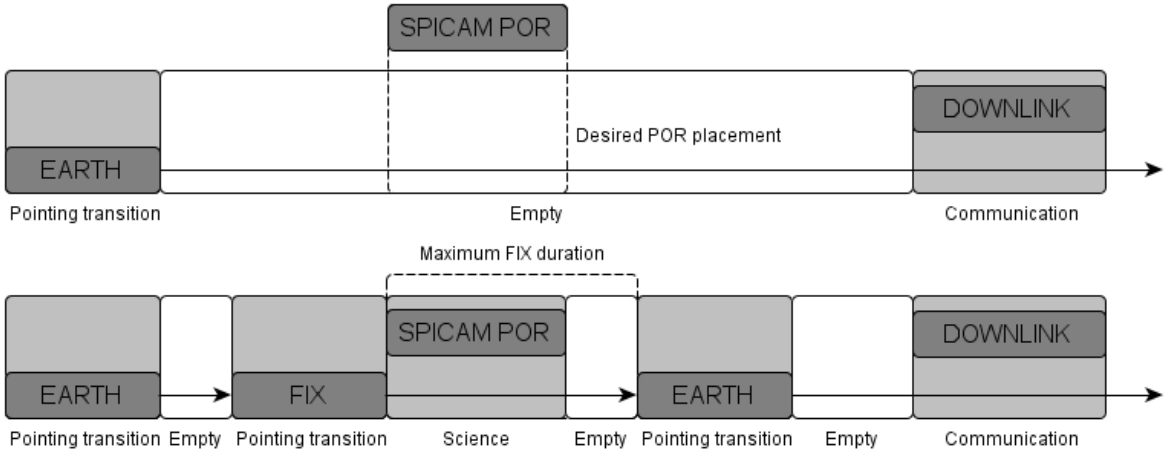


Figure 4.3: A diagram illustrating POR scheduling with pointing transitions

4.3 Postprocessing

Some simpler tasks were omitted in the previous steps because we address them in the post-processing phase. This allows us to keep the main search algorithm working with the least possible number of actions (in other words, with a smaller search space).

We facilitated the problem by not handling telecommands which instruct MEX to move a TC from its internal memory to the Mission Timeline (MTL). These commands can also run in all states (except Maintenance) and impose no other conditions. So their scheduling is also an easy task.

There is an issue with this approach however. Since every action executed by MEX has to have a command starting it (and most actions need one more for ending), the scheduling in postprocessing also has to schedule uplinks of these commands. The easiest way to overcome this was to tell the planner to leave about 25% of an uplink session empty (TCs are really small). These empty gaps can then be used in the postprocessing phase to schedule the needed TC uplinks. Since uplinks are short and there is a constraint to have a 2-hours-lasting uplink window every 12 hours, it seems having a quarter of the window empty isn't a problem.

Pointing postprocess Post processing will also instantiate any empty pointing transition actions with the EARTH pointing. This does not break any constraints as the empty pointing action is only created if no window after it requires any specific pointing. The EARTH pointing then has no limit on its duration, so there are no constraints on scheduling of the next pointing action. The algorithm only checks if the next pointing action has EARTH pointing and if it does, we can safely remove it to prevent doubling the transition actions.

5. Constraint Programming

In this chapter, we will at first introduce constraint programming and define a Constraint Satisfaction Problem (CSP). In the other part of the chapter, we will introduce the used solver - SICStus Prolog along with the properties we used in the solution.

5.1 Constraint Programming

Constraint Programming is a declarative technique for solving problems by encoding them as a set of variables and constraints imposed on them. Search and pruning techniques are then used to obtain a solution. A solution of a CSP is an assignment of values for variables such that all constraints are satisfied. We will briefly introduce the technique in this section.

5.1.1 Constraint satisfaction problem

A constraint satisfaction problem is defined as composing of: [3]

- a finite set of variables
- a finite set of variable domains
- a finite set of constraints

Variables Each variable describes one attribute of the solution (eg. a start time of a specific action in planning problems). The domain of a variable is an enumeration of possible values for the said variable (such as all positive integers). Variable domains do not need to be continuous integer intervals. In fact the domain may not even consist of integers.

Constraints A constraint is any relation imposed on a subset of variables. Constraint can have any finite arity. Constraints can be defined either explicitly (by enumerating compatible sets of values) or implictly by a math formula.

Constraints are a very natural way of describing a problem. A commonly known constraint can for example be "The sum of all angles in a triangle is 180 degrees". Constraints in a CSP are very generic and can express any relationship between one or more variables.

Solution A solution of a CSP is a complete and consistent assignment of values from variable domains to the variables. The solution has to be:

- complete : each variable has a value assignment
- consistent : every constraint in the problem is satisfied

In a CSP, we may be looking for any one solution, all possible solutions or an optimal solution. In case of optimal solution, we have to define an objective function expressing the quality of each solution.

Binarization Any CSP can be encoded only by using binary constraints. Unary constraints can be encoded directly into the domains. N-Ary constraints are then encoded by adding extra variables and constraints on those. More detail on this can be found in [4]. This property allows for only considering binary constraints in all solution algorithms.

5.1.2 Solution techniques

Search CP uses systematic search algorithms for obtaining a solution. The most common one is simple backtracking. A solution is found when all variables have specific values assigned. As we know, backtracking is guaranteed to either find a solution or to prove that a solution does not exist. This can however take a very long time for large problems as backtracking is exponential in time complexity (to the number of variables). This is addressed by consistency techniques.

The idea of a single step of the backtracking algorithm in CP can be described as follows:

1. Select a variable V_i from V. If all values for all variables have been tried, fail.
2. Assign value X_i to V_i from the domain of V_i.
3. If all values from the domain of V_i have been tried, backtrack to the previous variable.
4. If any constraint is not satisfied, backtrack to the previous step and assign a different value.
5. If all variables have a value assigned, return the current assignment as the solution.

Consistency Consistency techniques are used to prune the domains of unassigned variables to reduce search space. We will describe two basic types of consistency used in CSP: Node Consistency and Arc Consistency (AC) to illustrate the principle.

Node consistency Node consistency simply prunes the domain of a single variable based on unary constraints:

- The node representing variable X is node consistent $\Leftrightarrow$ every value in the variable's domain D satisfies all the unary constraints imposed on the variable X
- A CSP is node consistent $\Leftrightarrow$ all the nodes are node consistent.

Node consistency only needs to be enforced once before the start of backtracking, or when new unary constraints are added.

Arc consistency We will define AC for binary constraints as we have already stated that all CSPs can be binarized.

Arc consistency is defined as:

- The arc (V_i,V_j) is arc consistent $\Leftrightarrow \forall x \, \epsilon$ dom(V_i) $\exists$ a value $y \, \epsilon$ dom(V_j) such that the assignment $V_i = x$ and $V_j = y$ satisfies all the binary constraints on V_i ,V_j.
- CSP is arc consistent $\Leftrightarrow \forall$ arc (V_i,V_j) is arc consistent (in both directions).

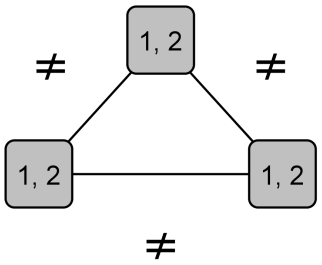


Figure 5.1: Example of an arc consistent problem that does not have a solution. We can assign values to any two variables in this diagram without breaking the constraint between them, but assigning any value to the third breaks one of the inequality constraints. The problem thus does not have a solution.

It should be noted that arc consistency is directional, i.e., arc consistency of (V_i,V_j) does not imply arc consistency of (V_j,V_i). We must however keep in mind, that a CSP can be arc consistent and yet have no solution. This can be clearly seen in figure 5.1.

Arc consistency needs to be updated during every back-tracking step by updating affected variable domains. We will not go into the details of how the arc consistency algorithms work, more on the most common ones can be read in [2]. There are also other, more powerful consistency techniques which can be used to further prune the search space at (usually) bigger cost in time complexity. Whether to use them or not usually depends on the precise nature of the problem.

