An Ad-hoc Planner for the Mars Express Mission

Martin Kolombo, Martin Pecka, Roman Barták
Charles University, Faculty of Mathematics and Physics
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
{kolombomartin,peci.jh}@gmail.com, bartak@ktiml.mff.cuni.cz

Abstract

Complete planning and scheduling of all spacecraft operations is a challenging area with the remote agent experiment at Deep Space 1 being a pioneering system. Still the complete approach is rare in practice. For example, in the Mars Express (MEX) mission, planning and scheduling techniques are used to solve some subproblems namely scheduling command upload and data download. In this paper we describe an approach to generate a complete schedule of the spacecraft that includes planning and scheduling of science, command uplink, data downlink, maintenance, and pointing operations. The proposed solving approach was designed to plan operations on the Mars Express (MEX) mission and it was motivated by the MEX challenge at the Fourth International Competition on Knowledge Engineering for Planning and Scheduling. The method is based on incremental addition of operations to a partial schedule and modifying the time allocation of already scheduled operations to fit the newly added operation. Despite its simplicity, the method seems to perform very well on experimental data, though a deeper evaluation with real data is still necessary. The paper briefly describes the problem solved, the integrated planning and scheduling algorithm, and the initial experimental results.

Introduction

The Mars Express Mission (MEX) is a successful mission of the European Space Agency with the spacecraft orbiting around Mars and producing 2-3 Gbit of scientific data per day. It is also one of the successful examples of application of planning and scheduling techniques in space. MEXAR 2 (Cesta et al. 2007) and RAXEM (Rabenau et al. 2008) are two tools operational at ESA-ESOC. MEXAR 2 was develop to schedule Data Dumping activities while RAXEM schedules Data Uplink activities. However, these tasks (downlink and uplink) are just two types of tasks necessary to operate the spacecraft. Naturally, the core role of MEX consists of scientific (observation) tasks that are complemented by the command uplink and data downlink activities. There are also maintenance activities necessary to keep the spacecraft in a good condition. Currently the complete planning process is realized through a collaborative problem solving process between the science team and the mission planning team. Two teams of human planners iteratively refine a plan of all activities of the mission. In 2012 this complex planning problem has been proposed as a challenge for the Fourth International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS 2012). The goal was to develop a system that takes the description of scientific operations together with operational constraints, ground station visibility, a spacecraft trajectory etc. as its input and generates a complete schedule of all the uplink, science, downlink, maintenance, and auxiliary operations. Notice in particular that the flight dynamics is not part of the solution as the spacecraft trajectory is given as the input. The schedule must respect all operational constraints and maximize the scientific outcome.

In this paper we describe a solving approach that we developed for the ICKEPS challenge. We see the problem mainly as a scheduling problem because we can generate (plan) the main activities to be scheduled (uplink, science, downlink, maintenance) in advance and only some auxiliary activities (pointing) need to be inserted to make the schedule consistent. This is very similar to inserting setup activities in the manufacturing scheduling problem (Barták 2003), which is a problem with some but weak planning component, where the scheduling component prevails. Our approach is based on incremental extension of a partial plan/schedule to which we add the science activities with their supporting (uplink and downlink) activities. This is similar to the approach used in the Mars Exploration Rovers employed by MAPGEN (Bresina et al. 2005). Science activities to be scheduled are explored by backtracking search while the time allocation of activities is done using a form of local search. Some pre-processing and post-processing is used to add the maintenance and pointing activities.

The paper is organized as follows. We will first shortly describe the MEX domain, the full description can be found in (Fratini and Policella 2012). Then we will introduce the core concept of our approach that is is based on maintaining a single timeline of activities. After that we will describe the details of the proposed scheduling algorithm including the pre-processing and post-processing stages. The paper will be concluded by the description of current user interface and summary of some initial experiments.

Problem Description

The *MEX domain* was proposed as a challenge problem for the Fourth International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS). The planning and scheduling problem is in detail described in (Fratini and Policella 2012). For this section we extracted its core parts that are necessary to understand the rest of the paper (the proposed solving techniques). The problem description for ICKEPS was already abstracted but still kept to be realistic. Recall that the current approach to solve the problem is based on manual scheduling by two teams of human planners and there is no automated approach covering the whole problem yet.

The *Mars Express Orbiter* (MEX) is a spacecraft operating at Mars orbit. The orbiter carries six operating scientific instruments called *payloads* that collect data about Martians atmosphere, planet's structure and its geology:

- ASPERA - Energetic Neutral 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.

