The Gateway Vehicle Systems Manager Planner

Michael Whitzer1, Laura Barron, Hemanth Koralla, Minh Do, Jeremy Frank, Chuck Fry, Chris Knight, Anthony Koutroulis, Abiola Akanni, J. Benton

[1]Intelligent Systems Division, NASA Ames Research Center
Authors listed in alphabetical order.

Abstract

NASA has developed and tested multiple technologies to enable the autonomous operation of a dormant space habitat. These technologies include a Vehicle System Manager (VSM), integrated with flight software, to control the habitat. We describe the knowledge engineering challenges in developing this planner. These challenges range from the use of a novel domain modeling language with multiple stakeholders, including the planner; the deployment environment, a slow, memory bounded radiation tolerant flight computer; and the high criticality needs of software operating a human spaceflight vehicle. We focus in this paper on how these challenges shaped development of the planner.**MD:** *The abstract sounds like KEPS abstract; maybe can be more general for this "overall report".*

1 Introduction

NASA plans to construct a habitable spacecraft, currently referred to as Gateway (Crusan et al. 2018), in the vicinity of the Moon. Gateway consists of a Habitat, Airlock, Power and Propulsion Element (PPE), and Logistics module. Gateway will support up to 4 crew for 30 days. The PPE will provide orbital maintenance, attitude control, communications with Earth, space-to-space communications, and radio frequency relay capability in support of extravehicular activity (EVA) communications. The Habitat provides habitable volume and short-duration life support functions for crew in cislunar space, docking ports, attach points for external robotics, external science and technology payloads or rendezvous sensors, and accommodations for crew exercise, science/utilization and stowage. The Airlock provides capability to enable astronaut EVAs as well as the potential to accommodate docking of additional elements, observation ports, or a science utilization airlock. A Logistics module will deliver cargo to the Gateway.

The need for autonomy for Gateway is reflected in the Gateway concept of operations (Crusan et al. 2018) and its requirements, as well as requirements for the first of its components, the PPE (NASA 2018). NASA has developed and tested multiple technologies to enable the autonomous operation of a dormant space habitat. These technologies include a Vehicle System Manager (VSM), integrated with flight software, to control the habitat. The work in this paper builds on previous VSM work (Aaseng et al. 2018, 2023; Badger, Strawser, and Claunch 2019) to develop and demonstrate autonomy technology using contemporary flight software and automated reasoning technology.

In this paper, we describe the automated planner we have developed as part of the Gateway VSM. This planner shares heritage with many model-based automated planners developed by the AI planning community, especially temporal planners and planners that reason about resources that have been developed in the last two decades. However, the requirements of human spaceflight and the integration of a planner with spacecraft flight software impose engineering design decisions, external (to the planner) interface requirements, software quality requirements, all of which influence the final product. We focus on these knowledge engineering aspects of the VSM planner and how they influence the current planner design.

In Section 2, we describe the design considerations imposed by the VSM, and Gateway more broadly, that influence the design of the planner. In Section 3, we describe our planning model. In Section 4, we describe different components of our VSM Planner including pre-processing routines, core planning algorithm, heuristic-guided search algorithms, and resource handling routines. In Section 5, we describe the external interfaces between the planner and the rest of the VSM. In Section 6, we describe testing. In Section 7, we describe software engineering decisions that influence planner development and algorithms. In Section 8, we describe related work in developing space-based automated planning applications. We conclude the paper with some of our future work in Section 9.

2 VSM Planner Design Considerations

Due to its role as part of the flight software on a human spaceflight vehicle, the VSM Planner design and implementation is driven by a number of design considerations. Taken together, these constitute a unique set of drivers for the implementation choices we faced in building the planner.

Modeling Language: The Gateway program devised a new modeling language for use by several VSM functions, including the Planner. This language shares many features with temporal and numeric variants of PDDL and other languages in use by academia and industry, but there are some notable differences, as we describe below. Many of the language elements and syntax are driven by the need to comply with the Spacecraft Onboard Interface Services (SOIS) compliant Electronic Data Sheets (SEDS)[1].

Model storage and retrieval: Models are stored in an SQLite database, as opposed to in flat files. Again, the database serves multiple VSM functions, not just as storage for the planner model. The storage of the information in an SQL database, and the specific organization of the data in the database, have implications for the planner design.

Planner interfaces: The Planner has multiple interfaces to the rest of the VSM, and more broadly, the Gateway flight software. The need to retrieve the planner model from the SQLite database is just one such interface; other interfaces impose added design complexity on the planner.

Execution Environment: The Planner, and all the rest of the VSM and Gateway flight software, will be executed in a PowerPC SP0[2] processor, which has significantly limited power and memory compared to desktop computers. This constraint drives many planner design decisions, ranging from the prosaic (memory management, planner interface choices) to strategic (planner algorithm selection).

Safety requirements: Human spaceflight software requires the highest levels of safety and quality assurance. This imposes a variety of software testing and quality control requirements on the planner, which in turn drives choices such as the language of implementation (C), coding standards, and algorithm design to facilitate automated software testing.

3 Planning Model

The input to our VSM planner contains three main parts: **Goals**, **Tasks**, and **Telemetry** (i.e., initial state). The output plan $P = \{\langle a_1, t_1\rangle ....\langle a_n, t_n\rangle\}$ is the list of tasks and their starting times. Tasks are allowed to overlap, and in general, tasks may start at any time (i.e. two tasks can start at the same time). In the first part of this section, we will describe each of those three key components and in the second part we will discuss how they are provided to the planner as a planning problem instance.

Key concepts in our model are:

- **State Variable**: the state of the system is defined by values assigned to state-variables of type discrete (i.e., multi-valued) or numerical.
- **Comparison Constraint**: is a relation $x \diamond y$ in which x is a variable (discrete or numeric), y is a legal value of x, and $\diamond \in \{=, \neq, >, \geq, <, \leq\}$.
- **Variable Assignment**: is an assignment $x \leftarrow y$ in which x is a variable, either discrete or numeric, and y is a legal value of x.
- **Resource**: is a numeric variable r that is changed by task t during its execution, and whose value is bounded above and below.
- **Resource Requirement (ResReq)**: is a relation $R_r = \langle r, v, p\rangle$ in which r is a resource, v is a value of resource that will be used, and $p \in \{RELEASE\_DURATION, RELEASE\_START, RELEASE\_END, RESERVE\_DURATION, RESERVE\_START, RESERVE\_END\}$ is a *resource-use-property* value that specifies how the v amount of resource r is impacted by a given task. Resource impact is additive; reservations reduce the available resource, release increases the available resources. All impacts are instantaneous. Figure 1 illustrates the 6 different types of resource-use-property.
- **Resource Kind**: In our data model, each resource is also labeled with a *resource-kind* value that can be one of the following:
  - ◇ *REUSABLE*: in which tasks can only "reserve" a given resource throughout its duration: acquire v unit at the beginning of the task and release exactly v unit back when the task ends. Thus, if r is a *REUSABLE* resource, then all resource requirements $\langle r, v, p\rangle$ need to satisfy $p = RESERVE\_DURATION$
  - ◇ *CLAIMABLE*: is a REUSABLE resource in which $v = 1$ for all resource requirements $\langle r, v, p\rangle$.
  - ◇ *CONSUMABLE*: in which tasks can only "consume" a given resource. Thus, if r is a CONSUMABLE resource, then all resource requirements $\langle r, v, p\rangle$ need to satisfy $p = RESERVE\_START$ or $p = RESERVE\_END$.
  - ◇ *REGENERATIVE*: in which tasks can both "consume" and "produce" a given resource. In other words, if the resource is not REUSABLE and also not CONSUMABLE then it's resource-kind is REGENERATIVE.

  In the later Section 4.1, we will discuss in more details the relationship between *resource-use-property* and *resource-kind* values and how they are used during planning.
- **Prevailing Constraints:** A *conditional* set of *comparison constraints* that the plan needs to satisfy at all time throughout the planning horizon. They are of the form: $(x_1 = y_1) \wedge ...(x_n = y_n) \implies (p_1 \diamond q_1) \wedge ....(p_m \diamond q_m)$. Thus, if special variables x_i, representing *mode* and *configuration* (ref. Section 3.3), are assigned certain values y_i then certain set of comparison constraints $p_j \diamond q_j$ need to be satisfied by all tasks in the plan. Note that p_j can be either a state variable or a resource variable.

3.1 Goals

Our planner supports both *hard* goals and *soft* goals with preferences. Soft goal value is a single scalar value, either for a (set of) tasks or a (set of) state-variables and comparison constraints. The final plan needs to achieve all hard-goals, while trying to maximize the sum of the preference values of all supported soft-goals. Goals come in two different types:

[1]https://public.ccsds.org/Publications/SOIS.aspx

[2]https://aitechsystems.com/product/sp0-rad-tolerant-3u-compactpci-sbc/

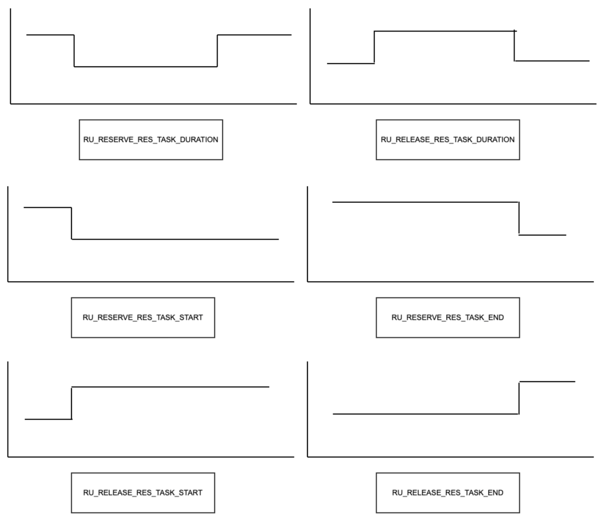
RU_RESERVE_RES_TASK_DURATION
RU_RELEASE_RES_TASK_DURATION
RU_RESERVE_RES_TASK_START
RU_RESERVE_RES_TASK_END
RU_RELEASE_RES_TASK_START
RU_RELEASE_RES_TASK_END

Figure 1: Resource Requirement Explanation **JF:** *consider replacing these with my PPT figs*

- **Task-As-Goal** (TAG): in which the goal g_T consists of a given set of tasks $T = \{a_1 ... a_n\}$ and:
  1. If g_T is a *hard*-goal: then plan P is deemed valid if all tasks a_i in T are part of P;
  2. If g_T is a *soft*-goal with a preference value v_g: then P gains an overall value of v_g if all tasks a_i in T are part of P. If P only contains a subset of T, then it doesn't gain any additional value.
- **Stage-Target Goal** (ST): in which the goal g_S consists of a given set of state-variables p_i and the comparison constraint (discussed earlier in this section) on the desired values q_i for them: $S_g = \{\langle p_1 \diamond q_1 \rangle, ...\langle p_n \diamond q_n \rangle\}$. In this paper, we will use "state-target" to mean a comparison-constraint that appear in a state-target goal. In essence, it's not that different from traditional goals for STRIP planning with the following features:
  1. p_i can be a multi-value discrete variable or a numeric variable.
  2. $\diamond$ can be any comparison operator that is compatible with the variable type of p_i.
  3. If g_S is a hard-goal: the plan P is deemed valid if the final final state S_e achieved after P is done executing satisfies S_g. Thus: $S_e \vDash S_g$.
  4. If g_S is a soft-goal, then g_S is associated with a value v_g and the plan P gains v_g if it achieves all of S_g (i.e. $S_g \subseteq S_e$).