Maintaining arc consistency is more expensive than node consistency, but on the other hand AC is more powerful than node consistency.

Search Heuristics As the basic backtracking is usually insufficient for large problems, CSP relies on heuristics to speed up the search. There are two basic types of heuristics for a CSP : Value ordering and Variable ordering.

Variable ordering determines the order in which the main backtracking search will select the variables for assignment. A common heuristic here is "most constrained first" which prioritizes variables that are included in the most constraints. A different heuristic could order the variables based on the size of their domains or the heuristic can be using some deep knowledge of the modeled problem to order the variables.

Value ordering heuristic determines the order in which values from the domain are assigned to a specific variable. A CSP can use a different value ordering

heuristic for different variables in the problem. The most common heuristics include assigning values in ascending/descending order, assigning values by interval halving, etc.

The search algorithm can also be further improved by making other choice points than assigning a specific value to a variable. This can be done by for instance restricting a certain variable to an interval of values by adding a constraint during the search.

5.2 SICStus prolog

In this section we will introduce the specific solver we have used to solve the problem - SICStus Prolog. The reason for describing it is that it offers several constraints that we rely on in the model. We will describe all of the used special constraints below.

5.2.1 Introduction

SICStus Prolog is a distribution of the well known logic programming language - Prolog. Prolog is well suited for a CSP for two reasons. Firstly, it's declarative, same as a CSP. Secondly, it uses backtracking to satisfy the predicates. Since backtracking is the primary solving algorithm for CSP, this makes Prolog a natural choice. The SICStus Prolog installation comes equipped with a library for solving CSP problems. More on SICStus prolog can be read on the official web page [11].

5.2.2 Special constraints

Table The table constraint is very simple. It defines possible combinations of values of several variables by enumeration. Best demonstrated by an example: let A, B, C be variables of a CSP. Table 5.1 then describes the constraint $((A + B = C) \wedge (C = 2)) \vee (A = B = C = 0)$. It might seem more effective to use the formula, however, disjunctions do not propagate during the search. This means that $A > 0 \wedge C < 2$ would not cause the constraint to fail until we try to assign a single value to both A and C. On the other hand the table looks at all its constraints globally when propagating, so it would immediately detect that the constraints cannot be satisfied and fail early on. We use table constraints heavily in our model precisely for the purpose of reducing disjunctions.

A	B	C
1	1	2
2	0	2
0	2	2
0	0	0

Table 5.1: Example table constraint

Cumulative The cumulative constraint represents cumulative scheduling with a single shared resource. Cumulative scheduling is a well researched topic, so we will just describe it very shortly. In cumulative scheduling, a set of actions A can share a single resource R, each action $a \in A$ consumes r_a resource units when it's being performed. The constraint imposes that the resource consumption at any given time is never larger than given capacity $cap(R)$. For $cap(R) = r_a \forall a \in A$ cumulative scheduling imposes that all actions are disjunctive. In our model, we use several cumulative constraints for restricting which actions can be performed in parallel and which are strictly disjunctive to others.

Cumulative constraints will be defined by the formula shown in 5.1. The formula simply contains a set of tasks in the given format and a number defining the resource capacity.

$$cumulative\left(\{task\left(Start, End, Consumption\right), ...\}, Capacity\right) \tag{5.1}$$

Element The element constraint can be usecd to look up a variable from a list of variables based on an index (which is also a variable). The constraint is shown in 5.2 where L is a list containing n variables, $I \leq n$ is a variable (index into the list) and E is a variable containing element of the list L at index I.

The element constraint is useful for imposing constraints on pairs of variables without having to know what variables need to be paired up exactly. Only other option would be to use a complicated disjunctive constraint which would again stop propagation.

In our model, we use this constraint for pairing up actions with pointing transition windows.

$$element\left(L, I\right) = E \tag{5.2}$$

5.2.3 Search in SICStus

SICStus prolog gives us backtracking for free as backtracking is an integral part in the Prolog language. SICStus also implements a system for maintaining consistency in the model and a simple search procedure that allows for a custom variable and value ordering heuristic.

However we can make a completely custom search procedure by defining a predicate that keeps adding extra constraints into the model. The CSP engine then tries to satisfy the constraints by restricting variable domains (maintaining consistency). If at any point a domain of any variable empties, prolog will automatically backtrack to a previous choice point. The final solution is found when each variable has a single value left in its domain.

6. MEX Constraint programming solution

In this chapter, we will describe the constraint programming solution for the Mars Express (MEX) problem in detail. We will at first describe our constraint model. In the second half of the chapter, we will describe the search heuristic used in the solution.

6.1 Constraint solution for the MEX problem

In this section, we will introduce our attempt at encoding the presented problem as a Constraint Satisfaction Problem (CSP) and solving it in SICStus Prolog.

6.1.1 Accepted simplifications

We have taken several simplifications for the CSP solution.

Single downlink We assume that there is a single downlink action for each Payload Operations Request (POR). This downlink action will download all data generated by the related POR. This simplification was taken for technical reasons of managing the inputs into the model.

No Uplinks We do not schedule uplinks at all as part of the CSP solution. This simplification was taken because the problem of scheduling uplinks is trivial as described in the MEX problem. The uplinks can be scheduled in a post-processing phase same as in the ad-hoc solution described in chapter 4.

No MTL We do not consider Mission Timeline (MTL) actions for moving Tele Commands (TCs) into the MTL. This was done for the same reasons given in chapter 4.

6.1.2 Overview

The model is based on encoding the MEX problem into a static form by pre-generating actions that are dynamic in the problem. We then model each action by its start, end and duration and define constraints restricting the schedule to fullfill the problem requirements. We use a set of cumulative resources to prevent actions from overlapping. We also use cumulative resources to model the data stores usage. The pointing states required by different actions are modeled by defining a pointing window after each Pointing Transition Action (PTA). Each pointing window has a specified pointing and every action needs to belong into one of the windows.

For the search, we use a variable ordering based heuristic to find the solution. The heuristic is based on scheduling the most constrained actions first and attempting to predict pointing requirements in the early phase of the scheduling to prevent too much backtracking later.

Input The input for the CSP solution is a Prolog source file which is described in attachment B. The source file is generated from the input files described in chapter 2.

6.1.3 Problem encoding

6.1.3.1 Mapping to scheduling

We have mentioned that CSP can only be used to solve static problems (with a known finite number of variables). There are two parts of the MEX problem that are not static:

- Maintenance actions
- PTAs

We work around this by pre-generating a fixed number of maintenance actions and PTAs. The formula to compute the number of maintenance actions is shown in 6.1. In the equation m is the number of maintenance actions, D is the total duration of the schedule in hours. We divide the duration by $28h$, which is the average between the imposed minimal and maximal gap between maintenance actions.

The formula for the desired number of PTAs is described in 6.2. In the formula n is the number of PTAs and P is the number of PORs in the input. The constant 2 was chosen based on experimentation. In theory, the needed number of PORs could be higher but in practice the formula generates more PTAs than required every time. The number is however still low enough not to slow the model down considerably.

$$m = \lfloor \frac{D}{28h} \rfloor \tag{6.1}$$

$$n = 2P \tag{6.2}$$

6.1.3.2 Model focus

The main focus of our model is to eliminate disjunctive constraints as much as possible. Disjunctive constraints essentially stop propagation of variable domains and the domains of adjoining variables are only pruned when a definite value is assigned. With the huge search space coming from the fact that many variables have to represent time, this would have a huge performance impact. There are several tricks that we use to get around disjunctive constraints:

- Global "cumulative" constraints
- Table constraints

- Element constraints for look-up in ordered lists of variables

This, combined with optimizing the number of variables as much as possible allows us to achieve good pruning capabilities of the model which in turn severely narrows down the search space.