There is one more instrument VMS (Visual Monitoring Camera) that is no more used and hence omitted from planning. The major task is to schedule as many experiments on the payloads as possible. In general, this is an oversubscribed scheduling problem though we have not performed analysis with more scientific requests yet.

Each payload has an accompanying *data store* with limited capacity. This data store is used exclusively by a given payload (no sharing), the data produced in the experiments is saved on these data stores and has to be downloaded to the Earth before it gets overwritten by data from another experiment. The experiments require particular position of the spacecraft in orbit and particular *pointing* which describes the orientation of the spacecraft in orbit. The spacecraft can focus on Mars in the *Inertial* (NAD) or *Fixed* (FIX) pointing – required by certain experiments – or the spacecraft can be directed on Earth in the *Earth* pointing – for uplink and downlink. The spacecraft can maintain NAD or FIX pointings for a limited time only. To transfer from one pointing to another, the orbiter needs to perform a *Pointing Transition Action* (PTA).

The main action of the spacecraft is a scientific experiment, that is fully described as a *Payload Operation Request* (POR). A POR action always has a fixed offset time restricting when it can be scheduled. The offset is tied to the MEX's passage of the orbit's pericenter. A single orbit takes roughly 7 hours to complete. It should be noted, that more instruments can perform POR actions at the same time, if the conditions for operating the instrument are met. The only condition considered in our solution is the pointing required by the specific instrument. The required pointing is specified in the Payload Operation Request. It is expected that there will be more PORs than the orbiter can accommodate, but in our current experimental setting, all PORs can be scheduled.

In addition to experiments, the spacecraft is expected to perform regular *maintenance* routines which need to run every 3 to 5 orbits. These maintenance actions must be part of the schedule.

The description of every action taken by the orbiter needs to be *uplinked* from the Earth to the spacecraft as a *TeleCommand* (TC). This is a part of communication actions, which can be either *downlink* (DL) of data produced by an experiment or *uplink* (UL) of TCs. The MEX can downlink and uplink data at the same time. For the communication to be possible, the orbiter needs to be in the Earth pointing and a ground station needs to be available at the time of the communication. The description of availability of ground stations as well as their bitrates is one of the problem inputs.

In summary, the problem input consists of the following files:

- spacecraft orbital events produced by flight dynamic (where and when the spacecraft will be),
- ground station availability and antennas transmission bitrates,
- a set of Payload Operation Requests including the amount of data produced.

The goal is to produce a complete plan/schedule of all operations performed by the spacecraft for a given period. In particular there are:

- observation actions (experiments),
- maintenance actions,
- pointing transition actions,
- command uploading actions,
- data downloading actions.

The Core Solving Concepts

Most of the actions to be scheduled at MEX are sequential and there are only specific cases where multiple actions can be performed at the same time. Therefore we have decided to model the domain using a concept that we call a *TimeLine*. Timelines are very popular in space applications. They were first proposed in HSTS (Heuristic Scheduling Testbed System) (Muscettola 1994) as a way to integrate planning and scheduling and then used in systems such as EUROPA (Barreiro et al. 2012) or in the OMPS framework (Fratini, Pecora, and Cesta 2008). We differ from existing timelines by using a single TimeLine describing the sequence of time windows containing particular actions.

A TimeLine is a linearly ordered sequence of windows where each window has a non-zero duration. The TimeLine itself has a start time and an end time. There are *windows* covering the whole TimeLine without gaps. Note that the number of windows can (and will) dynamically change during the scheduling process. See Figure 1 for a graphical representation of a part of TimeLine containing several different windows and actions.

We assume that the following properties are always valid for a TimeLine:

1. Each window has a non-zero duration.
2. There are no gaps between the adjacent windows.
3. A window is either empty, or contains only actions that can be conducted under identical conditions.
4. The TimeLine covers the entire time period of the schedule (for example one week); this time period is fixed and does not change during the scheduling process.
5. If a window is not empty, it wraps tightly around the actions in it (the window starts when the first action starts and ends when the last action ends).
6. No action takes more than one window.
7. Only science and communication windows can contain more actions. Science windows may especially contain more actions for the same payload (if the actions fit, of course).

On top of that, we call a TimeLine *consistent* when:

C1 An empty window has no empty neighbors.

C2 All scheduled actions satisfy all domain constraints (that is, the timeline represents a valid MEX plan).

We start with a single empty window spanning over the whole scheduling period. We then add the requested MEX orbiter actions to the windows using the following two operations on the windows: window splitting and adding actions.

