

# Enabling Limited Resource-Bounded Disjunction in Scheduling

## Abstract

We describe three approaches to enabling a *severely* computationally limited embedded scheduler to consider a small number of alternative activities based on resource availability. We consider the case where the scheduler is so computationally limited that it cannot backtrack search. The first two approaches precompile resource checks (called guards) that only enable selection of a preferred alternative activity if sufficient resources are estimated to be available to schedule the remaining activities. The third approach mimics backtracking by invoking the scheduler multiple times with the alternative activities. We present an evaluation of these techniques on Mars mission scenarios (called sol types) from NASA’s next planetary rover where these techniques have been evaluated for inclusion in an onboard scheduler.

## Introduction

Embedded schedulers must often operate with very limited computational resources. This makes it challenging to perform even simple schedule optimization when doing so may use resources needed for yet unscheduled activities.

In this paper, we present three algorithms to enable such a scheduler to consider a very limited type of preferred activity while still scheduling all required, hereafter called *mandatory*, activities. Preferred activities are grouped into *switch groups*, i.e. sets of activities where each activity in the set is called a *switch case*, and exactly one of the activities in the set must be scheduled. They differ only by how much time, energy, and data volume they consume and the goal is for the scheduler to schedule the most desirable activity (also the most resource consuming activity) without sacrificing any other mandatory activity.

The target scheduler is a non-backtracking scheduler baselined for flight onboard the NASA Mars 2020 planetary rover (Rabideau and Benowitz 2017) that schedules in priority first order and never removes or moves an activity after it is placed during a single run of the scheduler. Because the scheduler does not backtrack, it is challenging to ensure that scheduling a more consumptive switch case will not use too many resources and therefore prevent a later, yet unscheduled, mandatory activity from being scheduled.

The onboard scheduler is designed to make the rover more robust to run-time variations by rescheduling multiple

times during execution (Gaines et al. 2016a). If an activity ends earlier or later than expected, then rescheduling will allow the scheduler to consider changes in resource consumption and reschedule accordingly. Our algorithms to schedule switch groups must also be robust to varying execution durations and rescheduling.

The remainder of the paper is organized as follows. First we mathematically show why it is impractical to have a backtracking scheduler given the CPU limitations. Second, we explain the problem specification. Third, we describe two guard approaches to schedule switch groups which involve reserving enough resources to ensure all remaining required activities can be scheduled. Fourth, we describe another approach which emulates very limited backtracking under certain conditions by re-invoking the scheduler multiple times. Finally, we present empirical results to evaluate and compare these three approaches, and explain which method was chosen to be infused into the Mars 2020 onboard scheduler.

## Scheduler Runtime Limitations

The target scheduler has extremely limited CPU resources available<sup>1</sup> and a conservative estimated runtime of 60 seconds. To explain how much the estimated runtime would increase if the scheduler was allowed to backtrack, we consider the search trees explored by a non-backtracking and backtracking scheduler. The non-backtracking scheduler explores a tree  $T_1$  with branching factor  $B_1 = 1$  and height of  $h_A$  where  $h_A$  is also the number of activities to be scheduled. The backtracking scheduler considers a tree  $T_2$  with a branching factor,  $B_2$ , and height  $h_A$  where  $B_2 > 1$ . The total number of nodes in each tree corresponds to the number of times the scheduler will attempt to place an activity. Since the branching factor of  $T_1$  is 1, the total number of nodes in  $T_1$ , given by  $n_{T_1}$ , is  $h_A$ . The non-backtracking scheduler does not consider further search to move activities once they have already been placed. The total number of nodes in  $T_2$  is given by Equation 1.

<sup>1</sup>The RAD750 processor used by the Mars 2020 rover has measured performance in the 200-300 MIPS range. In comparison a 2016 Intel Core i7 measured over 300,000 MIPS or over 1000 times faster. Furthermore, the onboard scheduler is only allocated a portion of the computing cycles onboard the RAD750 resulting computation *several thousand times* slower than a typical laptop.

$$n_{T_2} = 1 + B_2 + B_2^2 + \dots + B_2^{h_A} = \frac{B_2^{h_A+1} - 1}{B_2 - 1} \quad (1)$$

A conservative runtime for a single iteration of M2020 scheduler is  $K = 2$  seconds. This includes (Rabideau and Benowitz 2017) computation of valid intervals for placement, heuristic selection of a start time, scheduling of any required preheat and maintenance activities and updating of the wake/sleep activities of the rover (Chi, Chien, and Agrawal 2019). The scheduler must schedule 30 activities typically and 50 activities maximally for the Mars 2020 use case. The total runtime of searching all nodes in  $T_1$  is given by Equation 2.

$$R_{T_1} = K \times n_{T_1} \quad (2)$$

The runtime for a backtracking scheduler represented by  $T_2$  is given by Equation 3

$$R_{T_2} = K \times n_{T_2} = 2 \times \frac{B_2^{h_A+1} - 1}{B_2 - 1} \quad (3)$$

Figure 1 shows the predicted runtime growth with the introduction of search even with modest increase in branching factor (note Y axis log scale; nominal 30 activities). This graphic highlights that even a modest amount of search is impractical with the extremely limited CPU resources<sup>2 3</sup>.