6.1.3.3 Basic action model

Every action is characterized by three basic variales:

- Start
- End
- Duration

The notation that we will use everywhere in this chapter for action variables is described in table 6.1. The only variable that warrants more explanation is the pointing window. The pointing window variable is an index of the PTA that sets the required pointing for the given action. The index is then used to add constraints on the PTA in relation to the action. This is done using the element constraint introduced earlier.

Action type	Start	End	Duration	Pointing window
POR (experiment)	Es_i	Ee_i	$Edur_i$	Ew_i
Downlink	Ds_i	De_i	$Ddur_i$	Ew_i
PTA (switch)	Ss_j	Se_j	$Sdur_j$	$-$
Maintenance	Ms_k	Me_k	$Mdur_k$	Mw_k

Table 6.1: Notation for variables for action i. $i\epsilon\,[1, P]$ where P is the total number of PORs. Analogically $j\epsilon\,[1, n]$ and $k\epsilon\,[1, m]$ where n is the number of PTAs and m is the number of maintenance actions.

If a variable appears without the index, for example Es, it means a list of variables ordered by their index.

Basic action constraints The basic constraints 6.3, 6.4, 6.5, 6.6 tie the the action temporal variables together.

$$Ee_i = Es_i + Edur_i \tag{6.3}$$

$$De_i = Ds_i + Ddur_i \tag{6.4}$$

$$Se_j = Ss_j + Sdur_j \tag{6.5}$$

$$Me_k = Ms_k + Mdur_k \tag{6.6}$$

POR starts domain PORs need to be scheduled at a specific offset from a pericenter. This effectively limits the number of available times for each POR to the total number of pericenters in the schedule. We use this to pre-compute the domains of the Es variables. The domain for each Es_i is defined in 6.7 where N is the total number of pericenters in the schedule, p_j is the time of pericenter j and o_i is the required offset of POR i.

$$dom(Es_i) = \bigcup_{j=1}^{N} \{p_j + o_i\} \tag{6.7}$$

6.1.3.4 Communication scheduling

Ground station intervals We are not explicitly modeling Ground Station Availability (GSA) intervals. This is because all of constraints imposed by the intervals can be encoded without using any extra variables.

To limit a communication action to be performed only in communication windows, we simply reduce the domain for the action temporal variables as shown in 6.8. $Edata_i$ is the data amount produced by experiment i and is part of the input, $w_{bitrate}$ is the fixed bitrate of the GSA window w which is also defined in the input. The second constraint imposed by comms windows is the fact that a downlink duration is depending on the chosen comm window bitrate. We solve this by adding one table constraint for each downlink. The constraint is described in table 6.2. In the table w_s and w_e is the start time (resp. end time) of the GSA window w which is again defined in the input.

These two constraints are enough to tie the durations to corresponding possible start times.

$$dom(Ds_i) = \langle 0, End \rangle \cap \left(\bigcup_{w \in W} \left[w_s, w_e - \lceil \frac{Ddata_i}{w_{bitrate}} \rceil \right] \right) \tag{6.8}$$

$Ddur_i$	Ds_i
$\frac{Edata_i}{w_{bitrate}}$	$[w_s, w_e - Ddur_i]$
...	...
$\forall_{w \in W}$	...

Table 6.2: Table limiting downlink durations based on window bitrates. The table contains one row for every communication window w.

Downlinks There are two more constraints to downlinks: A downlink needs to be performed after it's experiment, and it needs to be performed before the data capacity on the payload store is exceeded. The first constraint is very simple and is expressed by 6.10. The second constraint is more interesting so it deserves a detailed explanation.

The data capacity can be thought of as a consumable resource for cumulative scheduling. The actions that consume this resource are called *data intervals*. Each data interval DI_i represents the time during which the data produced by

POR i is stored in the Solid State Mass Memory (SSMM). A data interval DI_i has the following properties:

- $start(DI_i) = Es_i$
- $end(DI_i) = De_i$
- $res(DI_i) = Edata_i$

Where $Edata_i$ is the amount of data produced by POR i. This is a constant for each i and is defined in the input.

This simply means the interval consumes as much resource as is the data production of it's related POR. It starts with the start of the POR and ends with the end of the downlink. This can be viewed as a pessimistic approach to data modeling as we are reserving capacity for all the data until all of it is downloaded. Since downlinks cannot run in parallel anyway, this doesn't restrict us in any way.

We then simply add the constraint described in 6.9 for every payload. The indexing set L_s contains indices of all PORs that use s as their data store, $capacity(s)$ is then the data capacity of data store s. Since $Edata_l$ is defined in the input, this is a very efficient way of representing the downlinks as it doesn't require additional variables. Instead it prunes the search space so that the solver won't try to assign downlinks in a way where capacity would be overreached.

$$cumulative\left(\{task(Es_l, De_l, Edata_l) | \forall l \epsilon L_s\}, capacity(s)\right) \tag{6.9}$$

$$Ds_i \gtrsim Ee_i \tag{6.10}$$

6.1.3.5 Maintenance actions

Maintenance actions are very simple to schedule as the only imposed constraint on them is that they need to be spaced 3 - 5 orbits apart. This is represented by the constraint 6.11. We also add constraint 6.12 which ensures that the first maintenance is scheduled reasonably soon. This allows the previous plan-cycle to schedule its last maintenance action somewhere into the last orbit.

$$Ms_{k+1} - Ms_k \gtrsim 21h \wedge Ms_{k+1} - Ms_k \lesssim 35h \tag{6.11}$$

$$Ms_k \lesssim 28h \tag{6.12}$$

The constants $21h$ and $35h$ are the minumum and maximum spacing of maintenance actions defined in the MEX problem. The $28h$ time was again chosen to provide a reasonable result where we assume that the last maintenance occured roughly one orbit before the schedule.

6.1.3.6 Pointing transitions

We have already specified how we generate a fixed number of PTAs to schedule. It must be noted that not all PTAs will be needed in the final schedule. During the scheduling, we therefore specify the extra PTAs as having zero duration and therefore not being included in the final schedule. This is a common technique for encoding a dynamic number of actions into the static CSP. We also keep the PTAs ordered to reduce symmetry in search. We will describe the details later in this section.

Extra variables Pointing actions have two extra variables tied to them. The variables are described in the table 6.3. The Td_j variable is simply the duration from the end of PTA j until the next PTA $(j+1)$.

Variable	Description
Sst_j	pointing state after switch
Td_j	duration of the pointing after switch

Table 6.3: Extra variables tied to PTAs. The index j is handled analogically to previously described pointing variables.

Basic PTA constraints We order the PTAs to reduce symmetry and to drastically simplify the constraints. This can be easily done as the variables representing the PTAs are completely interchangeable. The ordering is imposed by the constraint described in 6.13. Because the duration of a PTA can only be 30 minutes or zero, the constraint 6.14 enforces that all unused pointings are at the end of the schedule. The table shown in 6.4 then limits the duration of the pointing after a PTA. In this section, we'll refer to this time spanned by the pointing set by a PTA as a *pointing window*.

$$Ss_{j+1} = Se_j + Td_j \tag{6.13}$$

$$Sdur_{j+1} \leq Sdur_j \tag{6.14}$$

Sst_j	Td_j
Earth	$[0, End]$
NAD	$[0, 68]$
FIX	$[0, 90]$

Table 6.4: Table constraint for switch durations.