Window splitting Any empty window can be split into two empty windows at any time. Note that this makes the timeline temporarily *inconsistent* as the condition (C1) is violated. For this reason, an empty window is only being split when we are adding an action into it. Analogically, two adjacent empty windows can be merged into one. We never split a window that already contains some actions.

Adding actions When adding an action to a window, the only requirement is that the desired action time intersects the window. If the action does not fit perfectly inside the window, the window is stretched (shortening the empty window around the action). The windows are always stretched just enough to fit the action perfectly without empty margins. Note that a window can contain multiple actions with different start and end times. The general formula for determining the window start and end times is (for a non-empty window):

- $start = min\{start(a)|a \in actions(window)\}$
- $end = max\{end(a)|a \in actions(window)\}$

The main task of the MEX domain is to schedule *payload operation requests* (or PORs) into a fixed TimeLine satisfying a number of conditions. The MEX orbiter is also required to regularly perform other preset actions: maintenance and reserved communication windows. We will denote all of these actions as *tasks* in this paper.

Each task requires a specific state of the orbiter (pointing, data store load, orbit phase etc.). To maintain the constraints specifying these states, the orbiter needs to perform additional actions on top of the tasks specified earlier. We simply schedule these *related actions* that need to accompany the task together with the task. For the POR tasks these related actions are:

1. downlink of data generated by the POR
2. uplink of TCs for the downlink action
3. uplink of TCs for the POR action

For the maintenance tasks, only the action (3) is required.

The Scheduling Algorithm

We have decided to use an ad-hoc planning approach using the TimeLine concept described above. Our algorithm can be divided into four stages: preprocessing of PORs, scheduling the tasks (using depth-first search), scheduling the related actions (using local search), and a postprocessing phase, which cleans up the schedule and adds actions that are trivial to schedule.

Preprocessing

The preprocessing phase deals with the following:

1. generate an appropriate number of maintenance tasks,
2. generate the related actions for each POR and for 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.

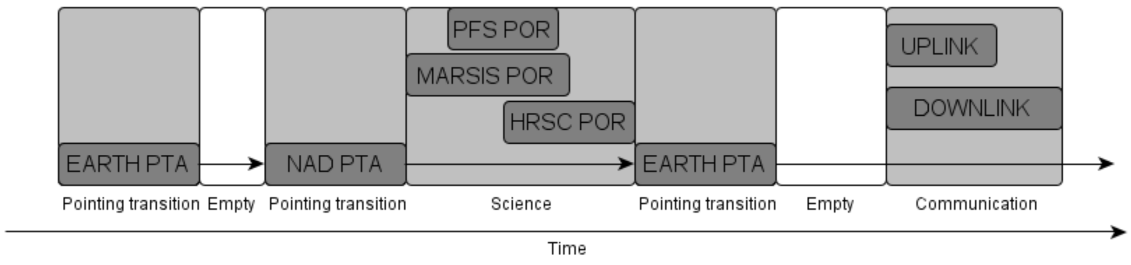

Figure 1: An example part of a timeline

Main search

The goal of the main scheduling stage is to allocate the tasks to time windows. The scheduling algorithm is divided into two phases:

1. Pre-schedule all maintenance tasks.
2. Use depth-first search (DFS) to schedule all POR tasks

Maintenance scheduling Scheduling of the maintenance actions is straightforward. Since we start with an empty timeline, we don't have to worry about the conflicts with other actions so we just schedule all the maintenance actions spaced four orbit-lengths apart. At the end of this step, we have a timeline consisting only of empty and maintenance windows, which we pass on to the next phase. Note that the times of activities decided at this stage are not final because the maintenance actions can be (and usually will be) shifted when scheduling the remaining actions.

POR scheduling The main POR tasks are scheduled using a classical depth-first search technique with the following steps:

1. Select and remove the first unscheduled task
2. Check if the POR action can be scheduled into any window in the TimeLine (starting from the first window) – this is done using the function $SchedulePor(por, w)$ described in Algorithm 4. The POR is successfully scheduled only if all the related actions can be scheduled as well. This logic is integrated in the $SchedulePor$ algorithm. We will describe the procedure in detail when we get to related actions further in the paper.
3. If the POR action was successfuly scheduled, continue with scheduling the next POR (step 1).
4. If the POR action (together with its related actions) cannot be scheduled into any window in the TimeLine, backtrack to the previously scheduled POR and try to find another window for it.

Note, that the TimeLine is always consistent after finishing each iteration. This is achieved by adding/removing the POR and its related actions always together which guarantees satisfying the MEX constraints (property C2). The property C1 is easily achieved locally when adding/removing actions.

The specific algorithm for checking whether a POR can be added to a given window will be described in the next section.