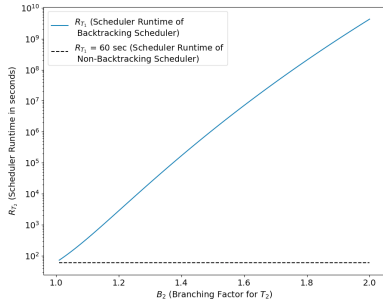


Figure 1: Scheduler runtime versus branching factor using a log scale where  $h_A = 30$ .

## Problem Definition

For the scheduling problem we adopt the definitions in (Rabideau and Benowitz 2017). The scheduler is given a set of activities  $\mathbb{A} = \{A_1 \langle p_1, d_1, T_1, D_1, R_1, e_1, dv_1, \Gamma_1 \rangle \dots A_n \langle p_n, d_n, T_n, D_n, R_n, e_n, dv_n, \Gamma_n \rangle\}$  where:

- $p_i$  is the scheduling priority of activity  $A_i$ ;
- $d_i$  is the nominal, or predicted, duration of activity  $A_i$ ;

<sup>2</sup>Note that the M2020 soft limit of 225 seconds runtime occurs at  $B_2 = 1.07$  exploring 112 nodes with minimum possible nodes = activities i.e. 30 (nominal) 50 (system requirement).  $B_2 = 1.25$  is a 134 minute runtime and  $B_2 = 1.5$  is a 2 week runtime.

<sup>3</sup>Indeed (Chi et al. 2019) describes the ground optimization of scheduling parameters to address the no search limitations onboard.

- $T_i$  is a set of start time windows  $[T_{i,j,start}, T_{i,j,end}] \dots [T_{i,o,start}, T_{i,o,end}]$  for activity  $A_i$ ;
- $D_i$  is a set of activity dependency constraints for activity  $A_i$  where  $A_p \rightarrow A_q$  means  $A_q$  must execute successfully before  $A_p$  starts;
- $R_i$  is the set of unit resources  $R_{i_1} \dots R_{i_m}$  that activity  $A_i$  will use;
- $e_i$  and  $dv_i$  are the rates at which the consumable resources energy and data volume respectively are consumed by activity  $A_i$ ;
- $\Gamma_{i_1} \dots \Gamma_{i_r}$  are non-depletable resources used such as sequence engines available or peak power for activity  $A_i$ .

Each activity in the aforementioned set of activities,  $\mathbb{A}$ , belongs to either a set of *mandatory activities*  $\mathbb{A}^M$ , or a set of *switch group activities*  $\mathbb{A}^G$ , where  $\mathbb{A} = \mathbb{A}^M \cup \mathbb{A}^G$ . Each activity in  $\mathbb{A}^G$  belongs to one and only one switch group  $G_k$  where  $G_k \in \mathbb{G} = \{G_1 \dots G_g\}$ .

The activities within a switch group  $G_k$ , i.e.  $A_i^{G_k}$ , are called *switch cases* of group  $G_k$  and they each vary from each other by how many resources (time, energy, and data volume) they consume.

Each activity is assigned a scheduling priority, which determines the order in which the activities will be considered for scheduling (Chi et al. 2019). The relative priorities of switch cases within each switch group correspond to how consumptive they are. That is, the highest priority switch case in switch group  $G_k$  is also the most consumptive and the lowest priority switch case is the least consumptive.

The scheduling process for each plan begins with the assumption that the rover is asleep for the entire plan horizon. Each time the scheduler places an activity, the rover must be awake and the energy level declines (Chi, Chien, and Agrawal 2019). When scheduling activities, the following plan-wide energy constraints must also be satisfied:

- **Minimum State of Charge (SOC).** The SOC (battery energy level) cannot go below the Minimum SOC at any point and an activity will not be scheduled if scheduling it will cause such a violation;
- **Minimum Handover SOC.** The state of charge cannot be below the Minimum Handover SOC at the *Handover Time*, in effect when the next schedule starts (e.g., the handover SOC of the previous plan is the expected beginning SOC for the subsequent schedule);
- **Maximum SOC.** The state of charge cannot exceed a maximum allowed SOC, due to the adverse effect on long term battery performance.

## Goal of Scheduler

The goal of the scheduler is to generate a schedule in which:

1. All mandatory activities in  $\mathbb{A}^M$  are scheduled;
2. For each switch group  $G_k$ , one and only one activity (switch case) in  $G_k$  is scheduled. It is preferable to schedule a more resource intensive switch case in each switch group, but not at the expense of another mandatory activity;

3. All individual activity temporal, resource, and dependency constraints are satisfied ( $T_i$ ,  $D_i$ ,  $R_i$  for each  $A_i$ );
4. All plan-wide energy constraints are satisfied.

During execution, the scheduler will be invoked multiple times in response to execution variations, e.g. activities end earlier or later than expected (Chi et al. 2018). Each time the scheduler is invoked, it must generate a schedule which satisfies the above conditions.