Action pointing constraints We use the element constraint to pair up the pointing windows with actual MEX actions. With this in mind, the constraint described in 6.15 is quite simple. Est_i is a constant describing what pointing the experiment needs. 6.16 then shows analogical constraint for downlinks which always require the Earth pointing. Analogical constraint 6.17 exists for maintenance.

$$element(Sst, Ew_i) = Est_i \tag{6.15}$$

$$element(Sst, Dw_i) = Earth \tag{6.16}$$

$$element(Sst, Mw_k) = Earth \tag{6.17}$$

We also enforce constraints on the duration of the pointing window by limiting it's start and end as described in 6.18, 6.19 and 6.20. All these constraints state that an action assigned to pointing window j must begin after the PTA j and end before PTA $j + 1$.

$$element(Ss, Ew_i) = Ss_j \wedge Ss_j \leq Es_i \wedge element(Ss, Ew_i{+}1) = Ss_{j+1} \wedge Ss_{j+1} \geq Ee_i \tag{6.18}$$

$$element(Ss, Dw_i) = Ss_j \wedge Ss_j \leq Ds_i \wedge element(Ss, Dw_i{+}1) = Ss_{j+1} \wedge Ss_{j+1} \geq De_i \tag{6.19}$$

$$element(Ss, Mw_k) = Ss_j \wedge Ss_j \leq Ms_k \wedge element(Ss, Mw_k{+}1) = Ss_{j+1} \wedge Ss_{j+1} \geq Me_k \tag{6.20}$$

6.1.3.7 Resource scheduling

To prevent the POR actions from overlapping on any single payload, we again use the cumulative constraint. Note that PTAs overlapping with PORs is already solved by constraint 6.18, analogically downlinks are taken care of by constraint 6.19. Potential overlap of PORs and downlinks is solved by the fact that experiments happen in a different pointing than downlinks.

For this reason, we add the constraint defined in 6.21. E_P represents all experiments on payload P, so we need to add this constraint for every possible payload.

$$cumulative\left(\{task(Es_i, Ee_i, 1) | \forall E_i \epsilon E_P\}, 1\right) \tag{6.21}$$

While this would be enough for satisfying constraints imposed by the problem, we also add the constraint 6.22 to prune the search space. The constraint simply represents that a maximum of 6 PORs can run in parallel (6 is the number of payloads) and no other actions can run in parallel with them or each other. E,D,S and M in the constraint represent all experiments, downlinks, PTAs and maintenance actions respectively.

$$\begin{aligned} cumulative(\quad & \{task(Es_i, Ee_i, 1) | \forall E_i \epsilon E\} \cup \\ & \{task(Ds_i, De_i, 6) | \forall D_i \epsilon D\} \cup \\ & \{task(Ss_j, Se_j, 6) | \forall S_j \epsilon S\} \cup \\ & \{task(Ms_k, Me_k, 6) | \forall M_i \epsilon M\} \quad , 6) \end{aligned} \tag{6.22}$$

6.1.4 Search

As we mentioned in the previous chapter, effectivity of search is largely dependent on choosing the right heuristic. Since the backtracking as well as maintaining arc consistency is already implemented in SICStus Prolog, we focus on describing how we utilized those techniques with our heuristic.

6.1.4.1 Algorithm

Heuristic The used search algorithm is based on directly assigning values to variables. Therefore we used both a variable ordering heuristic (by ordering actions) and value ordering heuristic (by defining our own choice points).

Our variable ordering heuristic is based on the "most constrained first" principle. This is very effective in minimizing the number of necessary backtracks before the first solution is found. Since we are only interested in the first solution, this is perfect for us.

However it turned out that scheduling downlinks before maintenance leads to a lot of backtracking when the maintenance doesn't fit. Therefore the maintenance actions are scheduled before downlinks.

The actions are scheduled in this order:

1. PORs
2. PTAs
3. Maintenance
4. Downlinks

This is natural as PORs only have a few values to choose from (because of the fixed offsets), PTAs are constrained by the PORs requiring correct pointings and downlinks can just fit anywhere in the gaps left after the PORs are in place.

Search As we have mentioned in the previous chapter, SICStus Prolog handles backtracking natively when searching for a CSP solution. Therefore we describe the search algorithm in an abstract form specifying in which order the different choice points are encountered. It is assumed that the solver can backtrack to a previous choice point and choose a different value if the value assignment does not satisfy any of the constraints.

We use four types of choice points in our search:

- *min* Assigns all possible values from the variable domain in *increasing* order.
- *max* Assigns all possible values from the variable domain in *decreasing* order.
- *or* Makes a choice between the left or right constraint.
- *minInterval* Assigns the min values from all possible value intervals in the variable domain in *increasing* order.

The *minInterval* choice point warrants more explanation. Every variable domain is a union of disjunctive intervals. In special cases, these intervals can have just one element or there can even be just a single interval. The *minInterval* choice point will simply assign a single value from each interval making up the domain. In our implementation, the single value is always the minimal value contained in the interval.

We will demonstrate on an example. Assume that a variable V has a domain defined by 6.23. The *minInterval* choice point will assign the values 50, 90 and 105 into variable V (in this order).

$$dom(V) = [50, 70] \cup [90, 95] \cup [105, 106] \tag{6.23}$$

The Prolog engine automatically handles remembering which values were already tried and handles backtracking to previous choice points correctly. This makes the entire search algorithm extremely short and elegant as can be seen below.

Algorithm 8 CSP Search

for $E_i \epsilon E$ **do**
  $Es_i = min(Es_i)$
  $Ew_i = min(Ew_i)$
  $j = Ew_i + 1$
  $Sst_j = Earth$ **or** $Sst_j \neq Earth$
end for
for $S_j \epsilon S$ **do**
  $Sdur_j = 0$ **or**
  $Sdur_j \gtrsim 0$ **and**
  ($Sst_j = Earth$ **and** $Ss_j = min(Ss_j)$ **or** $Sst_j \neq Earth$ **and** $Ss_j = max(Ss_i)$)
end for
for $M_k \epsilon M$ **do**
  $Mw_k = min(Mw_k)$
  $Ms_k = min(Ms_k)$
end for
for $D_i \epsilon D$ **do**
  $Dw_i = min(Dw_i)$
  $Ds_i = minInterval(Ds_i)$
end for

6.1.4.2 Search explanation

In this section, we'll explain why the selected value ordering heuristic works and what are the benefits and pitfalls of our search algorithm.

Experiment scheduling The experiments are clearly the most constrained action type. This is because each only has a very limited number of specific times when it can be performed. More specifically, each experiment can be performed

only at the specific offset but from any pericenter. Since pericenters are all identical, we are just exploring the space of all orderings between the experiments. There is very little other heuristic that we could use, we simply have to try all possible orderings anyway.

The line 5 in the algorithm 6.1.4.1 chooses the next pointing after the POR. Note that the heuristic will always choose Earth at first. This is based on real data observation, where experiments are usually performed in a single group of experiments that require the same pointing and then the group is followed by a set of downlinks which download the produced data.

$$Sst_j = EARTH \wedge Ss_j = min(Ss_j) \vee Sst_j \neq EARTH \wedge Ss_j = max(Ss_j) \quad (6.24)$$

Downlink scheduling We use the *minInterval* choice point for scheduling the dowlink actions because it does not matter that we won't try all values in each interval. For the downlink action each interval represents an unocuppied portion of a GSA window. It is trivial to see that if we schedule several downlinks into the same interval by using the *minInterval* choice point, they will all be tightly packed as the choice point always assigns the minimum of the interval as the start.

Now we will show that the ordering in which downlinks are scheduled into one interval does not matter. It is trivial, the effect of each downlink is removing a certain amount of data from a packet store. This can be represented as a vector with the amount of data removed from each packet store. For a single downlink, the vector will contain zeroes and one non-zero number. The effect of several consecutive downlinks is then simply a sum of their individual effects. Since the '+' operation is cumutative on integer vectors, the effect is always the same, no matter what the order of downlinks.

It is now trivial to see that anything else than a tightly packed schedule is sub-optimal. If there is a gap between downlinks A and B that allows for a downlink, we could have scheduled B earlier to cover the gap. Since we schedule downlinks last, the gap could potentially be only used for other downlinks, but if a downlink C would fit into the gap, it can also fit right after B if we move B to span the whole gap. This is illustrated in figure 6.1.

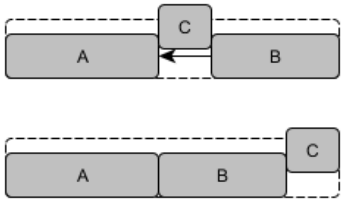


Figure 6.1: Illustration of tight downlink scheduling.