Local search

This section contains the in-depth description of the algorithm used to actually fit the POR actions and their related actions into the TimeLine. As the reader might have noticed, the main DFS algorithm is quite straightforward. This is because most of the actual scheduling is done in the local search phase by shifting the scheduled actions and adding new ones.

Before we describe how the different action types are precisely scheduled, we shall introduce the two main procedures of our algorithm's local search:

- fitting an action into a window,
- shifting scheduled actions in the timeline.

Algorithm 1 CanFit(a,w,t)

```
Require: $t \geq$ StartTime($w$)
  if $t$ + duration($a$) $>$ EndTime($w$) then
    if Next($w$) is empty then
      return $t$ + duration($a$) $\leq$ EndTime(Next($w$))
    else
      $shiftRight \leftarrow$ GetWindowShift(Next($w$),1)
      return $t$ + duration($a$) $\leq$ EndTime($w$) + $shiftRight$
    end if
  else
    return true
  end if
```

Fitting For fitting an action into a window, we use the function $CanFit(a, w, t)$ described in Algorithm 1. This function determines if action a can be added to window w in time t. As we can see, the algorithm only deals with the timing of the actions, it doesn't check any other constraints. When the action can't fit and an empty window is detected as the next window, we can extend w into the empty window. Otherwise we have to try and shift the adjacent window to make room for the new action. Checking for action-specific constraints is usually done only after this check passes. We will talk about these constraints as we encounter each action type later in the paper.

Algorithm 2 GetActionShift(a,dir)

```
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 $|p_t - t|$
```

Shifting Window shifting is based on two algorithms realized by the function $GetActionShift(a, dir)$ described in Algorithm 2 and the function $GetWindowShift(w, dir)$ described in Algorithm 3. The action shift serves as a subroutine of the window shift algorithm. Both functions return the maximum possible offset a (or w resp.) can be shifted by in the given direction dir. In both functions, we assume dir

Algorithm 3 GetWindowShift(w,dir)

```
Require: $w$ is not empty
  $shift \leftarrow min\{GetActionShift(a, dir)|a \in$Actions($w$)$\}$
  if Adjacent($w$,$dir$) is empty then
    return min($shift$,Duration(Adjacent($w$,$dir$)))
  else
    return min($shift$,GetWindowShift(Adjacent($w$,$dir$),$dir$)
  end if
```

-1 to mean *left* and +1 to mean *right* (this convention is used everywhere in the planner).

The basic idea is that if we assume an action-filled window in an otherwise empty timeline, we can shift the window in either direction as far as the most-constrained action allows (precisely this is calculated in $GetWindowShift$). If we consider other action-filled windows around our window w, we obviously need to shift the windows adjacent to w in dir as well, which is calculated by the recursion in $GetWindowShift$. To simplify the problem, we only consider windows in dir up to the first empty window. So we never shift empty windows, we just shrink the first empty window encountered to make room for the shifted actions. As we can see, the possible shift offset is then limited by the size of the first empty window. This limitation however doesn't seem to affect the algorithm in any bad way and reduces the complexity of the function $GetWindowShift$ significantly in practice.

The only thing from the algorithms that might need more explanation is the assignment of p_t in $GetActionShift$. In simple words, p_t is the farthest start time (in w) of a where all constraints for a would still be satisfied. This is represented by the $ConstraintsSatisfied$ statement. The statement returns true if constraints for a are satisfied in the whole interval $[t, t_0]$. We ignore all conflicting actions in the process of this check. We assume they will be shifted out as well, hence checking the adjacent windows in $GetWindowShift$. In practice, the action shifting algorithm is tailored for each action type and its constraints to speed the checking up. The basic skeleton is however identical as described for all action types.

See Figure 2 for an example illustrating the shifting process. In the example, we are trying to add a new action into the first empty window. As the top part of the figure shows, the new action would collide with action 3 and hence we need to shift the window with action 3 to make the room for the new action. In the example, we assume that the function $GetWindowShift$ returned a possible shifting offset marked by the dashed line (Shift frame). As we can clearly see, we only need to shift the window containing action 3 by the offset marked by the thick dashed line. Since that is smaller than the maximum possible offset, we can perform the shift. This results in shortening the rightmost empty window and lengthening the middle empty window so the new action can now fit.

When we will refer to shifting later in this paper, we will always have this process in mind. If we say that we shift a window (an action resp.), we mean running the function $GetWindowShift$ ($GetActionShift$ resp.) to determine the maximum possible offset and then shifting the window (the action resp.) by an offset not greater than the allowed maximum.