Our planner can be given an arbitrary combination of hard and soft goals of either type TAG or ST as specified above and the main objective function is: (1) satisfies all *hard-goals*; and (2) maximize the accumulated values of *soft-goals* (i.e., $\sum v_g$).

3.2 Tasks

At the high-level, our task model is very similar to the temporal action in PDDL2.1, the standard academic planning domain modeling language. Tasks are durative actions with conditions and effects. Each durative task T is specified by:

- **Prerequisites:** a list of conditions, each of type *comparison constraint*, that need to be true at the start time of the task's execution.
- **Invariants:** a list of conditions, each of type *comparison*, that need to be true throughout the task's execution.
- **Effects:** a list of *variable assignments* that happen at the end time point of the task's execution.
- **Resource Requirements:** a list of *resource requirement* that the task will incur.
- **Duration:** a pair of values $\langle dur_e, dur_w \rangle$ that represents the *estimated* and *worst-case* values for the task's duration. For planning purposes, we use only dur_e.

Discussion: The main differences between PDDL 2.1 and our task models are as follows:

- While *prerequisites* and *invariants* correspond to at-start instantaneous and overall durative conditions in PDDL2.1, our task models do not contain an equivalent of the *at-end* instantaneous condition that's possible in PDDL2.1.
- Our effects can only happen at the task's end time; there is no equivalent to the *at-start* effects available in PDDL2.1.
- Resources and resource-requirements are not explicitly modeled in PDDL while being first-class objects in our modeling framework.
- Task are fully grounded and thus there are no parameters to task models. Duration and resource use properties are constant numbers.

3.3 Telemetry

When the planner starts the planning process, it receives information about the current state of the system through *telemetry* messages coming from cFS apps monitoring different hardware modules. System information received from telemetry makes up the *initial state* of the planner, which consists of the starting values of state variables, as well as resource availability profiles.

In our planning model, the following information from telemetry make up the planner initial state:

- **Variable Assignments**: initial values assigned to all discrete and numeric variables.
- **Resource Profiles**: for each resource r, its profile is specified as: $P_r = \langle Max_r, min_r, A_r \rangle$ in which:
  - $\diamond$ Max_r is the maximum value of r; the planner needs to keep the resource level lower than Max_r throughout the plan.
  - $\diamond$ min_r is the minimum value of r; the planner needs to keep the resource level to be higher than min_r throughout the plan.
  - $\diamond$ $A_r = \{\langle v_1, t_1 \rangle ....\langle v_n, t_n \rangle\}\}$ is a step-function describes the amount of resource that is available for the planner to use over the planning horizon with $t_1 = 0$ specifies the amount of r that is initially available and

$\langle v_i, t_i \rangle$ specifies that v_i is the amount of r available to the planner from time point t_i to t_{i+1}.

- **Mode & Configuration:** Special state variables x_m and x_c describe the system mode and configuration in which the vehicle will operate throughout the planning process; the mode and configuration are retrieved with the other telemetry values. The mode and configuration can be thought of as a pair of variable-assignments of the form $(x_m \leftarrow y_m) \wedge (x_c \leftarrow y_c))$.

Discussion: The set of prevailing constraints, discussed earlier, is of the form $(x_m \leftarrow y_m) \wedge (x_c \leftarrow y_c) \implies (p_1 \diamond q_1) \wedge ....(p_m \diamond q_m)$. Thus, if *mode* x_m and *configuration* x_c are assigned certain values then there is a certain set of comparison constraints that the plan needs to obey throughout its whole execution duration. They are similar to plan-trajectory constraints introduced in PDDL3.0, which is modeled as Linear Temporal Logic (LTL) formulas.

4 Gateway VSM Planner

In this section, we describe the planner. The planner combines goal achievement and scheduling to resolve potential resource conflicts, but we refer to it as a 'planner' for brevity. The Planner interfaces with numerous other cFS applications, several of which are needed to construct the initial planning problem instance; in this section, we focus on describing all the steps in the planning process, alluding to these interfaces only when necessary. A full description of these interfaces is provided in Section 5.

4.1 Planning Model Building

The Onboard Data Model (ODM) is an SQLite database containing an extensive list of variables and tasks that capture the whole VSM system configuration, and all possible potential goals that the planner may have to plan for. Obviously, for a particular planning scenario with a particular set of goals and a concrete initial condition, large part of the ODM is either not needed or being irrelevant. Therefore, like many existing academic STRIPS planners, the VSM Planner also runs a pre-processing step to narrow down the set of tasks, goals, and the associated variables to make the planning model as compact as possible. Specifically, we first load the ODM from the SQLite database, after which we run three routines in sequence:

- *Conflict Analysis*: to pre-detect and store information about conflicting pairs of tasks. This helps speedup the subsequent phase of planning search where task conflicts lead to *task-orderings* and *causal-link threats*.
- *Relevance Analysis*: that finds the tasks/variable-assignments that are "relevant" to a particular set of goals;
- *Reachability Analysis* that finds the tasks/variable-assignment that are "reachable" from the initial state.

All of those routines, or variations of them, are commonly used by existing academic planners.

4.1.1 Before Receiving Goals

The planner is running in a mode of perpetually waiting to receive goals from the *Fault Manager* VSM application, and then find the plan to satisfy them. In this section, we will describe the routines invoked upon the planner's starting up and before going into the main waiting loop.

Loading Tasks from ODM: The entire task library is read in from the ODM at application startup. Task IDs, names, durations, and status are read in the first pass. Task consequences (effects), prerequisite and invariant conditions, and resource requirements are read in subsequent passes. Finally, variable metadata is loaded for all variables referenced by task conditions, consequences, and resource requirements. Internally, each state variable stores a list of consequences affecting that variable. Consequences in turn have back links to their tasks. This organization facilitates search and pre-search analyses.

Conflict Analysis: Task conflict analysis, which identifies effect-effect (i.e., two task effects changing the same variable to different values) and effect-condition (i.e., one task effect violates the other task's prerequisite/invariant) conflicts , is performed immediately after the task library is loaded. Task conflicts are stored in a map keyed by task ID pairs.

4.1.2 When Receiving Goals

Upon receipt of a plan-request, which includes the goal information:

- Task status (enabled/disabled) is refreshed and the set of tasks pre-loaded from the ODM is refreshed, filtering out from consideration disabled tasks.
- Detailed information about goals G in the plan-request are read from the ODM.
- Prevailing constraints, based on the mode (i.e., mission phase) and configuration of the assembled spacecraft, are also read from the ODM at this time.

Relevance Analysis: once the planner receives the goal set G from the *Fault Manager* cFS app, this routine looks for tasks, variables, and resources that are *relevant* to G. Relevant tasks are tasks that can be used to either achieving the goals directly or support the achievements or the conditions that enable tasks that achieving goals. Specifically, let T_R and C_R be the relevant set of tasks and comparison-constraints. A sketch of the Relevance Analysis is as follows:

1. *Initialize*: T_R with the set of Task-As-Goal (TAG) and C_R with the union of State-Target goals.
2. *Relevant Tasks Update*: add to T_R any task with effect that supports a comparison-constraint in C_R.
3. *Relevant Comparison-Constraints Update*: add to C_R all conditions (i.e., prerequisites and invariants) of all (new) tasks in T_R

Step 2 and 3 are run alternatively until a fixed-point is reached (i.e., no change to T_R and C_R). The final sets T_R and C_R contains all tasks and comparison-constraints relevant to the set of TAG and state-target goals given to the planner. Also note that in those two steps, any task (or goal) that violate prevailing constraints are disqualified.

Reachability Analysis: while Relevance Analysis can be thought of as "backward" reasoning from the goals to find relevant tasks and variable-assignments, Reachability Analysis runs in the opposite "forward" direction and find tasks and variable-assignments that can be *reached* from the initial state through executing a set of tasks that are applicable. A sketch of the Reachability Analysis:

1. *Initialize*: the reachable set of variable-assignments S_I with the set of variable-assignments in the initial state and the reachable set of tasks $T_I = \emptyset$.
2. *Reachable Tasks Update*: add to T_I any task that has its conditions fully supported by S_I.
3. *Reachable Variable-assignment Update*: add to S_I the effects of all tasks that are newly added to T_I.

Like Relevance Analysis, we also run the two updating steps 2 and 3 until a fixed-point (i.e. no change to S_I and T_I). The final sets S_I and T_I contains all variable-assignments and tasks that can be achieved with the given initial-state that the planner operate from.

Combined Algorithm: Running Relevance Analysis followed by Reachability analysis yields the following:

1. Run Relevance Analysis for a given set of goals and return T_R and C_R.
2. Run reachability analysis from the initial-state:
   (a) *Reachable Task Update*: only consider as candidate tasks that are in T_R when building T_I.
   (b) *Reachable Variable-assignment Update*: only consider adding task effects that support C_R to S_I.

Any task that is not in the final set T_I and any variable-assignment that is not in S_I can be removed from the planning model. If there is a TAG T_g that is not in T_I or a state-target goal ST_g that is not in S_I then:

- If T_g or ST_g is a *soft*-goal, it's removed from the planning model.
- If T_g or ST_g is a *hard*-goal, then we can declare the planning problem unsolvable.