PTA scheduling Scheduling of the PTAs has two parts. At first we need to determine if the PTA will even be present in the final plan. This is however

already defined from the experiment schedule as that will define which pointing windows will need to have a non-zero duration Td_j. The other part to PTA scheduling is setting the PTAs down to specific times. The output from the first phase will usually just be intervals in which the PTAs have to start and end to satisfy the experiment constraints. We use the formula shown in 6.24 to set the times. In words, we do the following:

- If the PTA is switching to EARTH, schedule it as early as possible.
- If the PTA is switching to any other pointing, schedule it as late as possible.

The purpose of this is to make the pointing windows around experiments as tight as possible. The reasoning for that is easy: only experiments and maintenance can be performed during the NAD and FIX pointings. Since we already have precise times for all the experiments, we know that we will not need any extra pontings for any more of them. As for maintenance actions, we will show that it is better to schedule them into earth pointing windows.

The reasoning is simple to prove. Assume that scheduling to a different pointing is better (frees up more space in the plan). The only pointing that is possible is NAD as it can just barely fit the maintenance action. We can't do experiments in that window because they can't run parallel to maintenance though, so we've switched to NAD just because of the maintenance action. If we switched from FIX, we might've as well switched to EARTH and the result would be the same, with the exception that we can fit other actions right after the maintenance without switching back. If we switched from EARTH, we might have as well stayed in EARTH pointing and saved the 30mins of time to switch. Therefore there's clearly no reason to do maintenance in other than EARTH pointing.

7. Experimental Evaluation

In this chapter we'll look at how we tested the solutions and discuss the results in the various cases.

While we did actually obtain real data from the ESA, we are not allowed to distribute them. This forced us to make data generator that tries to mimic the nature of the real data. Our tests were however always aimed at testing a specific part of the problem.

All data was randomly generated by our own generator. All data was also always generated solveable. The first sets of data were aimed at testintg the solver's capability to deal with a lot of PORs and were simplifying the problem by having a single GSA window spanning the whole length of the schedule. We then generated harder data sets that were including all the constraints of the problem.

7.1 Test types

7.1.1 Test-case generation

Generator The generator always creates solveable problems. The idea of the algorithm for generating the input consisting of N Payload Operations Requests (PORs) is:

1. Generate groups of PORs $\{P_1, P_2, ...P_n\}$ where n is the number of pericenters in the schedule and $\sum_{i=1}^{n} |P_i| = N$
2. Assign total amount of data D_i produced by each POR group P_i such that all the data can be downloaded before P_{i+1}
3. The previous step also handles keeping every 4th data size smaller to allow fitting of the maintenance action. The reduction in data size obviously depends on the bitrate of the available Ground Station Availability (GSA) windows as we need to reserve at least an hour of time for the maintenance.
4. Randomly divide the total data size D_i between the PORs in P_i.

Simple data The first set of tests was aimed at scheduling as many experiments as possible with their downlinks and while maintaining all constraints related to the pointing transitions and maintenance actions. The data thus had the following characteristic:

- A random number of experiments per pericenter that fit into one NAD or FIX window.
- Downlinks that take roughly 75 per cent of time between experiment groups.
- A single GSA event spanning the entire length of the timeline.

- The data amount produced by the experiments was significantly lower than the data cap of each packet store.
- Number of experiments assumed to be divided evenly on average between pericenters.

Data fitting The second set of tests was aimed at scheduling of downlinks with limited GSA windows. Tha data was very similar to the first data sets with the following changes:

- GSA windows only cover the time around apocenters. The average GSA time is 4h.
- Experiment data is generated accordingly to use up to 75 per cent of all available GSA time for downlinks.

Data capacity The data capacity tests are the same as data fitting tests except for the fact that the data amount produced by a single experiment is roughly half of the data capacity of its Solid State Mass Memory (SSMM) store. In other words, no more than 2 experiments on the same POR can be performed without downloading data from at least one.

7.2 Test results

7.2.1 Solution comparison

For each input, we have compared the solution assumed by the generator with the solutions presented by the two planners (if they were able to compute it). Since no metric to measure the quality of solutions was presented with the problem, we have focused on the following attributes:

- Ability to compute the solution.
- Total length of the schedule (excluding maintenance actions).
- Amount of unused space in the schedule.

All testing was done for a 5day or 7day period schedule with different amounts of experiments.

Computation times We do not include computation times in the tables because they did show anything interesting in our tests. The solutions were always found in a few seconds on a common Intel core i3 computer or the scheduler failed to stop in reasonable time. In all our tests, the time to successfully find a solution was never longer than a 30s and there was no visible correlation between the number of experiments and the computation time.

Our initial experiments have shown that if the scheduler does produce a solution in the 30s, it will not stop even when running over night. Therefore the method we used was to keep the scheduler running for 30s and if it didn't produce the solution by then, we would consider it unable to find a solution.

Presented data We ran each presented test four times with different data (generated with the same parameters). The results from every run were very similar, which is why each row in the presented tables is describing just one of the test results.

7.2.1.1 Simple data

We ran a comparison test first on the simple test-case data.

Scheduler	PORs	Schedule window	Result length	Unused time
Generator	15	5d	104h	40h
Ad-hoc	15	5d	103h	40h
CSP	15	5d	59h	5h
Generator	30	5d	117h	40h
Ad-hoc	30	5d	119h	29h
CSP	30	5d	91h	13h
Generator	50	7d	117h	30h
Ad-hoc	50	7d	120h	24h
CSP	50	7d	59h	5h
Generator	70	7d	167h	22h
Ad-hoc	70	7d	166h	23h
CSP	70	-	-	-
Generator	100	7d	166h	29h
Ad-hoc	100	7d	167h	25h
CSP	100	-	-	-

Table 7.1: Table comparing results of different schedulers on simple test-cases.

Several interesting observations came up in the data shown in the table 7.1:

- O1 The CSP scheduler produces by far the most effective schedules.
- O2 The Ad-Hoc solution is roughly on par with the generator as to the quality of the schedule.
- O3 The Ad-Hoc solution is much better when dealing with high number of PORs.

O1 was to be expected. The reason for the effective schedules produced by the Constraint Satisfaction Problem (CSP) solution is that the CSP is based on global search (backtracking). It is also the result of scheduling the actions by type (PORs first, then maintenance etc...).

O2 is actually just saying that the Ad-Hoc solution is not optimizing the schedule very much compared to the generator. The generator does not compact the schedule at all - it will always use all of the time available to it. There is no compacting built into the Ad-Hoc scheduler either. As long as it can add new actions, it will not attempt to reshuffle existing actions to get a more compact result. This behavior is making the Ad-Hoc use all of the time available to it most of the time, same as the generated solution.

O3 is the result of the local search in the Ad-Hoc solution. The local search techniques prevent most of the backtracking that would otherwise be necessary. The local search therefore delays the bad effects of the exponential complexity of the scheduling algorithm which only manifest when backtracking. The CSP solution on the other hand does not move actions after it has scheduled them. This means that any change to the existing schedule has to be done via backtracking. Therefore if the CSP heuristic fails in the beginning, the solver has to backtrack a lot which results in unfeasible computation times.

7.2.1.2 Data Fitting

We ran the data fitting test with the same parameter as the previous test. The results are shown in the table 7.2

Scheduler	PORs	Schedule window	Result length	Unused time
Generator	15	5d	120h	40h
Ad-hoc	15	5d	102h	28h
CSP	15	5d	114h	20h
Generator	30	5d	121h	31h
Ad-hoc	30	5d	113h	38h
CSP	30	5d	103h	15h
Generator	50	7d	169h	45h
Ad-hoc	50	7d	167h	38h
CSP	50	7d	149h	16h
Generator	70	7d	169h	25h
Ad-hoc	70	7d	168h	32h
CSP	70	7d	132h	11h
Generator	100	7d	172h	27h
Ad-hoc	100	7d	167h	25h
CSP	100	7d	132h	17h

Table 7.2: Table comparing results of different schedulers on data fitting test-cases.