Scheduling a task with related actions The POR and Maintenance tasks (the main tasks) have other tasks related to them - the communication tasks. There are three of them needed for the main POR task to complete - upload of the commands starting (and stopping) the main task, download of the produced data, and upload of the command which starts the download (we call it upload-for-download). This abstraction leaves out the uplink for these TCs, but such recursion would never stop and thus we address it in the post-processing phase.

Each action's scheduling is limited by two sets of constraints:

- *Temporal constraints related to the main task* specify for example that the downlink task has to be scheduled after the main task and after the upload-for-download.
- *Constraints related to MEX operations* include having the

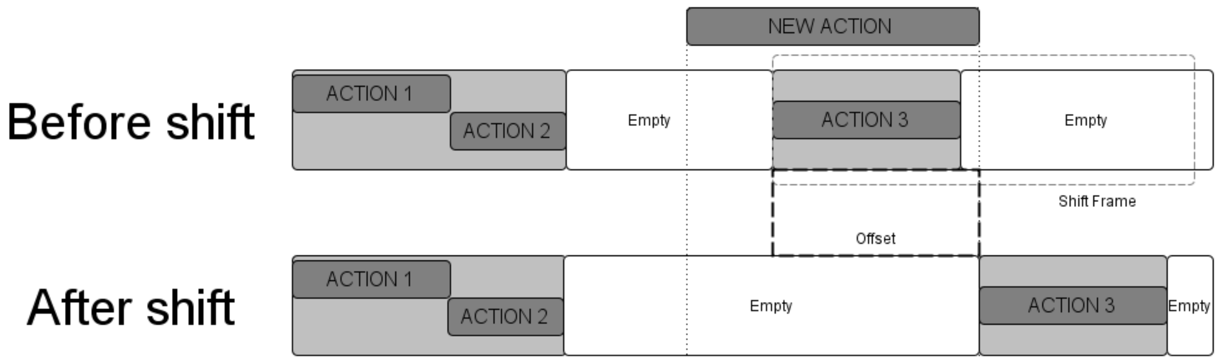

Figure 2: A diagram illustrating window shift

correct pointing, having a ground station available (for uplink/downlink), being in the correct orbit stage, having the required altitude or illumination and so forth.

Algorithm 4 SchedulePOR(por,w)

```
if $w$ does not cover correct orbit stage for $por$ then
    return false
end if
$t \leftarrow$ time of desired orbit stage in $w$
$added \leftarrow$ false
$pta \leftarrow$ NULL
if CanFit($por$,$w$,$t$) and Compatible($por$,$w$,$t$) then
    if Pointing($w$) = Pointing($por$) then
        add($w$, $por$, $t$)
        $added \leftarrow$ true
    else
        $pta \leftarrow$ new PTA(Pointing($por$))
        if AddPointing($w$,$pta$,$t-$PTAduration) then
            add($w$, $por$, $t$)
            $added \leftarrow$ true
        end if
    end if
end if
if $added$ then
    if ScheduleRelatedActions($por$) then
        return true
    else
        remove($w$,$por$)
        if $pta != $ NULL then
            remove(window($pta$),$pta$)
        end if
    end if
end if
return false
```

Local search - POR action The algorithm for determining whether a POR action can be added to a specific TimeLine window is described in Algorithm 4.

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 simply takes each related action in turn (uplinks and 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 these related actions are removed) and the function returns false. If all related actions are scheduled successfully, the function returns true.

Local search - related actions The algorithm for scheduling the related actions is based on searching the windows around the main POR action in a direction specific for each action type. The algorithm can be briefly described by:

1. searching all windows in the TimeLine in the given direction,
2. picking up the best fit according to *heuristic*,
3. if no such fit is found, fail, otherwise return the exact time using *heuristic*

Note, that whether or not we try to shift the actions already present in the tried window is determined by the heuristic.

The algorithm for determining the maximum possible shift for a specific action is straightforward. It uses the action constraints to calculate the possible timeframe in the current window and the closest adjacent window. The offset between the start (or end if shifting to the right) of this timeframe and the current action time is then returned.

Local search - completion As was said above, the heuristic is always dependent on the action type. We will look at the different heuristics in detail in this section.

Communication actions - heuristic

1. Find the first communication window in the specified direction. If it is large enough (or can be extended), fit the action to such time, that the action duration is minimal. Since the action duration is dependent only on the groundstation's bitrate, choosing the optimal groundstation event in the window (or adjacent to it) is very simple.
2. If no communication window is found, place the action into the first empty window that can fit the action (there is a groundstation with sufficient bitrate) and either has the correct pointing already or a new pointing transition action can be added before it.