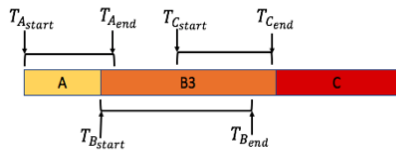
### Switch Group Example

A specific application of switch groups involves using a Mars 2020 instruments which takes images to fill mosaics that may vary in size, e.g.  $1 \times 4$ ,  $2 \times 4$ , or  $4 \times 4$  mosaics. Taking larger mosaics might be preferable, but taking a larger mosaic uses more time and energy, and produces more data volume. These alternatives would be modeled by a switch group as follows:

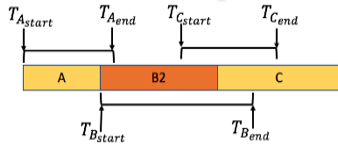
$$\text{Switch Group} = \begin{cases} \text{Mosaic}_{1 \times 4} & d = 100 \text{ sec} \\ \text{Mosaic}_{2 \times 4} & d = 200 \text{ sec} \\ \text{Mosaic}_{4 \times 4} & d = 400 \text{ sec} \end{cases} \quad (4)$$

The desire is for the scheduler to schedule the activity  $\text{Mosaic}_{4 \times 4}$  but if it does not fit then try scheduling  $\text{Mosaic}_{2 \times 4}$ , and eventually try  $\text{Mosaic}_{1 \times 4}$  if the other two fail to schedule. It is not worth scheduling a more consumptive switch case if doing so will prevent a future lower priority mandatory activity from being scheduled due to lack of resources. Because our computationally limited scheduler cannot search or backtrack, it is a challenge to predict if a higher level switch case will be able to fit in the schedule without consuming resources that will cause another mandatory activity to be forced out of the schedule.

Consider the example in Figure 2 where the switch group consists of activities B1, B2, and B3 and  $d_{B3} > d_{B2} > d_{B1}$ . Each activity in this example also has one start time window from  $T_{i_{start}}$  to  $T_{i_{end}}$ .



(a) Scheduling B3 first prevents activity C from being scheduled within its start time window.



(b) B2 can be successfully scheduled without dropping any other mandatory activities.

Figure 2: Challenge to Schedule Switch Cases.

B3 is the most resource intensive and has the highest priority within the switch group so the scheduler will first try

scheduling B3. As shown in Figure 2a, scheduling B3 will prevent placing activity C at a time satisfying its execution constraints. So, B3 should not be scheduled.

The question might arise as to why switch groups cannot simply be scheduled last in terms of scheduling order. This is difficult for several reasons: 1) we would like to avoid gaps in the schedule which is most effectively done by scheduling primarily left to right temporally, and 2) if another activity is dependent on an activity in a switch group, then scheduling the switch group last would introduce complications to ensure that the dependencies are satisfied. In what follows we address methods to effectively schedule switch groups.

### Guard Approaches

First we will discuss two guard methods to schedule switch groups, the Fixed Point guard and the Sol Wide guard. Both of these methods attempt to schedule switch cases by reserving enough time and energy to schedule the remaining mandatory activities. For switch groups, this means that resources will be reserved for the least resource consuming activity since it is required to schedule exactly one activity in the switch group. The method through which both of these guard approaches reserve enough time to schedule future mandatory activities is the same. They differ in how they ensure there is enough energy. While the Fixed Point guard reserves enough energy at a single fixed time point, the Sol Wide guard attempts to reserve sufficient energy by keeping track of the energy balance in the entire plan, or sol.

In this discussion, we do not attempt to reserve data volume while computing the guards as it is not expected to be as constraining of a resource as time or energy. We aim to consider data volume in future work.

Both the time and energy guards are calculated offline before execution occurs. Then, each time rescheduling occurs during execution, the constraints given by both the time and energy guards are applied to ensure that scheduling a higher level switch case will not prevent a future mandatory activity from being scheduled. If activities have ended sufficiently early and freed up resources, then it may be possible to reschedule with a more consumptive switch case.

Before we discuss how each guard approach uniquely reserves enough energy for remaining mandatory activities, we will present how the Fixed Point and Sol Wide guards both ensure enough time will be reserved to schedule remaining mandatory activities while attempting to schedule a more resource consuming switch case.

### Guarding for Time

First, we will discuss how the Fixed Point and Sol Wide guards reserve enough time to schedule remaining mandatory activities.

A *nominal schedule* is generated using predicted activity durations and only the nominal, or least resource consuming switch case in each switch group. The nominal schedule is then used to find the time at which the nominal switch case is scheduled to complete, as shown in Figure 3.

We then manipulate the execution time constraints of the more resource intensive switch cases, B2 and B3 in Figure

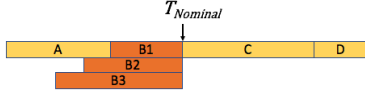


Figure 3: A, C, and D are mandatory activities and B1 is the nominal switch case.  $T_{Nominal}$  is the time at which B1 is scheduled to end in the nominal schedule.

3, so that they are constrained to complete by  $T_{Nominal}$  as shown in Equation 5. Thus, a more time consuming switch case will not use up time from a later mandatory activity. If an activity has more than one start time window, then we only alter the one which contains  $T_{Nominal}$  and remove the others. If a prior activity ends earlier than expected during execution and frees up some time, then it may be possible to schedule a more consumptive switch case while obeying the time guard given by the altered execution time constraints.

$$T_{B_{ij}.end} = T_{Nominal} - d_{B_i} \quad (5)$$

Next, we discuss how the guard methods uniquely ensure that scheduling a more consumptive switch case will not cause violations of the Minimum SOC or Minimum Handover SOC constraint when scheduling mandatory activities.