NOTE: when running relevance-analysis, we also filter out any task from consideration if it violates prevailing constraint. Thus if either (1) one of its effects conflict with a prevailing constraint; or (2) one of its condition contradicts a prevailing constraint; or (3) its resource requirement violates a prevailing constraint. We also remove any state-target that contradicts a prevailing constraint or any TAG that violates a prevailing constraint. We don't need to enforce prevailing constraint during reachability analysis because it's run after relevance-analysis and by that time all tasks & goals violating prevailing constraints are already filtered out.

Resource Type Inference: As outlined in Section 3, each resource R is classified as one of 4 different *resource-kind*: *REUSABLE, CLAIMABLE, REGENERATIVE*, and *CONSUMABLE*. The Planner uses these types during planning, as we describe below in section 4.6. Also as outlined in Section 3, each task can have a resource-requirement on R of one of the 6 different *resource-use-properties* outlined in Figure 1. However, there is no guardrail in the ODM to ensure that the *collective* resource requirements by all tasks are consistent with the resource-kind declaration for each resource. Furthermore, it's possible that relevance and reachability can reduce the set of tasks, and allow or force the planner to treat resources differently, based on the set of tasks used in the problem instance. Therefore, the pre-processing phase includes a routine to *infer* the resource-kind, if needed, to be consistent with how resources are utilized by the final set of relevant & reachable tasks.

	REL_S	REL_D	REL_E	RES_S	RES_D	RES_E	RES_KIND
1	Y	*	*	*	*	*	REGEN
2	*	Y	*	*	*	*	REGEN
3	*	*	Y	*	*	*	REGEN
4	*	*	*	Y	Y	*	REGEN
5	*	*	*	*	Y	Y	REGEN
6	N	N	N	N	Y	N	REUSE
7	N	N	N	Y	N	*	CONSUM
8	N	N	N	*	N	Y	CONSUM

Table 1: Inferring Resource-Kind from Resource-Use-Properties by resource-requirements of relevant and reachable tasks.

Specifically, let T be the set of tasks collected from the ODM after running the combined filtering algorithm (relevant + reachability analysis). For each resource r, we first collect the set of tasks $T_r \subseteq T$ that require r. Then we apply the rules outlined in Table 1 to all tasks in T_r to infer the correct resource-kind for r. Specifically:

- *Line 1-3*: If there is at least one task $t \in T_r$ that has a resource-requirement on r with type RELEASE_START, RELEASE_END, or RELEASE_DURATION, then r is classified as REGENERATIVE.
- *Line 4-5*: If there are at least two tasks $t_1, t_2 \in T_r$ in which t_1 requires r with type RESERVE_DURATION and t_2 requires r with type either RESERVE_START or RESERVE_END, then r is classified as REGENERATIVE.
- *Line 6*: If all tasks $t \in T_r$ uniformly requires r with type RESERVE_DURATION, then r is classified as REUSABLE. If all requirements are for exactly 1 unit of r then we classify r as CLAIMABLE.
- *Line 7-8*: If all tasks $t \in T_r$ requires r with type either RESERVE_START or RESERVE_END, then r is classified as CONSUMABLE.

Algorithm 1: Gateway VSM Planner as cFS Application

```
1: $T_{all} \leftarrow$ read all tasks from the ODM using SQL query
2: Conduct ConflictAnalysis to store all conflicting task pairs.
3:
4: loop
5:     Receive plan-request $PR$ from FaultManager
6:     Call PreProcessing(PR)
7:     Call RelevanceAnalysis(G)
8:     Call AcquireInitialValues($V_{rel}, R_{rel}$)
9:     Call ReachabilityAnalysis($S_{init}, T_{rel}$)
10:    Call BuildFinalPlanningModel
11:    Call POCLPlanningAlgorithm
12:    Send the plan to Dispatcher to execute
13:    Ask GoalTracker to track goal-achievement
14: end loop
15:
16: procedure PreProcessing(PR)
17:     Filter out from $T_{all}$ all tasks currently disabled
18:     $G \leftarrow$ retrieve from ODM using goal IDs in $PR$
19:     Retrieve prevailing constraints from ODM using mode & configuration in $PR$
20: end procedure
21:
22: procedure RelevantAnalysis($G$)
23:     $T_{rel} \leftarrow$ tasks in $T_{all}$ that are relevant to $G$
24:     $V_{rel} \leftarrow$ variables appear in conditions of $T_{rel}$
25:     $R_{rel} \leftarrow$ resources utilized by tasks in $T_{rel}$
26: end procedure
27:
28: procedure AcquireInitialValues($V, R$)
29:     $S_{init} \leftarrow$ values for $V$ from the StateDetermination cFS app
30:     $RP_{init} \leftarrow$ resource profiles for $R$ from the ResourceManager cFS app
31: end procedure
32:
33: procedure ReachabilityAnalysis($S, T$)
34:     $T_{rec} \leftarrow$ tasks in $T$ that are reachable from $S$
35:     $S_{rec} \leftarrow$ effects of $T_{rec}$
36: end procedure
37:
38: procedure BuildFinalPlanningModel
39:     $V_{final} \leftarrow$ variables appear in both $V_{rel}$ and $S_{rec}$
40:     $T_{final} \leftarrow T_{rec}$
41:     $R_{final} \leftarrow$ resources used by $T_{final}$
42:     $G_{final} \leftarrow$ goals supported by tasks in $T_{rec}$
43:     Run ResourceTypeInference.
44: end procedure
```

4.2 Planner Setup

Algorithm 1 captures the key steps described in this section. Specifically, when the Planner app is starting up, it will go through the step of loading in all tasks from the ODM (line 1) and conduct *Conflict Analysis* to pre-build the set of all pairs of tasks that conflict with each other (line 2).

The planner then enters the perpetual mode of waiting for the *plan request* from *Fault Manager*. Upon receiving such a request (line 5), it will start the process of building the planning model, going through several key steps:

1. Call the *pre-processing* routine (line 16-20) to first check for the current task-status flags and filter out from considerable all tasks that are currently disabled (line 17), retrieve the goal information (line 18), and the set of *prevailing constraints* (line 19) from the ODM.
2. With the concrete set of goals and tasks, call the *Relevance Analysis* (line 22-26) to find all tasks, variable, and resources relevant to the goals.
3. The planner will retrieve the initial values for the set of relevant variable and resources, acquired from the last step, from other cFS apps (line 28-31): (1) *State Determination* for the current variable values; and (2) *Resource Manager* for the resource allocation profiles.
4. With the initial values for all relevant variables, it will then run *Reachability Analysis* (line 33-36) to narrow down the set of tasks relevant to goals to contain only ones that are also reachable from the initial state.
5. The final planning model is built (line 38-44) that includes tasks & goals that are both relevant and reachable, and the variables and resources that are utilized by those tasks and goals.

After the planning-model is built, the planner will try to find a valid plan using the POCL planning algorithm (ref. Section 4.3) and when found will send the plan to the *Dispatcher* cFS app for execution (line 12). At the same time the planner will also ask the *Goal Tracker* cFS app to track the achievements of the given goals (line 13). After that, the planner returns to the idle loop waiting for the new plan-request.

4.3 Partial Order Causal-link Planning

Our Gateway Planner utilizes the Partial Order Causal-link Planning (POCL) planning algorithm, one of the mainstays in the planning research community. At the high-level, the POCL algorithm starts with an empty plan and gradually extends it “backward” from the goals by adding support for goals and conditions of tasks in the partial-plan as follow:

1. Selects an un-supported goal g and try to establish a causal-link support from another task T_g's effect e to g, and resolve potential conflicts between T_g and other tasks in the (partial) plan.
2. If task T_g is a newly added task, then add all conditions of T_g to the list of unsupported-goals.
3. *Termination*: Repeat those two steps until there is no goal that is not supported.

For the rest of this section, we will elaborate on the implementation of the POCL algorithm in our planner. Given that our goal representation (ref. Section 3.1) is much more complex than a typical STRIPS planning, the treatment of resource as a first class object, and other modeling differences, there are some adjustments from the traditional POCL algorithm needed for it to work in our domain.

4.3.1 Plan Components

We will start with some key concepts:

Action: In our planner, we use a separate data-structure action to represent an instance of a task in a partial or complete plan. An action is represented as a tuple: $a = \langle t_a, st_a, O_a \rangle$ in which:

- t_a is the task that a represents.
- st_a is the time-point represent the start time of a that will be managed by a temporal network.
- O_a is the list of of open-conditions for a; thus, conditions (i.e., prerequisites + invariants) of a that are not supported by any other action in the current plan.

For the rest of this section, we will use the terminologies of action's prerequisite/invariant/effect to refer to the same entity belonging to the task that it encapsulates. We will also create a special task t_{init} to represent the initial-state with: (i) no prerequisite or invariant; (ii) have all variable-assignments making up the initial state as its effects; (iii) $duration(t_{init}) = 0$; and an action $a_{init} = \langle t_{init}, st_{init} = 0, O_{init} = \emptyset \rangle$.

Causal Link: $a_1 \overset{e}{\underset{c}{\rightarrow}} a_2$ indicates that a variable assignment e is an effect of task t_{a_1} and is used to support the condition c of task t_{a_2}. In essence, the variable-assignment of e satisfies the comparison-constraint in c.

State-Target Support: $a \overset{e}{\rightarrow} \langle x \diamond y \rangle$ in which the variable-assignment of the effect e of action a satisfies the comparison-constraint of the state-target $\langle x \diamond y \rangle$, which is part of some goal (ref. Section 3.1)[3].

Task Conflict: tasks may have conflicting effects (i.e., assign different values to the same variable) or one's effect may conflict with another task's prerequisite/invariant (i.e., variable-assignment of a task effect violate the comparison-constraint of the other task's condition).

Action Ordering: When tasks represented by two actions conflict, we need to establish an ordering between them. There are 5 different types of ordering between a pair of actions $\langle a_1, a_2 \rangle$:

1. $a_1 \prec^{e_1}_{e_2} a_2$: effect e_1 of a_1 happens before effect e_2 of a_2.
2. $a_1 \prec^{e_1}_{p_2} a_2$: effect e_1 of a_1 happens before prerequisite p_2 of a_2.
3. $a_1 \succ^{e_1}_{p_2} a_2$: effect e_1 of a_1 happens after prerequisite p_2 of a_2.
4. $a_1 \prec^{e_1}_{i_2} a_2$: effect e_1 of a_1 happens before invariant i_2 of a_2.
5. $a_1 \succ^{e_1}_{i_2} a_2$: effect e_1 of a_1 happens after invariant i_2 of a_2.