Pointing action The pointing actions are a bit special because they are not directly generated in the preprocess phase but rather generated on-demand whenever we need to add an action to a window with incompatible pointing. Another issue to consider is 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.

Algorithm 5 PointingFix(w,w_1,t,p,p_1)

```
$w_f \leftarrow w$
**while** endTime(next($w_f$)) $\leq t$ + maxDuration($p$) **and** $w_f$ != $w_1$ **do**
   $w_f \leftarrow$next($w_f$)
**end while**
**if** $p_1$ != $\emptyset$ **then**
   $pa_f \leftarrow$ emptyPointingAction()
**else**
   $pa_f \leftarrow$ pointingAction($p_1$)
**end if**
**if** CanFit($pa_f$,$w_f$) **then**
   add($pa_f$,$w_f$)
   **return** true
**else**
   **return** false
**end if**
```

In either case, we address the problem by scheduling another pointing action pa_f using the algorithm described in Algorithm 5.

The reader may have noticed that 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 (Nau, Ghallab, and Traverso 2004).

The full algorithm for adding a pointing action p_a to window w before time t is described in Algorithm 6.

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.

Let us demonstrate the above in an example. See Figure 3 for reference. In the example, we are trying to schedule a new POR for the SPICAM instrument, which requires the FIX pointing. As we can see, the MEX is in the earth pointing in the timeframe, where we want to add the POR action. 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 resources planner Filuta handles new events in timelines (Dvořák and Barták 2010).

Algorithm 6 AddPointing(p_a,w,t)

```
$w \leftarrow$ split($w$,$t$) {left portion of $w$ before $t$}
$OK \leftarrow$ false
**if** CanFit($p_a$,$w$) **then**
   add($p_a$,$w$)
   $p \leftarrow$ TargetPointing($p_a$)
   $t \leftarrow$ StartTime($p_a$)
   **if** $\exists w_1$ : $w_1$ is conflict window of $w$ **then**
      $p_1 \leftarrow$ RequiredPointing($w_1$)
      $OK \leftarrow$ PointingFix($w$, $w_1$, $t$, $p$, $p_1$)
   **else if** maxDuration($p$) ¡ $\infty$ **then**
      $OK \leftarrow$ PointingFix($w$, EndWindow($timeLine$), $t$, $p$, $\emptyset$)
   **else**
      $OK \leftarrow$ true
   **end if**
**else**
   merge($w$)
   **return** false
**end if**
**if** $OK$ **then**
   **return** true
**else**
   remove($p_a$,$w$)
   merge($w$)
   **return** false
**end if**
```

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).

The first simplification was ignoring PORs on the ASPERA payload. This one can work even during communication sessions and needs no specific pointing. The only constraint is that it cannot run during maintenance, which is simply achievable in this phase. So the algorithm just finds a place without maintenance and schedules this POR with its related actions. Scheduling the related actions will always work due to the assumptions on communication sessions (see below).

We further facilitated the problem by not handling telecommands which instruct MEX to move a TC from its internal memory to the Mission Timeline. 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. Allowing to schedule ASPERA data downlink in postprocessing is done by an analogous principle on downlink sessions.

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.

User interface

We created a simple graphical user interface (GUI) in Java which allows to select the input files, run the planner and inspect the generated timeline. The interface doesn't allow interacting with the timeline (other than zooming and panning). The timeline can be copied to a clipboard or saved to an image file (the textual form of the timeline is printed to stdout).

A preview of a timeline is provided in Figure 4. There is a row for each payload, for maintenance, for uplink and downlink and for pointing transitions. These rows contain boxes representing the scheduled actions (the width of the box corresponds with the duration of the action). Very short actions (which would be displayed as a few pixels) have a 'glow' around them to make them better recognizable. There are also two rows displaying the state of the orbiter. One for Mission Timeline (MTL), which shows the used capacity of this memory in time. The last row displays the used capacities for all other data stores (all scaled to be at most as high as the row).

We are planning to add two more major features to the GUI:

- interaction with the boxes to show more details about the corresponding actions (like showing the TCs uploaded by an uplink action),
- connecting main tasks with their related actions by arrows. This should be similar to the Olligram as described in (Rabenau et al. 2008).

Results

We have evaluated the planner with various sets of randomly generated data inspired by real data provided to us by ESA. The reason why we did not use the real data is some difficulty with translating them to a common format. The data sets range from 30 to roughly 80 PORs. The computation in each case takes less than a second on a regular laptop.