Again, we can make several observations by looking at the results:

- OD1 The CSP scheduler still produces more effective schedules.
- OD2 The length of the plans differs by less between different solutions than in the previous tests.
- OD3 The Ad-Hoc solver produces shorter plans than generator but with more free space in most cases.

The most surprising observation is however that the CSP solution suddenly manages to schedule up to 100 PORs. We'll look at that in the next paragraph after we explain the other observations.

The reasons for OD1 are exactly the same as in the previous test-case.

OD2 is caused by the fact that the downlinks can only happen in the predefined times. This leaves less room for optimization by inserting downlinks into

whatever free space is available. This inevitably leads to all the schedules being closer in length to the generated solution.

OD3 is caused by the fact that the planner is able to overlap experiments effectively (run more in parallel) which the generator is not doing when it's generating the schedule. This results in more free space in the plan but because the downlinks can only happen in pre-defined intervals as mentioned above, it doesn't save much on the total length of the plan. The result is more free space in the plan.

CSP effectiveness The CSP solution was much better at handling more PORs here than in the previous test-case. Why is that?

First of all, it must be said that in this case, the CSP search worked only for about $\frac{3}{4}$ of inputs. The search failed to stop in reasonable time for the remaining inputs, just as it did in the previous tests.

As we already mentioned several times, the CSP search is very dependent on the heuristic. If it makes a bad choice in the beginning, then the search will most likely not stop in reasonable time. The extra constraints for the problem which limit the times downlinks can be placed into prune the search space. This is most likely helping the heuristic to make more correct decisions early on. The heuristic may still fail however, which is why the search doesn't finish in time $\frac{1}{4}$ of cases.

7.2.1.3 Data capacity test

We ran a last set of tests where the data productions by the experiments were close to the packet store capacities.

Scheduler	PORs	Schedule window	Result length	Unused time
Generator	20	5d	118h	53h
Ad-hoc	20	5d	117h	30h
CSP	20	5d	-	-
Generator	30	5d	120h	54h
Ad-hoc	30	5d	117h	32h
CSP	30	5d	-	-
Generator	50	7d	171h	60h
Ad-hoc	50	7d	164h	40h
CSP	50	7d	-	-

Table 7.3: Table comparing results of different schedulers on data capacity test-cases.

As we can see from the results in table 7.3, this test has proven to be the weak point of the CSP model. Since the model schedules the PORs first and downlinks only after all PORs are scheduled, it will schedule more PORs than the data capacity allows for. This leads to a lot of backtracking as the solver only fails after starting to schedule the downlinks. In our tests, the backtracking never finished in reasonable amount of time and in several cases the solver simply ran out of memory even for 20 PORs.

This is clearly a weak point of the model itself, where we anticipated the global cumulative constraint to be able to catch this inconsistency early on and not allow scheduling PORs without leaving room for the downlink in between them. The possible solution would be to try and catch this manually by keeping track of the data scheduled in a certain POR group (experiments around a single pericenter) and not allow this data to exceed the packet store capacity.

The ad-hoc solver did not have any problem handling up to 50 PORs and we didn't run any larger test with this type of data. As we can see, the solutions are again comparable in length to the generator provided solutions but contain much less unused time. This is caused by extra pointings in the ad-hoc created schedules. The generator solutions use only one Pointing Transition Action (PTA) per pericenter and one PTA per downlink group. The ad-hoc solution on the other hand contains a lot of PTAs around pericenters. These PTAs handle switching between the different experiment pointings or are handling the maximum pointing duration.

7.2.2 Solution attributes

There were also some interesting facts about the CSP solutions:

1. The solver schedules most experiments for FIX pointing first and the experiments for NAD after.

2. The pericenters closer to the beginning of the plan tend to be much more full.

These all come from the used heuristic which attempts to schedule everything as early as possible (2) and prefers to switch to the EARTH pointing instead of the other Mars pointing (1).

CSP output details A visualisation of a part of one of the schedules for 70 PORs can be seen in Figure 7.1. The figure illustrates a part of the schedule spanning two pericenters.

We can clearly see a number of PORs clustered into three groups, where two groups belong a single pericenter and the third group is clustered around the other. As we mentioned earlier, the solver tends to cluster experiments with the same pointing requirements into the same part of the schedule. This is clearly visible here.

We can also see that the solver is capable of dealing with maximum pointing length of the FIX pointing by inserting an EARTH pointing betewen them (seen near the first pericenter). Remember that we had to specifically handle this case in the ad-hoc solution presented in Chapter 4.

The figure also shows how the downlinks are restricted by the GSA windows.

Ad-hoc output details The ad-hoc solution is in principle very similar to the CSP solution. We have chosen an example from the last set of tests (data capacity test). The example is from a schedule containing 50 PORs.

The main focus of the visualisation shown in figure 7.2 are the data stores at the bottom. We can see that the AS packet store (green line) is almost full around the first apocenter, then gets emptied during the downlink period and is filled again by the next two ASPERA PORs.

We can also clearly see an Earth PTA inserted into a FIX pointing window next to the second pericenter. This is the pointing fix mentioned in chapter 4.

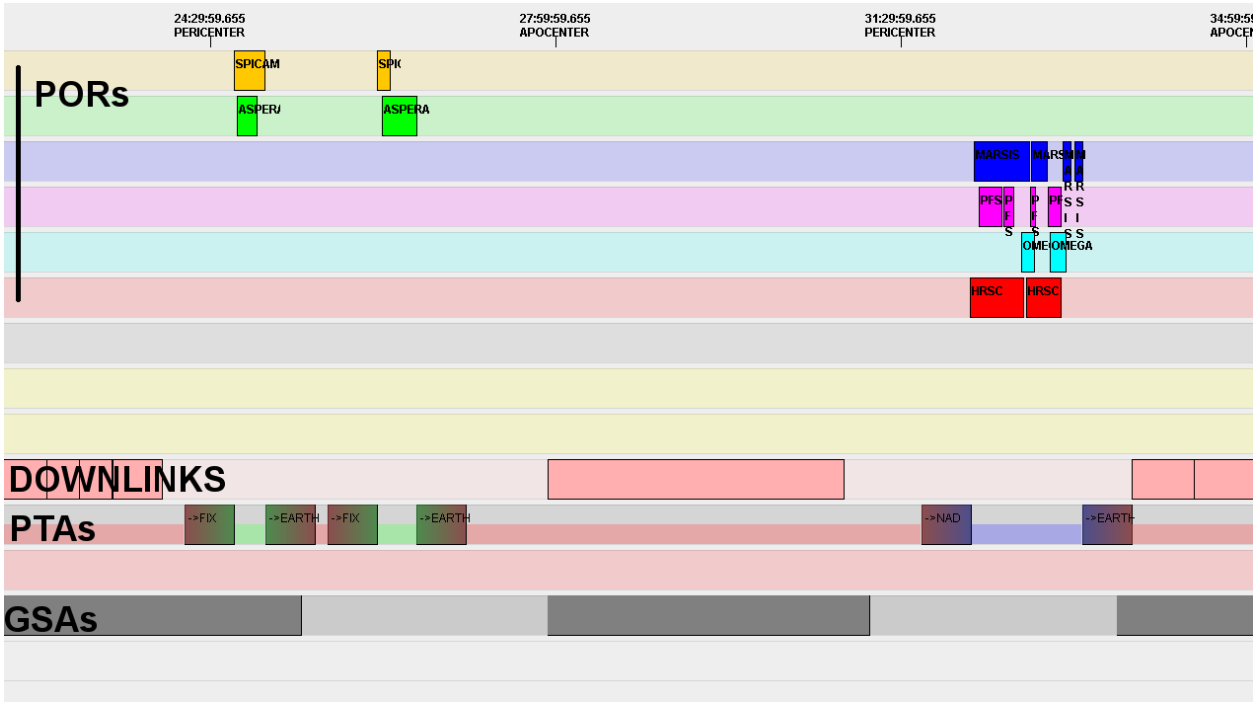


Figure 7.1: Part of the CSP solver schedule with 70 PORs and GSA windows.