Causal-link Threat: when a causal-link $cl = a_1 \xrightarrow[c_2]{e_1} a_2$ is established and there is another action a_3 that has an effect e' that violate the condition c_2 then e_3 is considered a "threat" to cl. To *resolve* this threat, we need to establish one of the two action-orderings: $a_3 \prec^{e_1}_{e_3} a_1$ or $a_3 \succ^{e_3}_{c_2} a_2$. Establishing one of these orderings ensures that e_3 happen "before" or "after" the causal-link cl. Note that if e_3 is in conflict with e_1 but doesn't violate c_2 then it's not considered a threat[4].

Temporal Network: goal-supports, causal-links, threats, and action-orderings lead to complex set of temporal relations between different actions. We use a *Simple Temporal Network* (STN) to manage those temporal relations (Dechter, Meiri, and Pearl 1991). There are two main components of the STN: (1) set of time-points t_i; and (2) the temporal constraints between those time-points of the form: $t_i - t_j \leq d$. Constraint propagation routines that runs in polynomial time allow the STN to compute the lower/upper bound value for each time-point quickly, and detect temporal inconsistency. In our planner, most of the time-points in our STN represents the start-times of actions in the plan and the constraints represent the temporal constraints between actions due to causal-links, action-orderings, and causal-link threats. Specifically, let st_a and dur_a be the start-time and duration of action a, and $et_a = st_a + dur_a$ be the end-time of a. With time-points in the STN representing st_a, we have:

- *Causal-link* $a_1 \overset{e}{\underset{c}{\rightarrow}} a_2$: implies that the end-time of a_1 should be before the start time of a_2, thus the temporal constraint: $st_{a_1} + dur_{a_1} < st_{a_2}$. In STN constraint form: $st_{a_1} - st_{a_2} < -dur_{a_1}$.
- *Action-ordering* $a_1 \prec^{e_1}_{e_2} a_2$: implies the end time of a_1 should be before the end time of a_2, thus the temporal constraint: $st_{a_1} + dur_{a_1} < st_{a_2} + dur_{a_2}$. In STN constraint form: $st_{a_1} - st_{a_2} < dur_{a_2} - dur_{a_1}$.
- *Action-ordering* $a_1 \prec^{e_1}_{p_2} a_2$ or $a_1 \prec^{e_1}_{i_2} a_2$: the two constraints have the same implication and thus incur the same STN constraint as the causal-link.
- *Action-ordering* $a_1 \succ^{e_1}_{p_2} a_2$: implies that the end time of a_1 should be after the start time of a_2, thus the temporal constraint $st_{a_1} + dur_{a_1} > st_{a_2}$. In STN constraint form: $st_{a_2} - st_{a_1} < dur_{a_1}$.
- *Action-ordering* $a_1 \succ^{e_1}_{i_2} a_2$: implies that the end time of a_1 should be after the end time of a_2 (where the

[3]In POCL algorithm for academic STRIPS planning, goal-support and causal-link for action's conditions are treated the same because they are both represented by comparison-constraint: $x = y$ and both are "hard" constraints that need to be satisfied. However, our goals are considerably more complex (ref. Section 3.1), leading us to differentiate between those two entities.

[4]A concrete example: $c_2 : v > 5$, $e_1 : v \leftarrow 7$, and $e_3 : v \leftarrow 10$

invariant condition ends), thus the temporal constraint: $st_{a_1} + dur_{a_1} > st_{a_2} + dur_{a_2}$. In STN constraint form: $st_{a_2} - st_{a_1} < dur_{a_1} - dur_{a_2}$.

- *Causal-link Threat*: as explained earlier, if action a_3 violate a causal-link $cl = a_1 \xrightarrow[c]{e} a_2$ then the violating effect, happens at the end-time of a_3, needs to happen either: (1) before the start of cl, which is the end-time of a_1, or (2) after the end of cl, which is either: (i) the start-time of a_2 if c is its prerequisite; or (ii) the end-time of a_2 if c is its invariant. Thus, causal-link thread leads to the temporal constraint:
  - ◇ $(st_{a_3} + dur_{a_3} < st_{a_1} + dur_{a_1}) \vee (st_{a_3} + dur_{a_3} > st_{a_2})$ if the causal-link supports a prerequisite.
  - ◇ $(st_{a_3} + dur_{a_3} < st_{a_1} + dur_{a_1}) \vee (st_{a_3} + dur_{a_3} > st_{a_2} + dur_{a_2})$ if the causal-link supports an invariant.

  Given that the STN doesn't support disjunctive constraints (i.e., "or" relations), we will elaborate in the later part of this section on how the planning algorithm "branches" over the disjunctive relations to break it into two different separate choices in handling each causal-link threat.

4.3.2 The Algorithm

As outlined at the beginning of this section, the POCL planning algorithm, which we utilizes in our VSM Planner, starts from an initial "empty" plan and extend it until we reach a complete plan. While doing so, it navigates a space of partial-plans, using one of the graph-search algorithms (ref. Section 4.4) guided by heuristics (ref. Section 4.5) to choose the most promising partial (i.e., incomplete) plan, represented as a "search-node", to extend. We will first explain how the partial-plan is represented using the key concepts described above:

Partial Plan: P consists of:

- A_P: a set of actions.
- GS_P: a set of state-target goal-support in P.
- CL_P: a list of causal-links established in P.
- AO_P: a list of ordering between conflicting actions.
- stn_P: a consistent STN with time-points representing start-time of actions in A_P and constraints representing causal-links and action-orderings[5].

Initial Partial Plan: the initial partial-plan P_{init} to start the POCL search process is constructed with:

- $A_{P_{init}} = \{a_{init}, a \in G^h_{TAG}\}$ with a_{init} is the action representing the initial state (discussed earlier in this section), and G^h_{TAG} is the set of "hard-constraint" task-as-goals;
- $GS_{P_{init}} = \emptyset$;
- $CL_{P_{init}} = \emptyset$;
- $AO_{P_{init}} = \emptyset$;
- $stn_{P_{init}}$ contains a single constraint $st_{a_{init}} = 0$.

[5]Note that there can be multiple causal-links and/or action-orderings between the same pair of actions that all map into a single STN constraint. For example, multiple causal-links and $a_1 \prec^{e_1}_{p_2} a_2$ orderings lead to a single temporal constraints $et_{a_1} < st_{a_2}$

Complete Plan: is a plan P in which (i) all "hard" goals (TAG and state-target) are supported and (ii) all actions in P have all of their conditions supported.

Algorithm 2: VSM Gateway POCL Planning Algorithm

```
1:  Input: $G = G^h_{ST} \bigcup G^s_{ST} \bigcup G^s_{TAG}$
2:         $S = \{P_{init}\}$
3:  Output: a complete plan or no-plan
4:
5:  while $S \neq \emptyset$ do
6:      Heuristically pick the "best" partial-plan $P$ from $S$
7:      if $P$ is a complete plan then
8:          Return $P$ and terminate
9:      else
10:         Extend $P$ to get a set of child-plans $S_P$
11:         Add $S_P$ to $S$
12:     end if
13: end while
14: Return no-plan.
```

Algorithm 2 outlines the high-level POCL algorithm: starting with the set S of partial-plan consisting of a single initial-partial-plan P_{init}, it will:

- Heuristically pick a partial-plan P;
- Expand P and generate "child-plans" of P by finding potential supporters for the un-supported goals or action's conditions in P;
- The generated child-plans are then added back to S.

The procedure stops when a complete plan is picked from S or S is empty, in that case the planner signal that there is no plan for the input problem.

Algorithm 3 describes the key procedures in Algorithm 2 on how to generate child-plans of a given partial plan P, resulted in the set S_P that will be added to the collection of partial-plans for Algorithm 2 to choose from in the next iteration. This is done through the process of adding support for either (1) an unsatisfied goal (TAG or ST) or (2) an un-supported condition of an action already in the current plan. This resulted in three different procedures:

Add "Soft" Task-As-Goal (TAG) (Line 5-10): the first type of goal that can be supported is the TAG, in which the goal specifies certain task needs to be in the plan. While tasks captured by mandatory "hard" TAGs are included in the initial plan P_{init}, supporting tasks for "soft" TAGs are optionally added to the set of partial-plan for the planning search algorithm to consider. Specifically, when a given partial plan P is expanded, for each of the unsupported TAGs t, the expansion routine will add one new child-plan of P in which a new action a representing t is added to P.

```
Algorithm 3: Generate Child-plans
1: Input: partial-plan $P = \langle A, GS, CL, AO, stn \rangle$
2: Output: set of child plans $S_P$.
3: Initialization: $S_P \leftarrow \emptyset$
4:
5: procedure ADD NEW "SOFT" TASK-AS-GOAL
6:     for all $t \in G^s_{TAG}$ s.t. no $a \in A$ represents $t$ do
7:         Add new action $a$ represents $t$ to $A$
8:         Add resulting new partial-plan to $S_P$
9:     end for
10: end procedure
11:
12: procedure ADD NEW STATE-TARGET SUPPORT
13:     for all state-target $g \notin GS$ do
14:         for all task $t$ that has effect $e$ satisfies $g$ do
15:             if $\nexists a \in A$ represents $t$ then
16:                 Add new action $a$ represents $t$ to $A$
17:             end if
18:             Add new goal-support $a \xrightarrow{e} g$ to $CL$
19:             for all $a_i \in A$ with effect $e_1$ violating $g$ do
20:                 Add ordering $a_i \prec^{e_1}_{e} a$ to $AO$
21:                 Add constraint $et_{a_i} < et_a$ to $stn$
22:             end for
23:             if $stn$ is consistent then
24:                 Add resulting new partial-plan to $S_P$
25:             end if
26:         end for
27:     end for
28: end procedure
29:
30: procedure ADD NEW CAUSAL-LINK
31:     for all action $a \in A$ do
32:         for all un-supported condition $c$ of $a$ do
33:             for all task $t$ that has effect $e$ satisfies $c$ do
34:                 if $\nexists a' \in A$ represents $t$ then
35:                     Add new action $a'$ represents $t$ to $A$
36:                 end if
37:                 Add new causal-link $a' \xrightarrow[c]{e} a$ to $CL$
38:                 for all $a_i \in A$ with $e_1$ violating $c$ do
39:                     procedure PROMOTION
40:                         Add ordering $a' \prec^{e_1}_{c} a$ to $AO$
41:                         Add constraint $et_{a'} < st_a$ to $stn$
42:                         if stn is inconsistent then
43:                             discard child-plan
44:                         end if
45:                     end procedure
46:                     procedure DEMOTION
47:                         Add ordering $a' \succ^{e_1}_{c} a$ to $AO$
48:                         if $c$ is prerequisite then
49:                             Add $et_{a'} > st_a$ to $stn$
50:                         else                    ▷ ($c$ is invariant)
51:                             Add $et_{a'} > et_a$ to $stn$
52:                         end if
53:                         if stn is inconsistent then
54:                             discard child-plan
55:                         end if
56:                     end procedure
57:                 end for
58:                 Add resulting child-plans to $S_P$
59:             end for
60:         end for
61:     end for
62: end procedure
```