The generated plan has some interesting features. Firstly, uplinks tend to get clustered into 3 or 4 windows spaced roughly regularly, which seems to be optimal from the number of communication sessions required. Downlinks have this tendency as well, but the number of these windows is greater (depends on how data-intensive the PORs are). The algorithm also optimally schedules the uplinks to run in parallel with downlinks, which again reduces the number of required communication windows. Also, more tasks are scheduled to the start of the week (which is a consequence of the DFS scheduling algorithm, which schedules left to right). The effect of this is, that there is usually a lot of free space at the end of the plan.

If we look at Figure 4, we can clearly see the clusters of downlink and uplink actions as well as a number of the pointing fix actions we mentioned earlier in the document.

Testing on a larger data set, we provided data covering about a month of operation and 164 PORs. The computation finished within 280 ms using about 5 MB of RAM. An even larger data set with 192 PORs was scheduled within 600-800 ms using about 10 MB of RAM.

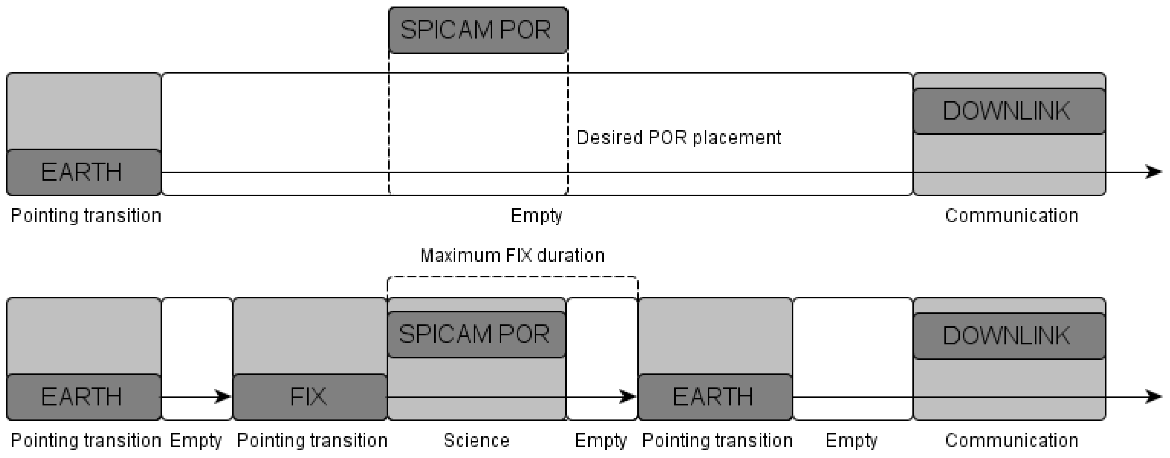

Figure 3: A diagram illustrating POR scheduling with pointing transitions

What is interesting – even in the large data sets there was no need to perform backtracking – because the local search was efficient.

There are some problems in validating these random data sets however as it is virtually impossible to create a schedule manually. Therefore we have little way of knowing if the tested data set has a solution or how does the optimal solution look. We can only run the planner on it and check for any errors in the resulting plan.

We are currently in the process of obtaining more information about the real data sets from ESA to properly test our planner with real data.

Conclusions

The paper presented an ad-hoc approach to solve a complete planning and scheduling problem for the Mars Express Mission. The approach uses a ”human-like” style of scheduling where the operations are added to a partial plan that is modified to accept the new operation. We group the related operations and schedule the group together. For example the observation operation is scheduled together with its supporting operations for uplinking the commands to the orbiter and downloading the obtained data back to the Earth. The data storage capacity constraints are checked when the operation is added and necessary operations to change pointing of the orbiter are also added when necessary. The preliminary results show that this approach works well even for large sets of observation operations to be scheduled. However the results still need to be verified by ESA and experiments with real data need to be performed.

Though the presented approach is an ad-hoc technique developed for the MEX scheduling problem, we believe that the core ideas can be applied in other scheduling problems. In fact, the idea of incremental addition of task to a partial schedule has already been explored when scheduling operations for the Mars Exploration Rovers by the MAPGEN system (Bresina et al. 2005). The idea of locally shifting activities to make space for a new activity is similar to a technique called bulldozing (Smith and Pyle 2004).

Acknowledgements

We would like to thank Simone Fratini and Nicola Policella from ESA for opening this challenge, for useful discussions to clarify the problem specification, and for providing data for initial experiments. The research is supported by the Czech Science Foundation under the contract P202/10/1188.