### Fixed Point Guard Approach

**Fixed Point Minimum State of Charge Guard** The Fixed Point method attempts to ensure that scheduling a more resource consuming switch case will not violate the Minimum SOC while scheduling any future mandatory activities by reserving sufficient energy at a single, fixed point in time,  $T_{Nominal}$  as shown in Figure 4. The guard value for the Minimum SOC is the state of charge value at  $T_{Nominal}$  while constructing the nominal schedule. When attempting to schedule a more resource intensive switch case, a constraint is placed on the scheduler so that the energy cannot fall below the Minimum SOC guard value at time  $T_{Nominal}$ . If an activity ends early (and uses fewer resources than expected) during execution, it may be possible to satisfy this guard while scheduling a more consumptive switch case.

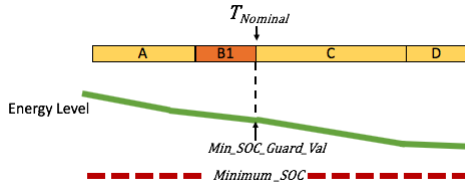


Figure 4: A, C, and D are mandatory activities and B1 is the nominal switch case in the nominal schedule. A constraint is placed so that the energy cannot dip below  $Min\_SOC\_Guard\_Val$  at time  $T_{Nominal}$  while trying to schedule a more consumptive switch case.

**Fixed Point Handover State of Charge Guard** The Fixed Point method guards for the Minimum Handover SOC by first calculating how much extra energy is left over in the

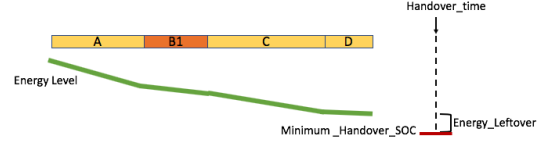


Figure 5: A, C, and D are mandatory activities and B1 is the nominal switch case in the nominal schedule. A constraint is placed so that the extra energy a higher level switch case consumes cannot exceed  $Energy\_Leftover$ .

nominal schedule at handover time after scheduling all activities, as shown in Figure 5.

Then, while attempting to place a more consumptive switch case, a constraint is placed on the scheduler so that the extra energy required by the switch case does not exceed  $Energy\_Leftover$  from the nominal schedule in Figure 5. For example, if we have a switch group consisting of three activities, B1, B2, and B3 and  $d_{B3} > d_{B2} > d_{B1}$  and each switch case consumes  $e$  Watts of power, we must ensure that the following inequality holds at the time the scheduler is attempting to schedule a higher level switch case:

$$(d_{B_i} \times e_{B_i}) - (d_{B_1} \times e_{B_1}) \geq Energy\_Leftover \quad (6)$$

There may be more than one switch group in the schedule. Each time a higher level switch case is scheduled, the  $Energy\_Leftover$  value is decreased by the extra energy required to schedule it. When the scheduler tries to place a switch case in another switch group, it will check against the updated  $Energy\_Leftover$ .

### Sol Wide Guard Approach

**Sol Wide Handover State of Charge Guard** The Sol Wide handover SOC guard only schedules a more resource consumptive switch case if doing so will not cause the energy to dip below the Handover SOC at handover time. First, we use the nominal schedule to calculate how much energy is needed to schedule remaining mandatory activities. Having a Maximum SOC constraint while calculating this value may produce an inaccurate result since any energy that would exceed the Maximum SOC would not be taken into account. So, in order to have an accurate prediction of the energy balance as activities are being scheduled, this value is calculated assuming there is no Maximum SOC constraint. The Maximum SOC constraint is only removed while computing the guard offline to gain a clear understanding of the energy balance but during execution it is enforced.

As shown in Figure 6, the energy needed to schedule the remaining mandatory activities is the difference between the energy level just after the nominal switch case has been scheduled, call this  $E1$ , and after all activities have been scheduled, call this energy level  $E2$ .

$$Energy\_Needed = E1 - E2 \quad (7)$$

Then, a constraint is placed on the scheduler so that the energy value after a higher level switch case is scheduled must be at least the value given by Equation 8.



(a) E1 is the energy level of the nominal schedule with no Maximum SOC constraint after all activities up to and including the nominal switch case (A, D, B1) have been scheduled.



(b) E2 is the energy level of the nominal schedule with no Maximum SOC constraint after all activities in the nominal schedule have been scheduled. The activities were scheduled in the following order: A, D, B1, C, E.

Figure 6: Calculating Energy Needed to Schedule Remaining Mandatory Activities.

$$Energy \geq Min\_Handover\_SOC + Energy\_Needed \quad (8)$$

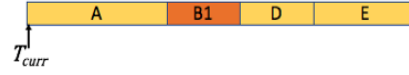
By placing this energy constraint, we prevent the energy level from falling under the Minimum Handover SOC by the time all activities have been scheduled.

**Sol Wide Minimum State of Charge Guard** To limit the possibility of the energy falling below the Minimum SOC constraint, we run a Monte Carlo simulation of execution offline while computing the sol wide energy guard. We use the Monte Carlo to determine if a mandatory activity was not scheduled due to a more consumptive switch case being scheduled earlier. If this occurs in any of the Monte Carlo simulations, then we increase the guard constraint in Equation 8. Each Monte Carlo differs in how long each activity takes to execute compared to its original predicted duration in the schedule. If a mandatory activity was not executed in any of the Monte Carlo runs and a more resource consuming switch case was executed before the time at which that mandatory activity was scheduled to complete in the nominal schedule, then we increase the Sol Wide energy guard value in Equation 8 by a fixed amount. We aim to compose a better heuristic to increase the guard value as future work.