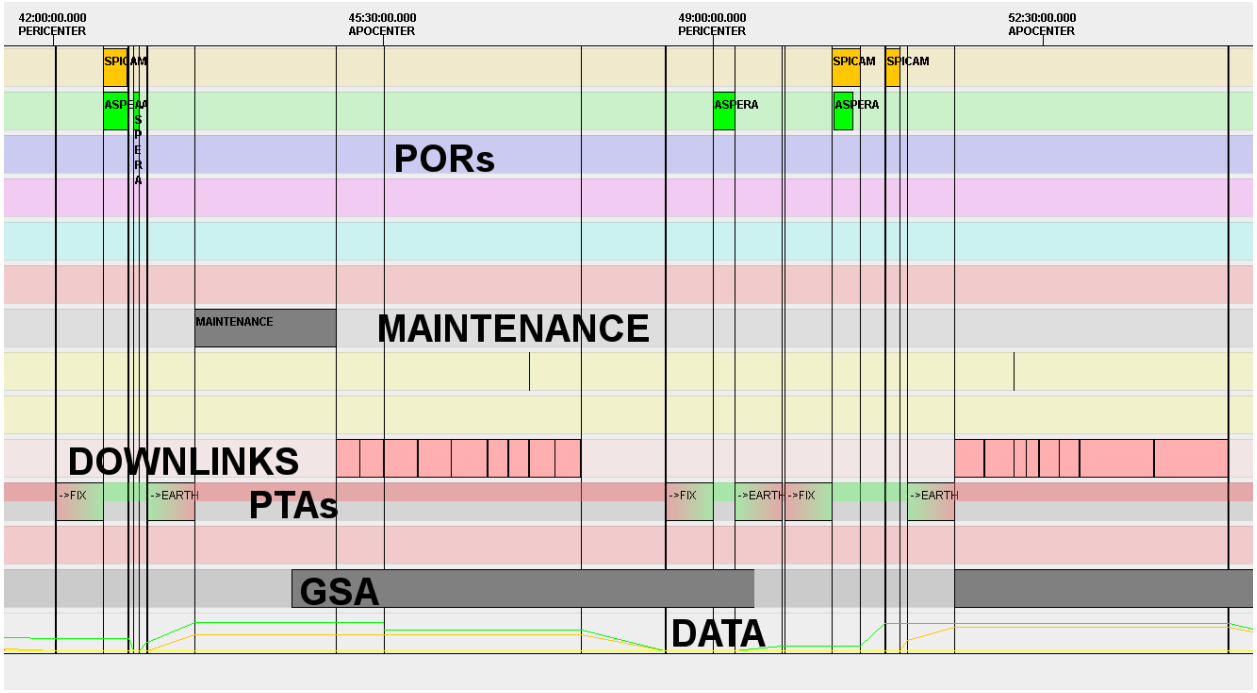


Figure 7.2: Part of the ad-hoc solver schedule from the data capacity test.

8. Conclusion

As part of this thesis, we have implemented and tested two completely different approaches to the presented MEX problem. The first one was a totally ad-hoc custom scheduling solution while the other one was using a more common model of encoding the problem into CSP. The nature of the problem which was not completely a planning problem neither a classical scheduling problem was however preventing us from using any estabilished solution method. In the end, both solutions were able to solve at least a subset of the problems inputs.

8.1 Solutions

When we analyze the results, the CSP solution is clearly able to produce higher quality solutions, the ad-hoc solution on the other hand is able to handle more experiments more easily. These are by obviously the most notable advantages of both solutions. We have also proven that both solutions are capable of computing schedules containing up to 50 PORs without any difficulties. The ad-hoc solver and in some cases the CSP solver as well are capable of computing schedules containing even up to 100 PORs.

8.1.1 Ad-hoc solution

The ad-hoc solution was proven sufficient to handle the problem. The main strength of the ad-hoc solution were the local search techniques which eliminated a lot backtracking.

The problems of the ad-hoc solution included the inability to globally compact the schedule leading to a lot of empty intervals in the result. The ad-hoc solution is also more difficult to extend and maintain when compared to the declarative CSP solution.

8.1.2 CSP model

The biggest strength of the CSP solver is its ability to produce tightly packed optimized schedules. It is also able to cope with the problem of the dynamic PTAs and even manages to handle the intermediate PTAs to cope with maximum pointing duration.

The most notable weakness of the current CSP model and heuristic is its inability to cope with experiments producing chunks of data that are close in size to maximum packet store capability. This will usually lead to the solver having to search the whole exponential space of different POR permutations which is clearly unfeasible. Since we thought that this problem would be caught by the solver on its own, we didn't account for it in the model itself at all. This made it unfeasible to address when it was detected.

Another already mentioned problem of the CSP solver is that it is very dependent on the used heuristic. Either the heuristic catches on very quickly and the solution is produced in a few seconds or the heuristic does not work and

the program will not stop in reasonable time. The problem is clearly too large to allow for searching of the entire search space and therefore aggressive and domain-specific pruning has to be used to get a solution in feasible time. An obvious extension would therefore be to have several heuristics that are tried on the input in order (with each having a time cap) to obtain a solution quickly.

Developing a CSP-based solution can prove challenging as it is difficult to analyze the run of the algorithm within the CSP solver. It is thus very difficult to optimize the search or the model based on what is actually happening in the solver itself. Most optimizations in the model can be thought out in advance because of the well-defined nature of the CSP constraints and pruning, but there may still be hidden bottlenecks in the model which can considerably slow the solver down.

The presented CSP model is actually a second iteration of the CSP solution after the first iteration proved fruitless even though it looked well on paper. The first CSP model was much more naive in the approach as it was an attempt to almost literally encode the real-world constraints in CSP. The main bottleneck proved to be assigning downlinks to specific GSA windows in the same way we assign actions to pointing windows in the current model. While nothing seemed to be wrong with such an approach at the first glance, the result was that the old model could barely schedule 10 actions, while the current one can deal with a 100.

8.2 Potential extensions

It must be noted that both our solutions count on the problem always being solveable. There is no technique in place for providing incomplete solutions, such as scheduling only a subset of the experiments. This is an obvious extension to our solutions. The main reason why we didn't implement such an extension is that we didn't have any fitness function that would rate the solutions or any function that would assign priorities to different experiments.

An already mentioned extension to our work is to come up with several different heuristics, each aimed at a different area of the problem and then having the solver try each or a combination of them to find the solution quickly for a given data set. The first candidate for such a heuristic would be one that places PORs while accounting for maximum data capacity. It should even be possible to analyze the input data first and select the most likely to succeed heuristic based on this analysis.

Another set of possible extensions would be implementing the optional constraints described in the original challenge problem. In the end our CSP solution only focused on the mandatory constraints because the model was getting complicated even thus. We were also not sure if the CSP approach would even prove viable for this problem, which is why we wanted to keep things as simple as possible.

The last possible follow-on work would be building a decent GUI on top of the CSP solver as for the purpose of the thesis we ended up with a proof-of-concept solution that requires a lot of manual interaction between our original Java based application for visualisation and the SICStus Prolog based solver for the actual computation. A GUI similar to what was presented in the MEXAR2 paper would be an obvious next step were we to attempt to deploy this solution in practice.

8.3 Final word

It was shown to a certain extent that CSP which is usually used to solve well defined static issues can be used even for solving a complex scheduling problem with a limited dynamic component to it. This of course needs to be done by encoding the dynamic component in a static manner but we have proved that this is viable for the planning/scheduling purpose.