If there are k unsupported TAGs in P, then there are k new partial-plans that are children of P generated and added to S_P

Add State-Target Support (Line 12-28): the next type of support is for state-targets. Note that while each state-target *goal* can have multiple state-targets, each of which is a comparison constraint $\langle x \diamond y \rangle$ (ref. Section 3.1), each new child node generated by this procedure only try to support a single state-target. For each of the unsatisfied state-target g, we collect all tasks t in the domain that can support it (i.e., has an effect that can satisfies g). If an action a representing t is not already in P then we create a new action and add it to the current partial-plan[6]. For each new support $a \xrightarrow{e} g$, we then need to protect it from any "threat" of another action's effect violating g. Thus, for each action a_i with effect e_1 violating g, we need to ensure that e_1 happens before e. If the temporal constraint corresponding to the resulting ordering constraint $a_i \prec^{e_1}_{e} a$ lead to an STN inconsistency[7], then we discard this potential partial-plan resulting from $a \xrightarrow{e} g$. If all threat resolutions lead to a consistent STN, then we add the new partial-plan (with added new goal support and threat-resolution ordering constraints) to S_P.

Add Causal-Link Support (Line 30-62): adding support to an "open" (i.e., unsupported) condition of an action a in P is very similar to adding support to a state-target, given that both of them are represented as comparison constraints. However, there are several differences:

- Threat Resolution: unlike state-target support protection in which the violating effect needs to come before the st-supporting effect, in causal-link $a' \xrightarrow[c]{e} a$ protection, the threatening effect e_1 can come *before* or *after* the causal-link. This leads to two different procedure to handle threat: (1) PROMOTION (lines 39-45) in which e_1 is ordered before e; and (2) DEMOTION (lines 46-56) in which e_1 is ordered after c.
- Prerequisite vs Invariant: there are two types of condition: prerequisite and invariant. Support for prerequisites only needs to be satisfied at the start-time of a, while support for invariants needs to be maintained throughout the duration of a. Thus, for each threat, different temporal constraints, either $et_{a'} > st_a$ (line 49) or $et_{a'} > et_a$ (line 51) is added to the STN depending on whether c is a prerequisite or an invariant.
- Goal Preference: state-target is part of ST goal that have associated preference value while conditions are not.

[6]In our planning domain, we currently assume that each task can appear at most *once* in the final plan. Thus, if an action a representing t is already in P, we do not consider the possibility of adding another action a' also representing t to P. Not allowing multiple actions capturing the same task in a complete plan reduce the number of potential child nodes; in the future, if our domain requires plans having multiple instances of the same task in the valid plans, then we will adjust our algorithm accordingly.

[7]Since our requirement is that goals need to be satisfied at the end of the plan, we do not consider goal threat resolution in which e_1 happens after the goal support is established.

Therefore, to calculate the accumulated goal preference, we prefer to keep st-supports separated from causal-link supports.

Plan Output: Algorithm 2 returns a complete plan P with actions and various temporal constraints on their starting times due to goal-orderings, causal-links, action-orderings etc, all captured in the temporal network stn_P. The cFS application on the receiving end, named *Timeline Executioner*, on the other hand, expects to receive a plan in the form of $P_{te} = \{\langle t_1, st_1\rangle....\langle t_n, st_n\rangle\}$, that is, a set of tasks t_i with the associated fixed start-time st_i. Since all action's starting times are managed in stn_P and the Simple Temporal Network stores all possible *earliest* possible time for each timepoint managed, we can simply for each action a_i in P, extract the task t_i captured in a_i and the $earliest(st_{a_i})$ value from stn_P and send them to the *Timeline Executioner*.

Discussion: there are several prominent planning approaches to find plan, among them: Forward State-Space (FSS), Backward State-Space (BSS), Partial-Order Causal-Link (POCL), or compilation approaches (to SAT, CSP, MILP). Our evolving complex action and goal models, combined with strict deployment constraints of the flight computer, make it hard to find a suitable substrate to compile into, and thus rule out the compilation approach. While FSS has dominated planning competitions for the simpler form of planning (e.g., classical or simple temporal), we decided that POCL, the more flexible approach, is better to deal with the complex temporal and resource constraints that arise in our domain (e.g., durative tasks, resource profiles, exogeneous events such as eclipse-windows, planning horizon) and complex goal setting. By explicitly having a STN as part of each partial-plan, we can immediately deal with the temporal constraints in our planning model and to prepare for our future extensions to handle additional temporal and resource constraints from other Core-Flight Software Applications interacting with the planner. As we discuss in Section 4.6, some resources can also be handled directly at goal achievement time, while we defer other resource handling to a secondary scheduler.

4.4 Graph Search Algorithm

As described in the previous section, our POCL planning algorithm starts from the "root" P_{init} partial-plan and continue to expand it by generating "child" plans that have additional supports for goals and unsupported action conditions. There are different ways to manage and search among the set of partial-plans that are created using Algorithm 3 and we will elaborate on which search algorithm is used in our planner to navigate the set of partial-plans, until a complete plan is found.

Figure 2 shows one concrete example illustrating how partial-plans are generated and navigated by the planning search algorithm. Starting with the P_{init} as the lone partial plan, which is called a "root" node in search terminology, assuming that P_{init} is not already a complete plan then Algorithm 2 (line 9) will call Algorithm 3 to expand P_{init} and generate the set of child-plans $\{P_1, P_2, P_3, P_4\}$ and add them to the set of generated partial-plans S. In the second iteration of Algorithm 2, let's assume that the planner chooses P_1 from S and again calls Algorithm 3 to generate the child-plans $\{P_{11}, P_{12}, P_{13}, P_{14}\}$ and again add them to S. At any given moment, partial plans in S are those that are generated but has not been visited by the search algorithm and are called "leaf" nodes.

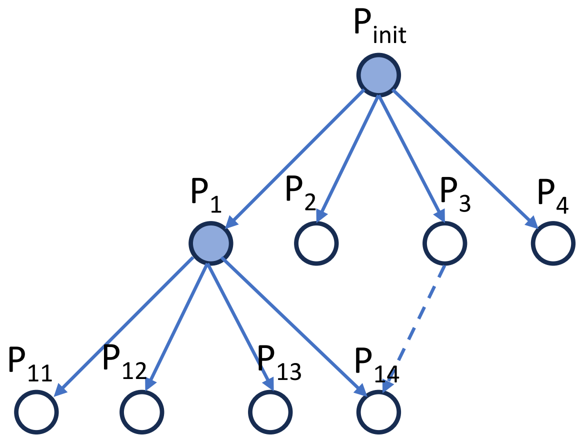

Figure 2: Planning Search Graph Example

Best-First Search: It's essential to have a good heuristic to guide Algorithm 2 (line 5) to pick out the "best" partial-plan from S to expand. Our problem is a constrained optimization problem; we both want to find "good" plans that satisfy many soft goals (as described in Section 3.1), and we want to quickly satisfy the hard goals. Balancing these competing drivers of the heuristic is a challenge. Section 4.5 explains how we implement such heuristics in our planner. In general, since we use the heuristic to pick the best partial-plan to expand next, our high-level search algorithm falls into the category of *Best-First Search* graph search. Several popular algorithms that are in this category are:

- *Breadth-First Search*: "best" is defined as "closest to root-node". Thus, the breadth-first algorithm would visit P_1, P_2, P_3, P_4 before P_{11}, P_{12}, P_{13}, P_{14}.
- *Depth-First Search*: "best" is defined as "furthest from root-node". Thus, depth-first algorithm would visit P_{11} and then its children before going back to visit nodes at its level (e.g., P_{12}) and the nodes at the highest level such as P_2 are only visited after all the nodes under P_1 are visited by Algorithm 2.
- A^*: "best" is defined as a combination of distance to the root-node and the estimated distance to the closest complete (optimal) plan.

We use a variant of Best-First Search with the heuristic function deciding the "best" node explained in Section 4.5. To quickly pick out the best partial-plan for each iteration, the set S of generated partial-plans is normally sorted according to a given heuristic function used to rate them and thus normally implemented as a *priority queue*. Therefore, in this paper we will occasionally refer to S as a *search queue*.

Anytime Search: Algorithm 2 is designed to stop when it finds a complete plan. Ideally, the heuristic function would ensure that first complete plan is also the highest-value complete plan. However, in practice, such heuristics are either very resource-intensive to compute, or requires the search algorithm to visit a large number of nodes before finding the optimal plan. Therefore, most heuristics in use, especially for planning problems with complex resource and temporal constraints, are geared to find the first non-optimal complete plan of balancing between plan quality and search time.

To compensate for the heuristic quality, we implemented an *anytime* search algorithm. We provide a search-time limit t^s_{limit}; if the planner finds a complete plan before t^s_{limit} is over, then we will keep running Algorithm 2, as long as there are still not-visited search nodes, for potentially finding better solution. Thus, our algorithm may find multiple solutions with gradually better plan quality, and at "anytime" $t_{stop} \leq t^s_{limit}$, we can return the best found solution at t_{stop}. For example, the first complete plan P_1 that it returns at $t_1 < t^s_{limit}$ may satisfies all hard-goals and none of the "soft" optional goal that bring additional value. If we keep searching after t_1 we can find plans with better quality by satisfying a higher-value subset of "soft" goals.

Duplicate Detection: a given partial-plan can be generated in different ways. For example, if there are two state-targets g_1 and g_2 that can be supported by two different actions a_1 and a_2. Using our example shown in Figure 2, starting from P_{init}, two different child-plans are generated: (1) P_1 with a new state-target support $sg_1 = a_1 \rightarrow g_1$; and (2) P_3 with a new state-target support $gs_2 = a_2 \rightarrow g_2$. If we pick P_1 to expand, among its children assume that P_{14} represent the partial-plan where gs_2 is added to P_1, it's easy to see that P_{14} is also the child-plan of P_3 if gs_1 is added to it.