References

Barreiro, J.; Boyce, M.; Do, M.; Frank, J.; Iatauro, M.; Kichkayloz, T.; Morris, P.; Ong, J.; Remolina, E.; Smith, T.; and Smith, D. 2012. Europa: A platform for ai planning, scheduling, constraint programming, and optimization. In *Proceedings of the Fourth International Competition on Knowledge Engineering for AI Planning and Scheduling (ICKEPS): Design Process Track*.

Barták, R. 2003. Visopt shopfloor: Going beyond traditional scheduling. In *Recent Advances in Constraints. LNAI 2627*, 185–199. Springer Verlag.

Bresina, J.; Jonsson, A.; Morris, P.; and Rajan, K. 2005. Activity planning for the mars exploration rovers. In *Proceedings of ICAPS 2005*, 40–49. AAAI Press.

Cesta, A.; Cortellessa, G.; Denis, M.; Donati, A.; Fratini, S.; Oddi, A.; Policella, N.; Rabenau, E.; and Schulster, J. 2007. Ai solves mission planner problems. *IEEE Intelligent Systems* 22.

Dvořák, F., and Barták, R. 2010. Integrating time and resources into planning. In *Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence*, 71–78. IEEE Computer Society.

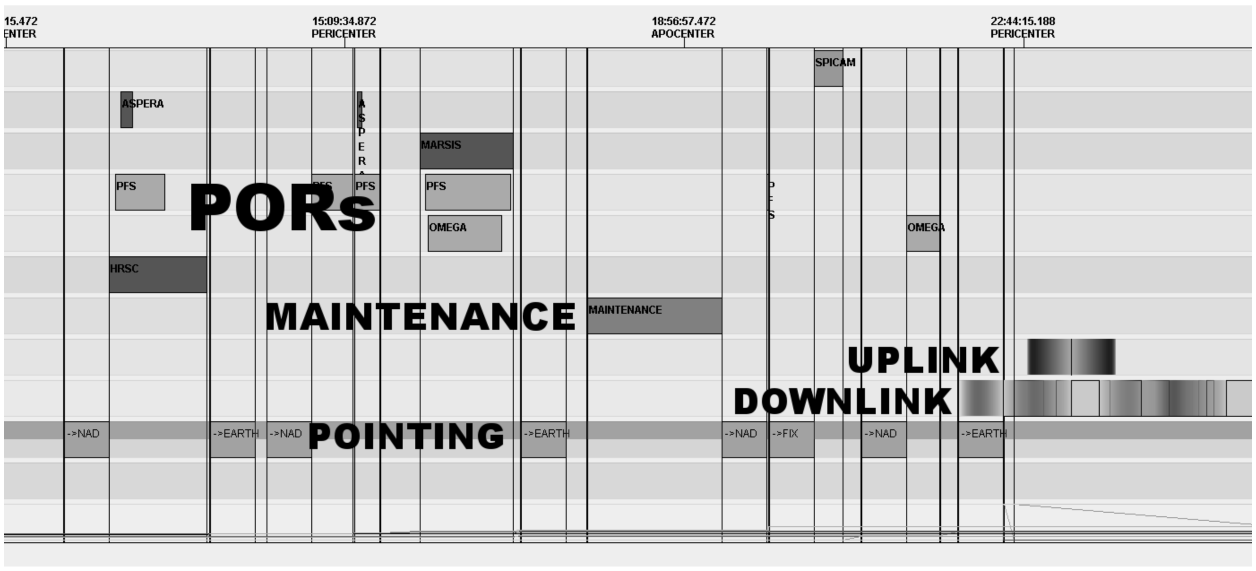

Figure 4: Timeline scheduled from a generated input data set; time goes horizontally

Fratini, S., and Policella, N. 2012. Ickeps 2012 challenge domain: Planning operations on the mars express mission. Available at http://icaps12.poli.usp.br/icaps12/sites/default/files/ickeps/mexdomain/MEX_KEPS_domain_v12.pdf.

Fratini, S.; Pecora, F.; and Cesta, A. 2008. Unifying planning and scheduling as timelines in a component-based perspective. *Archives of Control Sciences* 18(2):231–271.

Muscettola, N. 1994. Hsts: Integrating planning and scheduling. In *Intelligent Scheduling*. Morgan Kauffmann.

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

Rabenau, E.; Donati, A.; Denis, M.; Policella, N.; Schulster, J.; Cesta, A.; Cortellessa, G.; Fratini, S.; and Oddi, A. 2008. 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*.

Smith, T., and Pyle, J. 2004. An effective algorithm for project scheduling with arbitrary temporal constraints. In *Proceedings of AAAI 2004*, 544–549. AAAI Press.