### Multiple Scheduler Invocation Approach

The Multiple Scheduler Invocation (MSI) approach emulates backtracking in a very limited context by reinvoking the scheduler multiple times with the switch cases. MSI does not require any precomputation offline before execution as with the guards, and instead reinvokes the scheduler multiple times during execution. The scheduler reschedules (e.g., when activities end early) with only the nominal switch case as shown in Figure 7a until an MSI trigger is satisfied. At this point, the scheduler is reinvoked multiple times, at most once per switch case in each switch group. In the first

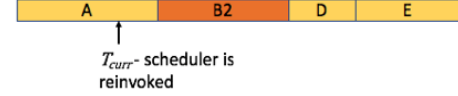
MSI invocation, the scheduler attempts to schedule the highest level switch case as shown in Figure 7b. If the resulting schedule does not contain all mandatory activities, then the scheduler will attempt to schedule the next highest level switch case, as in 7c, and so on. If none of the higher level switch cases can be successfully scheduled then the scheduler is regenerated with the nominal switch case. If activities have ended early by the time MSI is triggered and resulted in more resources than expected, then the goal is for this approach to generate a schedule with a more consumptive switch case if it will fit (assuming nominal activity durations for any activities that have not yet executed).



(a) MSI has not yet begun. Currently, the nominal switch case, B1, is scheduled.



(b) MSI begins. Scheduling the highest level switch case, B3, prevents D from being scheduled. Therefore, try B2.



(c) B2 is successfully scheduled along with the other mandatory activities so MSI is complete.

Figure 7: Order of MSI Invocations.

There are multiple factors that must be taken into consideration when implementing MSI:

**When to Trigger MSI** There are two options to determine when to trigger the MSI process (first invocation while trying to schedule the switch case):

1. *Time Offset*. Start MSI when the current time during execution is some fixed amount of time,  $X$ , from the time at which the nominal switch case is scheduled to start in the current schedule.
2. *Switch Ready*. Start MSI when an activity has finished executing and the nominal switch case activity is the next activity scheduled to start (shown in Figure 8).

**Spacing Between MSI Invocations** If the highest level switch case activity is not able to be scheduled in the first invocation of MSI, then the scheduler must be invoked again. We choose to reschedule as soon as possible after the most recent MSI invocation. This method risks over-consumption of the CPU if the scheduler is invoked too frequently. To handle this, we may need to rely on a process within the scheduler called *throttling*. Throttling places a constraint

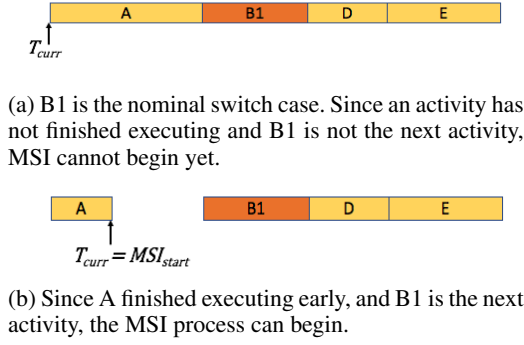


Figure 8: MSI Switch Ready.

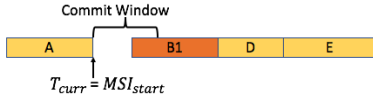


Figure 9: Switch case is committed during MSI.  $T_{curr}$  is the current time during execution.  $MSI_{start}$  is the time at which MSI begins. The nominal switch case, B1, is committed when MSI begins.

which imposes a minimum time delay between invocations, preventing the scheduler from being invoked at too high of a rate. An alternative is to reschedule at an evenly split, fixed cadence to avoid over-consumption of the CPU; we plan to explore this approach in the future.

**Switch Case Becomes Committed** In some situations, the nominal switch case activity in the original plan may become committed before or during the MSI invocations (Figure 9). An activity is *committed* if its scheduled start time is between the start and end of the commit window (Chien et al. 2000). A committed activity cannot be rescheduled and is committed to execute. If the nominal switch case is committed, the scheduler cannot elevate to a preferred switch case. There are two ways to handle this situation:

1. *Commit the activity.* Keep nominal switch case committed and do not try to elevate to a higher level switch case.
2. *Veto the switch case.* Veto the nominal switch case so that it is no longer considered in the current schedule. When an activity is vetoed, it is removed from the current schedule and will be considered in a future scheduler invocation. Therefore, by vetoing the nominal switch case, it will no longer be committed and the scheduler will continue the MSI invocations in an effort to elevate the switch case.

**Handling Rescheduling After MSI Completes but before the Switch Case is Committed** After MSI completes, there may be events that warrant rescheduling (e.g., an activity ending early) before the switch case is committed. When the scheduler is reinvoked to account for the event, it must know which level switch case to consider. If we successfully elevated a switch case, we choose to reschedule with that higher level switch case. Since the original schedule generated by MSI with the elevated switch case was in the past

and did not undergo changes from this rescheduling, it is possible the schedule will be inconsistent and may lead to complications while scheduling later mandatory activities. An alternative we plan to explore in the future is to disable rescheduling until the switch case is committed. However, this approach would not allow the scheduler to regain time if an activity ended early and caused rescheduling.