Duplicate copies of the same partial-plans generated in different paths can lead to increased planning time by repeatedly visiting and expanding the same partial-plan, and increased memory usage in holding larger number of generated partial-plans. However, given the complexity of our plan representation, duplicate-detection can incur significant cost of checking for plan equality for each newly generated partial-plan against every plan currently stored in the queue. In our planner, duplicate-detection helps reduce planning time significantly in most of the larger test cases.

Since comparing all plan's components for equality checking is time-consuming, we want to compare each newly generated plan with only a small number of partial-plans currently in the search queue. Therefore, for each partial-plan P we generate a numerical *key* value k_P with the following properties: (i) k_P can be generated quickly; (ii) if $k_{P_1} \neq k_{P_2}$ then $P_1 \neq P_2$; (iii) if $k_{P_1} = k_{P_2}$, the probability that $P_1 = P_2$ is high. Specifically, the formula computing k_P takes into account task-orderings, causal-link, goal-supports, and actions in P.

When a partial-plan is generated, the duplicate-detection routine performs binary search in the table storing the sorted key values to find partial-plans with the same key values. If any such plans with the same key exist, a more thorough plan-comparison routine will be performed to detect duplicates.

4.5 Heuristics

Heuristics are crucial in planning algorithms, providing estimates to guide the search towards the goal state efficiently. This section focuses on the heuristics we have implemented to guide our search, including a planning graph-based heuristic. The heuristic sums the number of expected actions required to achieve each open condition, leveraging the structured representation of the planning graph to assess progress towards a complete plan. In the following, we discuss the planning graph-based heuristic we implemented and follow with a discussion of our tie-breaking strategy.

4.5.1 Unsupported Open Conditions

MD: *I suggest we start with higher-ranked heuristic first. This is not our top-ranked heuristic at the moment: hard-goals, then goal-priority related.* One approach often used to guide search in partial order planning is to focus on satisfying the highest number of open conditions (Gerevini and Schubert 1996), generating heuristics that focus on that metric. Intuitively, number of open conditions in a partial plan offers a possible measure of the number of search steps that remain. We define this heuristic as $h_{OC}(P)$. In essence, measuring open conditions assumes that each open condition will provide one step toward a complete plan. As it turns out, this heuristic can sometimes lead to poor search guidance in some scenarios. For example, in domains where some tasks and goals require more search steps to reach than others, each unsatisfied open condition may require a varying number of steps. Further, in domains where positive interactions exist, e.g., when one task satisfies more than one open condition, this heuristic may overestimate the number of tasks required.

4.5.2 Priority-based Heuristics

While the number of search steps offers a view into how far a search state is from the goal, another metric to looking solely at the number of steps (via the open conditions), we also can look at total goal priority and the potential goal priority

4.5.3 Planning Graph-based Heuristic

We first rank the partial plans using a planning graph-based heuristic similar to that developed by Nguyen & Kambhampati (2001). Like their heuristic, we generate a planning graph based on the current planning problem as part of our heuristic initialization. The planning graph structure has the advantage that it contains the possible prerequisites and causal links that we may have in each partial plan.

As usual with planning graph-based heuristics, we begin by building a planning graph from the initial state of the problem, following the graph building step of the Graph-Plan algorithm (Blum and Furst 1995) without accounting for mutual exclusion information. Though our planning domain uses multi-value variables where a variable x takes an a valid assignment y (as discussed in Section 3), planning graph data structures typically work by identifying whether a certain "fact" is given by an action or task. Hence, we view all variable-value pairs as literals and interpret the task prerequisites, invariants, and effects as such. Currently, our planning graph heuristic ignores temporal and resource aspects of the problem.

The data structure is used to represent the possible literals and tasks within a planning problem. It consists of alternating layers of literal and task nodes, starting with the initial state at its base layer and progressing from there. Each literal layer contains all possible states that can be reached from the initial state given the actions taken up to that point, while each task layer contains all tasks that could be executed in the proceeding literal layer. The graph expands forward until a literal layer includes the goal state, or until it becomes evident that no plan can achieve the goal. While we discuss the graph as one consisting as many layers, in practice we can implement it using a labeled bipartite graph. Each task or literal is labeled with the layer in which it would first appear in the expanded graph. We can then use the data structure we built to lookup the number of tasks required to reach a given literal.

More formally, Algorithm 4 operates by iteratively building a graph that alternates between task layers and literal layers. Initially, the graph starts with the initial literals at layer 0. For each subsequent layer i, the algorithm introduces a new task layer T_i containing tasks that are applicable in the preceding literal layer L_{i-1}. Each task in T_i is connected to its preconditions in L_{i-1} and its effects in the newly formed literal layer L_i. This process continues iteratively until the graph reaches a state where all goals are present**MD:** *"Goals" in our planning model involves both state-target and TAGs, both can be either "hard" or "soft" constraint with different priority values. So probably need to elaborate a little more on the propagation and stopping criteria.*, or no new literals can be added. The resulting bipartite graph captures the potential causal links between tasks and literals across different planning steps, providing a structure that gives an estimate of the number of steps required reach each literal (and therefore each open condition).

The planning graph data structure has the advantage that we can propagate each literal or task whatever information we would like. In this case, we will use it to analyze partial plans.[8] **JF:** *Is it also worth saying we can build the plan graph once and use it at any time in search, i.e. don't need to propagate it when new partial plans built? But then disadvantage is that it's a big thing in memory? and if we do say these things, where is it best to say them?*

JB: *Need to check with notation with rest of paper and fix formatting. Draft algorithm writeup, so also want to check for mistakes.* Using the bipartite planning graph constructed through Algorithm 4, we can analyze the cost of achieving a set of open conditions in a partial plan. For each open condition $l \in OC$, where OC is the set of open conditions, we determine the earliest layer i in which l appears in the planning graph PG using a lookup on the graph. The layer number i estimates the minimum number of steps required to achieve the literal l starting from the initial state, considering the actions (or tasks) and their supporting actions defined in PG. By aggregating the layer numbers for all open conditions in OC, we obtain a heuristic estimate of the cost to reach all open conditions from the initial state. Specifically, let $lev(l)$ be the layer number of the literal l, then the the maximum layer number among all open conditions, $\max_{l \in OC} \text{lev}(l)$, provides a lower bound on the number of steps required to achieve the entire set OC concurrently. We can also use the sum of all layers where the open conditions appear, $\sum_{l \in \text{OC}} \text{lev}(l)$, giving the effort required to achieve all open conditions concurrently. It is this latter approach that we use.**MD:** *We separate the open action conditions from unsupported state-target goals (since unsupported goals can be hard or soft constraint), so are we only summing level information for open conditions or also for state-target goals? Do we differentiate "hard" and "soft" goals? What are about TAGs?*

[8] It's also possible to propagate other information through the planning graph, including soft goal value; these are topics we plan to explore later.

Algorithm 4: Build Planning Graph with Bipartite Structure

```
1: Input: Initial literals in $P_{init}$, Tasks $T$, Goals $G$
2: Output: Bipartite planning graph $PG$
3: Initialization: $PG \leftarrow \emptyset$, Add $L_0$ to $PG$ with label 0
4: $i \leftarrow 0$                    ▷ $i$ tracks the current layer number
5: while not all goals in $G$ are in $PG$ OR new literals are added in the last iteration do
6:     $i \leftarrow i + 1$
7:     procedure EXPAND TASK LAYER
8:         for all tasks $t$ applicable in the literal set of $PG$ do
9:             if all preconditions of $t$ are in $PG$ with label $i - 1$ then
10:                Add $t$ to $PG$ with label $i$
11:            end if
12:        end for
13:    end procedure
14:    procedure EXPAND LITERAL LAYER
15:        for all tasks $t$ in $PG$ with label $i$ do
16:            for all effects $e$ of $t$ do
17:                if $e$ not in $PG$ then
18:                    Add $e$ to $PG$ with label $i + 1$
19:                end if
20:            end for
21:        end for
22:        $i \leftarrow i + 1$
23:    end procedure
24: end while
25: return $PG$
```

4.5.4 Tie-breaking Heuristics

Our planning graph heuristic serves as part of a suite of heuristics used to rank child nodes. At present, it acts as the part of a tie-breaking scheme where initially the number of mandatory goals left unsatisfied serves as the primary ranking. When the algorithm generates child nodes, we assign a heuristic value based on the total cost calculated from the planning graph. In instances where multiple child nodes share the same heuristic value, a tie-breaking mechanism is employed that relies on additional heuristic metrics. First, we consider the soft goal values of goals satisfied in the child node; nodes satisfying higher-value goals are preferred. If a tie still exists, we then compare the number of unsatisfied open conditions, favoring nodes with fewer remaining conditions. Finally, if the nodes remain indistinguishable, we evaluate them based on the number of actions in the partial plan, giving preference to nodes with a smaller number of actions.

4.6 Resource Handling

As described in Section 3, there are different resource types in our planning model: *REUSABLE* (with *CLAIMABLE* a special case of), *CONSUMABLE*, *REGENERATIVE*. The current version of the planner handles *CONSUMABLE* and *REUSABLE* resource types; for the rest of this section we will describe how we handle resource constraints for each of those two types.

4.6.1 Consumable Resource Handling

A resource r is of kind *CONSUMABLE* if all tasks using r can only "consume" it, thus, all resource requirements $\langle r, v, p\rangle$ satisfies $p = RESERVE\_START$ or $p = RESERVE\_END$ (ref. Section 3). In other words, all tasks t using r will consume an amount of v either at its start-time st_t (if $p = RESERVE\_START$) or end-time et_t (if $p = RESERVE\_END$).

Currently, for consumable resources r, the resource profile $P_r = \{Max_r, min_r, A_r\}$ (ref. Section 3.3) for all consumable resources are constant, in the sense that: $A_r = \{\langle v_1, t_1 = 0\rangle\}$ has a single value with $min_r \leq v_1 \leq Max_r$. For simplicity, we will assume for the rest of the section that $min_r = 0$ and for each consumable resource r we denote V_r as the constant allowable value of r throughout the planning horizon[9]. This "flat-line" profiles allow us to use a rather simple resource constraint enforcement during the planning search process.

Algorithm 5 describes our procedure, which is called when a partial-plan P is created by adding a new action a to its parent partial-plan. Note that if P is created by adding a state-target support or a causal-link using an existing action in its parent plan, there is no need to check for consumable resource violation in P. For each resource r consumed by a:

- First, we compute the total consumption T_r of r by all actions in P, including a (line 4-7).
- If T_r is higher than the value V_r allowed by the resource profile of r, then return *Violate* (line 8-10).
- Finally, we check T_r against all *prevailing constraints* (ref. Section 3.3) involving r. Return *violated* if T_r doesn't satisfy any of those prevailing constraint (line 11-15).