We have also shown that the CSP approach provides high quality results for most inputs even without a complicated domain-specific heuristic. In this respect it was clearly shown that a CSP based solution can be as good if not better than an ad-hoc solution. The weakness of CSP becomes apparent when the solution is difficult in an area of the problem not handled by the used heuristic (such as one of the constraints can only be satisfied with a small solution subset). Then the CSP solution becomes only a little better than common backtracking.

The CSP solution is however relatively easy to extend to include multiple heuristics to remedy this problem. Therefore the CSP solution comes out as superior to the ad-hoc solver based on backtracking.

Bibliography

[1] Agency, E. S.: Mars Express Summaru. 2014, [Online; accessed 12-January-2014].
URL http://sci.esa.int/mars-express/31021-summary/

[2] Barták, R.: CSP Lectures - Arc Consistency. 2014, [Online; accessed 12-January-2014].
URL http://ktiml.mff.cuni.cz/~bartak/podminky/lectures/CSP-lecture04eng.pdf

[3] Barták, R.: CSP Lectures - Introduction. 2014, [Online; accessed 12-January-2014].
URL http://kti.mff.cuni.cz/~bartak/podminky/lectures/CSP-lecture01eng.pdf

[4] Barták, R.: CSP Tutorial - Binarization of Constraints. 2014, [Online; accessed 17-June-2014].
URL http://ktiml.mff.cuni.cz/~bartak/constraints/binary.html

[5] Cesta, A.; Cortellessa, G.; Denis, M.; aj.: AI Solves Mission Planner Problems. *IEEE Intelligent Systems*, ročník 22, 2007.

[6] Dvořák, F.; Barták, R.: Integrating Time and Resources into Planning. In *Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence*, IEEE Computer Society, 2010, s. 71–78.

[7] Fratini, S.; Policella, N.: ICKEPS 2012 Challenge Domain: Planning Operations on the Mars Express Mission. 5 2012, available at http://icaps12.poli.usp.br/icaps12/sites/default/files/ickeps/mexdomain/MEX_KEPS_domain_v12.pdf.
URL http://icaps12.poli.usp.br/icaps12/sites/default/files/ickeps/mexdomain/MEX\textunderscoreKEPS\textunderscoredomain\textunderscorev12.pdf

[8] Kolombo, M.; Pecka, M.; Barták, R.: An Ad-hoc Planner for the Mars Express Mission, 2013.

[9] Nau, D.; Ghallab, M.; Traverso, P.: *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004, ISBN 1558608567.

[10] Rabenau, E.; Donati, A.; Denis, M.; aj.: The RAXEM Tool on Mars Express - Uplink Planning Optimisation and Scheduling Using AI Constraint Resolution. In *SpaceOps-08. Proceedings of the 10th International Conference on Space Operations*, 5 2008.

[11] SICSTUS: SICSTUS Prolog. 2014, [Online; accessed 17-June-2014].
URL http://sicstus.sics.se/index.html

List of Tables

List of Abbreviations

ESA European Space Agency

MEX Mars Express

PTA Pointing Transition Action

POR Payload Operations Request

MDAF Master timeline Detailed Agenda File

GSA Ground Station Availability

TC Tele Command

MTL Mission Timeline

SSMM Solid State Mass Memory

DL Downlink

UL Uplink

FIX Fixed

NAD Inertial

CSP Constraint Satisfaction Problem

CP Constraint Programmings

AC Arc Consistency

DFS Depth First Search

S/C Space Craft

Attachments

A. File formats

A.1 POR File

The POR file consists of individual POR entries. Each entry is composed of a header which specifies the instrument and then a set of rows describing data productions.

Each data production is composed of a packet store, an offset from pericenter (in ms) and by the amount of data produced (in bits). The data generator described in the thesis always generates PORs with exactly two data productions where one marks the beginning of the POR and the other marks the end.

```
OMEGA
OM 2520000 1260000
OM 3180000 1260000

ASPERA
AS 2700000 900000
AS 6000000 900000

SPICAM
SI 6060000 1620000
SI 7260000 1620000

SPICAM
SI 7320000 2100000
SI 8220000 2100000

MARSIS
MI -900000 4260000
MI 420000 4260000

PFS
PS 480000 1440000
PS 1440000 1440000

OMEGA
OM 1500000 1860000
OM 1860000 1860000
```

A.2 EVTM File

The EVTM file is a csv file where each line defines a single orbit event. The CSV has five columns:

- ID of the event (unused)
- Date and time of the event

- Event type: can only be `PERICENTER_PASSAGE` or `APOCENTER_PASSAGE`
- Passage height of the orbiter above Mars (unused)
- Martian day/night cycle stage (unused)

```
1,2004-03-01 03:30:00.103,PERICENTER_PASSAGE,10000,DAY
1,2004-03-01 07:00:00.103,APOCENTER_PASSAGE,10000,NIGHT
1,2004-03-01 10:30:00.103,PERICENTER_PASSAGE,10000,DAY
1,2004-03-01 14:00:00.103,APOCENTER_PASSAGE,10000,NIGHT
1,2004-03-01 17:30:00.103,PERICENTER_PASSAGE,10000,DAY
1,2004-03-01 21:00:00.103,APOCENTER_PASSAGE,10000,NIGHT
1,2004-03-02 00:30:00.103,PERICENTER_PASSAGE,10000,DAY
1,2004-03-02 04:00:00.103,APOCENTER_PASSAGE,10000,NIGHT
```

A.3 GSA File

The GSA file is a CSV file where each line defines a single GSA window. The CSV has four columns:

- ID of the ground station (unused)
- Date and time which marks the beginning of the event
- Duration of the event in ms
- Bitrate in bits/ms

```
GSA-0,2004-03-01 05:45:00.103,13740000,1
GSA-1,2004-03-01 11:54:00.103,16440000,1
GSA-2,2004-03-01 19:04:00.103,18900000,1
GSA-3,2004-03-02 02:49:00.103,16140000,1
GSA-4,2004-03-02 09:48:00.103,14100000,1
GSA-5,2004-03-02 15:26:00.103,15240000,1
GSA-6,2004-03-02 22:51:00.103,21240000,1
GSA-7,2004-03-03 07:15:00.103,12600000,1
GSA-8,2004-03-03 12:35:00.103,23340000,1
GSA-9,2004-03-03 19:12:00.103,9240000,1
```

B. Prolog input

The SICStus solver uses Prolog source files generated by processing the GSA,EVTM and POR files. The input has several parts:

- the `end` predicate specifying length of the schedule in minutes
- the `switches_max` predicate specifying the maximum possible number of PTAs
- the `maintenance_max` predicate specifying the number of generated maintenance actions
- a number of `in_exp` predicates which define the duration (in minutes), data production, list of possible start times, required pointing and payload for each experiment
- a number of `in_dwin` predicates where each defines a single GSA event with start, end and bitrate.

1 unit of data production in the SICStus input is 60000 bits. The bitrate units are $\frac{bits}{ms}$.

```
end(2400).
switches_max(12).
maintenance_max(2).
%define experiments
%0 - AS, 1 - HR, 2 - MI, 3 - OM, 4 - PFS, 5 - SI
%0 - earth, 1 - NAD, 2 - FIX
%in_exp(Dur,Data,Starts,Pointing,Payload)
in_exp(17,154,[287,707,1127,1547,1967],1,2).
in_exp(50,174,[279,699,1119,1539,1959],1,2).
in_exp(51,50,[305,725,1145,1565,1985],1,1).
in_exp(50,296,[203,623,1043,1463,1883],1,1).
in_exp(50,300,[187,607,1027,1447,1867],2,5).
in_exp(72,236,[226,646,1066,1486,1906],2,0).
in_exp(68,226,[287,707,1127,1547,1967],1,3).
%list of downlink windows
%in_dwin(S,E,Rate)
in_dwin(385,662,1).
in_dwin(711,861,1).
in_dwin(1232,1498,1).
in_dwin(1648,1876,1).
in_dwin(2008,2325,1).
```