## Empirical Analysis

We evaluate the above methods' ability to schedule switch groups on inputs derived from *sol types*, the best available data on expected Mars 2020 rover operations (Jet Propulsion Laboratory 2017a). In order to construct a schedule and simulate plan execution, we use a ground testbed Linux workstation environment, designed to produce the same schedules as the operational scheduler but runs much faster.

Each sol type contains between 20 and 40 activities. Data from the Mars Science Laboratory Mission (Jet Propulsion Laboratory 2017b; Gaines et al. 2016a; 2016b) indicates that activity durations were conservative and completed early by roughly 30% of their nominal duration. However, there is a desire by the mission to operate with a less conservative margin to increase productivity. In our model to determine activity execution durations, we choose from a normal distribution where the mean is 90% of the predicted activity duration. The standard deviation is set so that 10% of activity execution durations will be greater than the nominal duration. For our analysis, if an activity's chosen execution duration is longer than its nominal duration, then it is set to be the nominal duration to avoid many complications which result from activities running long (e.g., an activity may not be scheduled solely because another activity ran late). Detailed discussion of this is the subject of another paper. We do not explicitly change other activity resources such as energy and data volume since they are generally modeled as rates and changing activity durations implicitly changes energy and data volume as well.

We create 10 variants derived from each of 8 sol types by adding one switch group to each set of inputs for a total of 80 variants. The switch group contains three switch cases,  $A_{nominal}$ ,  $A_{2x}$ , and  $A_{4x}$  where  $d_{A_{4x}} = 4 \times d_{A_{nominal}}$  and  $d_{A_{2x}} = 2 \times d_{A_{nominal}}$ .

In order to evaluate the effectiveness of each method, we score how many and what type of activities are able to be scheduled successfully. Each mandatory activity that is successfully scheduled, including whichever switch case activity is scheduled, contributes one point to the *mandatory score*. If the longest, most consumptive switch case,  $A_{4x}$ , is successfully scheduled then it contributes 1 to the *switch group score*. If the second most consumptive switch case,  $A_{2x}$ , is scheduled then it contributes 1/2 to the switch group score. If only the nominal switch case,  $A_{nominal}$ , is able to be scheduled then it does not contribute to the switch case score at all. Since there is only one switch group in each variant, the maximum switch group score for each variant is 1. Scheduling all mandatory activities and one activity from each switch group is much more important than scheduling any number of higher level switch cases. Therefore, the mandatory score is weighted at a significantly higher value

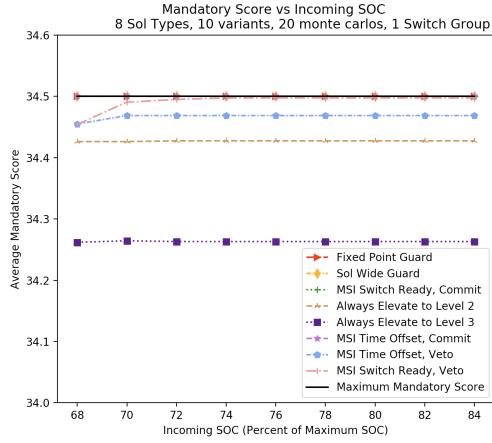


Figure 10: Mandatory score vs Incoming SOC for Various Methods to Schedule Switch Cases.

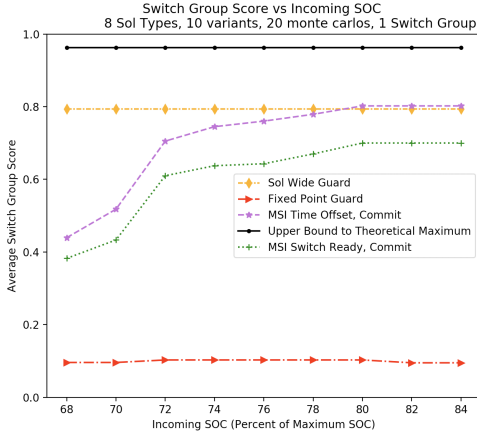


Figure 11: Switch Group Score vs Incoming SOC for Methods which Schedule all Mandatory Activities.

than the switch group score. In the following empirical results, we average the mandatory and switch groups scores over 20 Monte Carlo simulations of execution for each variant.

We compare the different methods to schedule switch cases over varying incoming state of charge values (how much energy exists at the start) and determine which methods result in 1) scheduling all mandatory activities and one switch case from each switch group and 2) the highest switch group score. First, we determine which methods are able to successfully schedule all mandatory activities as well as one switch case activity from each switch group, given by the Maximum Mandatory Score in Figure 10. We only compare switch group scores between methods that successfully achieve the maximum possible mandatory score.

In order to evaluate the ability of each method to schedule all mandatory activities, we also compare against two other methods, one which always elevates to the highest

level, most consumptive switch case while the other always elevates to the medium level switch case. We see in Figure 10 that always elevating to the highest (3rd) level performs the worst and drops approximately 0.25 mandatory activities per sol, or 1 activity per 4 sols on average while always elevating to the second highest level drops close to 0.07 mandatory activities per sol, or 1 activity per 14 sols on average. For comparison, the study described in (Gaines et al. 2016a) showed that approximately 1 mandatory activity was dropped every 90 sols, indicating that both of these heuristics perform poorly.