[9]From the planner's reasoning point of view, if $min_r > 0$, we can transform P_r to: $P'_r = \{Max_r - min_r, 0, \{\langle v_1 - min_r, t_1 = 0\rangle\}\}$.

Algorithm 5: Consumable Resource Handling

```
 1: Input: partial-plan $P$ where new action $a$ is added
 2: Output: Satisfy/Violate
 3: for all resource-requirement $\langle r, v, p\rangle$ of $a$ do
 4:     Initialize the total consumption of $r$: $T_r \leftarrow 0$
 5:     for all $a_i \in P$ consuming $r$ with $\langle r, v_i, p\rangle \in a_i$ do
 6:         $T_r \leftarrow T_r + v_i$
 7:     end for
 8:     if $T_r > V_r$ then
 9:         Return Violate
10:     end if
11:     for all prevailing constraint $c$ involving $r$ do
12:         if $T_r$ violates $c$ then
13:             Return Violate
14:         end if
15:     end for
16: end for
17: Return Satisfy
```

4.6.2 Reusable Resource Handling

A resource r is of kind *REUSABLE* if all tasks using r can only "use" it during the task, then return it when the task is complete; thus, all resource requirements $\langle r, v, p\rangle$ satisfies $p = RESERVE\_DURATION$ (ref. Section 3). In other words, all tasks t using r will reserve an amount of v at its start-time st_t and release the same amount v at the end-time et_t of t.

Unlike *CONSUMABLE* resources, for which the total resource consumption T_r is independent of the ordering relations between different actions in the plan, overall usage of reusable-resource r depends on exactly how actions using r are ordered in the plan P. Given the fact that actions in our partial-plan are only partially ordered against each other, there are many ways in which those actions can overlap. Combined with an arbitrary resource profile P_r then checking for a reusable-resource violation in P can be quite costly[10]. Therefore, in our current implementation, reusable resource constraint enforcement is done as a *post-processing* greedy scheduling step after a complete plan satisfying all other constraints. This is similar to strategy of handling planning (causal) and scheduling (resource) in two separate phases in the RealPlan planner (Srivastava, Kambhampati, and Do 2001).

Algorithm 6 shows the key steps in our greedy algorithm to fit reusable resource usages by all actions in the plan within the allocation profiles. We do so by moving "forward" starting from the plan-start, fixing any encountered violation, one at a time (lines 11-19) by repeatedly calling the procedure FIXVIOLATIONS(r,P_r) (lines 21-39) for each resource r and the set of tasks P_r in the plan that utilizes r. Note that fixing resource violations for a resource r_1 by moving tasks in P_{r_1} around may cause additional violations

[10]This involves checking for all possible sets of starting times for all actions in P satisfying the current goal-supports, causal-links, action-orderings, and other temporal constraints in the stn of P if all such set would lead to the violation of the profile P_r of r.

```
Algorithm 6: Reusable Resource Handling
 1: $Input$: A complete-plan $P$
 2:         Profiles: $\langle Max_r, min_r, \{\langle v_1^r, t_1^r\rangle ... \langle v_n^r, t_n^r\rangle\}\rangle$
 3: $Output$: A revised plan with no resource violation
 4:
 5: $u_a^r$: amount of $r$ used by action $a$
 6: $U_A^r$: total amount of $r$ used by set of actions $A$
 7: $P_r^t \subseteq P_r$: actions in $P_r$ using $r$ at time $t$
 8: $P_r^{>t} \subseteq P_r$: actions in $P_r$ using $r$ after time $t$
 9: $c_r^t$: earliest time after time $t$ that profile of $r$ changes
10: $M_t^r$: amount of $r$ allocated at time $t$
11: repeat
12:     for all reusable resource $r$ do
13:         $P_r \subseteq P$: set of actions using $r$
14:         HASVIOLATION ← FIXVIOLATION($r, P_r$)
15:         if HASVIOLATION = TRUE then
16:             Break out of FOR loop
17:         end if
18:     end for
19: until HASVIOLATION = FALSE
20:
21: procedure FIXVIOLATION($r, P_r$)
22:     HASVIOLATION ← FALSE
23:     repeat
24:         $t \leftarrow$ PLANSTART
25:         if $U_{P_r^t}^r > M_t^r$ then
26:             HASVIOLATION ← TRUE
27:             MOVEACTION($P_r^t, M_t^r, c_r^t$)
28:             $t \leftarrow \min_{a \in P_r^t} st_a$
29:         else
30:             if $P_r^{>t} \neq \emptyset$ then
31:                 $t \leftarrow \min_{a \in P_r^{>t}} st_a$
32:             else
33:                 $t \leftarrow \infty$
34:             end if
35:         end if
36:         $t \leftarrow min(t, c_r^t)$
37:     until $t$ reaches PLANEND
38:     Return HASVIOLATION
39: end procedure
40:
41: procedure MOVEACTION($A, M, t_{pc}$)
42:     for all $a_i \in A$ do
43:         EARLIESTEND ← $\min_{a_j \in A \wedge a_j \neq a_i} et_{a_j}$
44:         NEWSTART$_{a_i}$ ← $min$(EARLIESTEND, $t_{pc}$)
45:         MOVEDISTANCE$_{a_i}$ ← NEWSTART$_{a_i} - st_{a_i}$
46:     end for
47:     $A^\star \leftarrow \{a \in A : U_{A \setminus a}^r \leq M\}$
48:     if $A^\star \neq \emptyset$ then
49:         $A_{min\_dis} \leftarrow \{a_i : \min_{a_i \in A^\star}$ MOVEDISTANCE$_{a_i}$ $\}$
50:         $a^\star \leftarrow a_i : \min_{a_i \in A_{min\_dis}} u_{a_i}^r$
51:     else
52:         $A_{min\_dis} \leftarrow \{a_i : \min_{a_i \in A}$ MOVEDISTANCE$_{a_i}$ $\}$
53:         $a^\star \leftarrow a_i : \max_{a_i \in A_{min\_dis}} u_{a_i}^r$
54:     end if
55:     $st_{a^\star} \leftarrow$ NEWSTART$_{a^\star}$
56:     if Detect Tempocal Network Inconsistency then
57:         Exit with FAILURE
58:     end if
59: end procedure
```

for another resources r_2 because of (1) overlapping between P_{r_1} and P_{r_2} and also (2) due to the complex temporal relationship between different actions in the plan (moving an action a_1 in P_{r_1} that supports another action a_2 in P_{r_2} will move a_2 too). Therefore, we need to repeat the main loop (line 11-19) whenever a violation has been detected and fixed (line 15) until there is no violation for any resource.

The FIXVIOLATIONS(r,P_r) procedure works as follows:

- Start "scanning" from time t = PLANSTART, increasing t to either (1) the (earliest allowed by the *stn*) start-time of the next action in the plan that use resource r, or (2) the next time when the amount in the allocation profile of r changes. Thus, move to the next time-point t where either the *total-usage* or *allowed amount* of r changes.
- At time t:
  - ◇ if the total-usage is higher than the allocation amount (line 25), then we need to *reduce* the resource contention by moving one action in the contending set to later time point, this is done by calling the procedure MOVEACTION (line 51-59). After moving, we reset t to be the earliest start time among actions that originally caused the violation (which may still exist since moving one action may not be enough to resolve the violation).
  - ◇ else if the total-usage is within the allowed amount, then we simply move to the next start-time after t (line 29-35) or the next time that the profile changes (line 36), whatever is earlier.

The MOVEACTION procedure that "move" one action to a later start-time uses a couple of greedy heuristics:

- For each action a_i in the contending set A, if we move a then we will move its start-time to be either: (1) after the earliest end-time of another task $a_j \in A$, thus totally ordering a_i and a_j (line 43); or (2) the time where the allocation profile change, since the allocation profile may increase and thus "solve" the contention. Whichever is earlier between the two options (line 44). The potential change in the start-time of a_i is then logged (line 45).
- Since we prefer a "minimum-change" from the original plan P, our tiered heuristics are:
  - ◇ Identify the set $A^\star$ of tasks (line 47) such that if we move one task in this set, it would resolve the contention.
  - ◇ Among tasks in $A^\star$, then prioritize the tasks that "move" by the smallest distance (line 49).
  - ◇ Break ties by moving task that use the least amount of resource (line 50), thus lowering the chance that it will cause a resource violation at the new time that it is moved to.
  - ◇ If set $A^\star$ is empty (line 51), which means that no single task that can be moved and solve the resource violation at t, we will need to gradually move multiple tasks and we will do it one at a time. We still chose to move the task with the shortest distance (line 52). Unlike the case where $A^* \neq \emptyset$ (line 48), we break ties by prefer

tasks that use the most amount of the resource (line 53) – thus reducing the contention by the highest amount, and lowering the chance that we need to move a higher number of tasks to fix the violation.

- When an action's start-time is moved to a new (earliest-possible) value, then we add an appropriate temporal constraint to the temporal network stn. If it leads to a temporal inconsistency, we exit the resource-handling post-processing program with failure and ask the planner to continue to search for a new plan.

After all resource contentions are solved, let t_a be the earliest possible value for time-point st_a managed by the temporal network $stn(P)$. Executing each action $a \in P$ at t_a will satisfy all causal, resource, and temporal constraints. Thus, the action sequence $\{\langle a_1, t_1\rangle .... \langle a_n, t_n\rangle\}$ is a valid fixed-time plan to send to the *Dispatcher* cFS application (ref. Figure 3).

Discussion: Our post-processing algorithm runs greedily in polynomial time and doesn't guarantee completeness.

5 The Planner's Interface to Other Applications

In order to create plans the *Planner* must interface with a number of other VSM applications. These applications include *SQLite Interface* (SQLITE_IF), *Fault Manager*, *Resource Manager*, *State Determination*, *Dispatcher*, and *Goal Tracker*:

- S*QLITE_IF*: As mentioned previously, the *Gateway Onboard Data Model* (ODM) stores all of the onboard data required for the operation of the vehicle. VSM applications utilize the SQLITE_IF to retrieve and write data to the database.
- *Fault Manager* (FM): The FM app is the entity in charge of determining if a fault response is necessary. FM is the application in the middle between *State Determination* (SD) and *Planner*. FM is the gatekeeper to determining if safing and recovery responses are required. It's also currently responsible to send the plan-request to the planner, which trigger the new planning process.
- *Resource Manager* (RM): The RM app is specifically responsible for calculating and reporting current state, projected state, and trending information for a subset of the resources such as power, bandwidth, and storage capacity. These reports and calculations, resulted in the resource profiles sent to the planner, aid in plan creation and execution, operator awareness, and fault management.
- *State Determination* (SD): The SD app monitors telemetry from all over the vehicle in the form of state variables. Some state variables are derived through calculations in SD. Using the current values of state variables, plus additional predictions requested from the *Resource Manager*, the *Planner* can set up the resource conditions and projections at the beginning of the new plan.