Both MSI approaches which veto to handle the nominal switch case becoming committed before or during MSI drop mandatory activities. Whenever an activity is vetoed, there is the risk that it will not be scheduled in a future invocation, more so if the sol type is very tightly time constrained, which is true for one of our sol types. As a result, vetoing the nominal switch case can cause the activity to be dropped. The guard methods and the MSI methods that keep the nominal switch case committed and do not try to elevate to a higher level switch case successfully schedule all mandatory activities as well as one switch case per switch group.

The Fixed Point guard, Sol Wide guard, and two of the MSI approaches result in the Maximum Mandatory Score for all incoming SOC values. As shown in Figure 11, the Sol Wide guard and MSI approach using the Time Offset and Commit options result in the highest switch group scores closest to the upper bound for the theoretical maximum. The upper bound for the theoretical maximum switch group score is determined using an *omniscient scheduler* – a scheduler which has prior knowledge of the execution duration for each activity. This scheduler is aware of the amount of resources that will be available to schedule higher level switch cases given how long activities take to execute compared to their predicted duration. The input activity durations fed to this omniscient scheduler are the actual execution durations. We run the omniscient scheduler at most once per switch case. First, we attempt to schedule with only the highest level switch case and if that fails to schedule all mandatory activities, then we attempt with the next level switch case, and so on.

Both MSI approaches have increasing switch group scores with increasing incoming SOC since a higher incoming energy results in more energy to schedule a consumptive switch case during MSI. The less time there is to complete all MSI invocations, the more likely it is for the nominal switch case to become committed. Since we give up trying to elevate switch cases and keep the switch case committed if this occurs, fewer switch cases will be elevated. Because our time offset value is fairly large (15 min), this situation is more likely to occur using the Switch Ready approach to choose when to start MSI, explaining why using Switch Ready results in a lower switch score than Time Offset.

The Fixed Point guard results in a significantly lower switch case score because it checks against a SOC constraint at a particular time regardless of what occurs during execution. Even if the scheduler is attempting to schedule a switch case at a different time than  $T_{Nominal}$  in Figure 3 (e.g., because prior activities ended early), the guard constraint will

still be enforced at  $T_{Nominal}$ . Since we simulate activities ending early, more activities will likely finish by  $T_{Nominal}$ , causing the energy level to fall under the Minimum SOC guard value. Unlike the Fixed Point guard, since the Sol Wide guard checks if there is sufficient energy to schedule a higher level switch case at the time the scheduler is attempting to schedule it, not at a set time, it is better able to consider resources regained from an activity ending early.

We also see that using the Fixed Point guard begins to result in a lower switch group score with higher incoming SOC levels after the incoming SOC is 80% of the Maximum SOC. Energy is more likely to reach the Maximum SOC constraint with a higher incoming SOC. The energy gained by an activity taking less time than predicted will not be able to be used if the resulting energy level would exceed the Maximum SOC. If this occurs, then since the extra energy cannot be used, the energy level may dip below the Minimum SOC guard value in Figure 4 at time  $T_{Nominal}$  while trying to schedule a higher level switch case even if an activity ended sufficiently early.

## Discussion, Related and Future Work

The analysis described in this paper informed the M2020 project decision on switch group implementation, providing: (1) specific prototype implementations of all three approaches (e.g. specific insights into complexity of flight implementation) and (2) specific evidence of relative performance (runtime and schedule quality) of the three approaches. The Mars 2020 project ultimately selected the MSI Time Offset method due to superior schedule quality and moderate complexity implementation and validation.

Just-In-Case Scheduling (Drummond, Bresina, and Swanson 1994) uses a nominal schedule to find likely breaks and produces a branching (tree) schedule to cover execution contingencies. Our approaches all (re)schedule on the fly although the guard methods can be viewed as forcing schedule branches based on time and resource availability.

Kellenbrink and Helber (Kellenbrink and Helber 2015) solve a resource-constrained project scheduling problem where all activities that must be scheduled are not known in advance and the scheduler must decide whether or not to perform certain activities of varying resource consumption. Similarly, our scheduler does not know which of the switch cases to schedule in advance, using runtime resource information to drive (re)scheduling.

Integrated planning and scheduling can also be considered scheduling disjuncts chosen based on prevailing conditions (e.g., (Barták 2000)), but these methods typically search whereas we are too computationally limited to search.

Maillard et al (Maillard et al. 2015) describes onboard satellite execution with ground precomputed energy thresholds to enable lower priority observations when additional energy is available. This is very close to fixed time energy guard when activities are fixed (or nearly so) in time as with satellite overflights, but this is not the case with rovers.

Disjunctive Temporal Problem with Preferences methods (Peintner, Moffitt, and Pollack 2005) solve the time guard problem for upper and lower execution - extending

these methods to concurrent consideration of additional resources is promising future work. Temporal Constraint Satisfaction Problem methods (Shah and Williams 2008) enables disjunctive temporal intervals and could be applied to switch group durations and combining efficient aspects of these methods (Bhargava and Williams 2019) with other resource constraints is also promising future work. Disjunctive temporal intervals are intended to enable multiple non-contiguous execution windows however these reasoning techniques could be adapted to the varying execution duration switch group use case but would need to be combined with resource reasoning techniques (including duration to resource consumption) to apply to switch group analysis (again, an excellent topic for future work).

There are many areas for future work. Currently the time guard heavily limits the placement of activities. Ideally analysis of start time windows and dependencies could determine where an activity could be placed without blocking other mandatory activities. Additionally, in computing the guard for Minimum SOC using the Sol Wide Guard, instead of increasing the guard value by a predetermined fixed amount which could result in over-conservatism, binary search via Monte Carlo analysis could more precisely determine the guard amount. Currently we consider only a single switch group per plan. Additional work is needed to extend to multiple switch groups.

Further exploration of all of the MSI variants is needed. Study of starting MSI invocations if an activity ends early by at least some amount and the switch case is the next activity is planned. We would like to analyze the effects of evenly spacing the MSI invocations in order to avoid relying on throttling and we would like to try disabling rescheduling after MSI is complete until the switch case has been committed and understand if this results in major drawbacks.

We have studied the effects of time and energy on switch cases, and we would like to extend these approaches and analysis to data volume.

## Conclusions

We have presented several algorithms to allow a very computationally limited, non-backtracking scheduler to consider a schedule containing required, or mandatory, activities and sets of activities called switch groups where each activity in such sets differs only by its resource consumption. These algorithms strive to schedule the most preferred, which happens to be the most consumptive, activity possible in the set without dropping any other mandatory activity. First, we discuss two guard methods which use different approaches to reserve enough resources to schedule remaining mandatory activities. We then discuss a third algorithm, MSI, which emulates backtracking by reinvoking the scheduler at most once per level of switch case. We present empirical analysis using input sets of activities derived from data on expected planetary rover operations to show the effects of using each of these methods. These implementations and empirical results have been evaluated in the context of the Mars 2020 onboard scheduler.

## References

- Barták, R. 2000. Conceptual models for combined planning and scheduling. *Electronic Notes in Discrete Mathematics* 4(1).
- Bhargava, N., and Williams, B. C. 2019. Complexity bounds for the controllability of temporal networks with conditions, disjunctions, and uncertainty. *Artificial Intelligence* 271:1–17.
- Chi, W.; Chien, S.; Agrawal, J.; Rabideau, G.; Benowitz, E.; Gaines, D.; Fosse, E.; Kuhn, S.; and Biehl, J. 2018. Embedding a scheduler in execution for a planetary rover. In *ICAPS*.
- Chi, W.; Agrawal, J.; Chien, S.; Fosse, E.; and Guduri, U. 2019. Optimizing parameters for uncertain execution and rescheduling robustness. In *International Conference on Automated Planning and Scheduling (ICAPS 2019)*.
- Chi, W.; Chien, S.; and Agrawal, J. 2019. Scheduling with complex consumptive resources for a planetary rover. In *International Workshop for Planning and Scheduling for Space (IWSPSS 2019)*, 25–33.
- Chien, S. A.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using iterative repair to improve the responsiveness of planning and scheduling. In *Artificial Intelligence Planning and Scheduling*, 300–307.
- Drummond, M.; Bresina, J.; and Swanson, K. 1994. Just-in-case scheduling. In *AAAI*, volume 94, 1098–1104.
- Gaines, D.; Anderson, R.; Doran, G.; Huffman, W.; Justice, H.; Mackey, R.; Rabideau, G.; Vasavada, A.; Verma, V.; Estlin, T.; et al. 2016a. Productivity challenges for mars rover operations. In *Proceedings of 4th Workshop on Planning and Robotics (PlanRob)*, 115–125. London, UK.
- Gaines, D.; Doran, G.; Justice, H.; Rabideau, G.; Schaffer, S.; Verma, V.; Wagstaff, K.; Vasavada, A.; Huffman, W.; Anderson, R.; et al. 2016b. Productivity challenges for mars rover operations: A case study of mars science laboratory operations. Technical report, Technical Report D-97908, Jet Propulsion Laboratory.
- Jet Propulsion Laboratory. 2017a. Mars 2020 rover mission <https://mars.nasa.gov/mars2020/> retrieved 2017-11-13.
- Jet Propulsion Laboratory. 2017b. Mars science laboratory mission <https://mars.nasa.gov/msl/> 2017-11-13.
- Kellenbrink, C., and Helber, S. 2015. Scheduling resource-constrained projects with a flexible project structure. *European Journal of Operational Research* 246(2):379–391.
- Maillard, A.; Verfaillie, G.; Pralet, C.; Jaubert, J.; Sebbag, I.; and Fontanari, F. 2015. Postponing decision-making to deal with resource uncertainty on earth-observation satellites. In *International Workshop on Planning and Scheduling for Space*.
- Peintner, B.; Moffitt, M. D.; and Pollack, M. E. 2005. Solving over-constrained disjunctive temporal problems with preferences. In *ICAPS*, 202–211.
- Rabideau, G., and Benowitz, E. 2017. Prototyping an on-board scheduler for the mars 2020 rover. In *International Workshop on Planning and Scheduling for Space*.
- Shah, J. A., and Williams, B. C. 2008. Fast dynamic scheduling of disjunctive temporal constraint networks through incremental compilation. In *ICAPS*, 322–329.