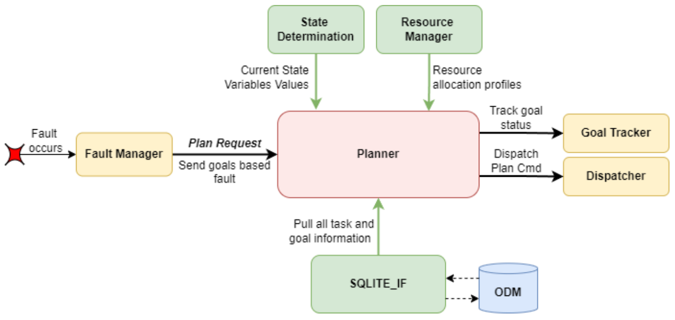

Figure 3: Planner interfaces to other VSM applications.

- *Dispatcher*: The *Dispatcher* acts as the plan executor which involves issuing task start requests as dictated by the plan. The *Dispatcher* also tracks the status of all planned tasks and generates plan reports.
- *Goal Tracker*: The *Goal Tracker* accepts internal VSM commands from the *Planner* to add a single goal or a set of goals to the active goal list as an output of planning. *Goal Tracker* maintains the active goal list as input and will log goal success and failure to a file as well as report goal status in cyclic telemetry.

Figure 3 describes these interfaces between the *Planner* and the other apps. We explain this figure, referring back to Algorithm 1 in Section 4, in the remainder of this section. Upon start up, *SQLITE_IF* is used to load all task data from the ODM (line 1). In the event that a fault occurs, *FM* may initiate a recovery process by sending a plan-request with recovery goals to the Planner after Gateway has achieved a safe state. When the *Planner* receives these goals from *FM* (Algorithm 1, line 5), *SQLITE_IF* is used to filter out disabled tasks and retrieve goal data from the *ODM* (line 6) and performs Relevance Analysis (line 7). During this analysis tasks are filtered based on their relevance to achieve the goals to recover the vehicle. The *Planner* then requests all current state variable values from the *SD* app and projections from the *RM* app (line 8), after which it performs Reachability Analysis (line 8). With this data, the *Planner* has all of the initial conditions needed to initiate the planning algorithm (line 9). The green arrows in Figure 3 represent data inputs to the *Planner* for the problem setup. Once a plan is found the *Planner* transmits the plan to the *Dispatcher* (line 12) and *Goal Tracker* (line 13) apps, which will execute and monitor the active plan.

6 VSM Planner Testing

Due to its high criticality role as part of the Gateway VSM, the Planner must be tested extensively to ensure there are no defects. Defects can arise in a number of different parts of the planner; inside any of the numerous constituent planner's algorithms (e.g. search or heuristics), in the interfaces between the Planner and other parts of the VSM, poorly specified problem instances due to model defects, and so on. Testing methodologies designed to quickly identify these errors are critical to ensuring a quality software product.

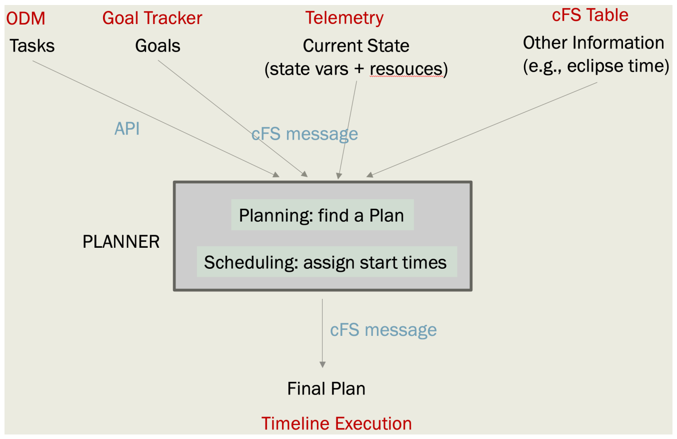

Figure 4: Planner running in the Online Mode.

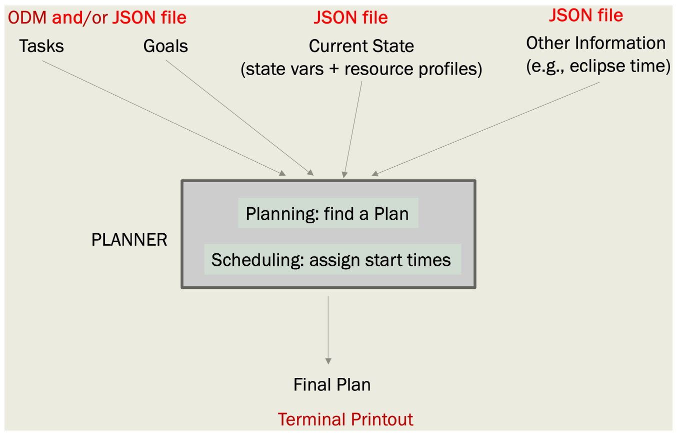

Figure 5: Planner running in the Offline Mode.

Algorithm 1 outlines our planner running in a continuous closed-loop interacting with other cFS apps. While this is how the planner is deployed, it is difficult to effectively test solely on the deployment architecture: (1) the ODM and other cFS apps need to be correctly configured and loaded for the planner to run. These are complex procedures involving many stakeholders and complicated computer system setup. In addition, (2) The ODM contents (the planning model and goals) are configuration managed, with multiple VSM stakeholders depending on its contents. Testing scenarios are limited to what is currently available in the current stable version of the ODM. Since the ODM is only updated infrequently while the planner is constantly changing, limiting testing to the configuration managed ODM would significantly impact planner development.

Therefore, we also developed an additional "offline" method to test the planner that allow us to: (1) test the planner without compiling or running other cFS apps providing its input such as Fault Manager, State Determination, or Resource Manager; (2) test any planning scenarios that may involve tasks/goals/variables that currently are not in the ODM, or test variations that are slightly different from how they are currently defined in the ODM. This "offline" test mode supports rapid development and alow us to work on future-looking features of the planner.

We provide more detail on these test methods and associated infrastructure below:

Online: in which the planner interacts with other Core Flight Software (cFS) application that communicate with each other through cFS messaging channels, as described in Figure 3. In this mode, which steps are captured in Algorithm 1 with the details of the interaction with other cFS apps described in Section 5:

- *Tasks*: are read from the *Onboard Data Model* (ODM), which is a SQL Lite database.
- *Goals*: the goal IDs are sent to the planner from the *Fault Manager* cFS app as part of the *plan request*. The planner then extracts the goal specification from the ODM using the received goal IDs.
- *Initial State:* the initial system state, which specifies the values of different state variables relevant to the planning problem, is received as cFS messages from the *State Determination* application.
- *Resource Profiles:* the availability and min/max values for each resource that the planner needs to obey throughout the *planning horizon* are given to the planner from the *Resource Manager* cFS app.

Offline: To support offline testing, we developed a modeling framework based on JSON that mimics all planner input described in Section 3.

For *Tasks* and *Goals* that the *online* setting extracts from the ODM, in the *offline* mode the planner can use either from the ODM or the JSON input. Specifically:

- The planner combines the set of tasks and goals read from the ODM and JSON file, if both are specified as input sources.
- If there is a task or goal with ID exists in both the ODM and the JSON input file, then the task or goal version from the JSON file is used.

All other input to the planner that make up the initial state, that in the "online" mode received from the *State Determination* and *Resource Manager* cFS apps, are also specified in JSON format.

Figure 4 and 5 show the input and output information flow diagram of our planner running in the online and offline modes.

ODM Checker:**JF:** *Not sure this is right place for ODM Checker but putting it here for now* Plan domain modeling for AI planners is a challenge, even for experts. The VSM Planner domain modeling language is rich and complex, and will be used by spacecraft systems engineers and mission operations staff who may not be familiar with this style of plan domain modeling. For example, modelers can make errors in parts of a task model, such as forgetting preconditions or effects, or constructing tasks with conflicting preconditions or effects which trivially can't be inserted into a plan. As a more complex example, as we described in Section 4.6, modelers could incorrectly identify the 'type' of a resource, given all of the tasks and their resource impacts. A final challenge is the translation between abstract model elements (tasks, goals, resources) and the ODM SQLite representation.

These factors led to the development of an ODM Checker. This checker is configurable with a variety of rules, and reads an ODM instance and reports on violations of those rules. These, in turn, can be used to help find problems with either the models that drive planner development, or translations between the language and the ODM SLQLite instance or schema that can cause the planner to fail. This checker complements our ability to test the planners' inputs prior to 'scenario-level' testing using the ODM itself.

7 Software Design Requirements

Human spaceflight software requires the highest levels of safety and quality assurance. This imposes a variety of software testing and quality control requirements on the planner, which in turn drives choices such as the language of implementation (C), coding standards, and algorithm design to facilitate automated software testing. We describe the impact of these criteria on Planner software design.

7.1 Planner-ODM Interaction

Gateway VSM has a number of components that rely on complex vehicle configuration information stored in the Onboard Data Model (ODM) and this model is stored in a SQLite database and model information is retrieved via the SQLite C API. In order to manage concurrent data access, this functionality is contained within a singular Core Flight Software application called 'SQLITE_IF'. As the planner logic is heavily-reliant on the ODM for goal and task definition, the VSM development team integrated the planner into the SQLITE_IF application and the planner can quickly down-select task and goal information as needed using the SQL language.

7.2 Memory Management

Gateway VSM runs as part of the flight software load on the HaLO module. The SP0 flight computer is a COTS single board computer with significant spaceflight heritage, based on an embedded system-on-a-chip introduced in 2005. It has one 32-bit CPU core. Its performance and memory capacity are at least an order of magnitude less than contemporary PC hardware. The COTS real-time operating system runs entirely in physical memory, and doesn't support virtual memory paging. The file system is loaded at startup from flash storage, and resides in main memory during execution.

The Core Flight Software system (McComas, Wilmot, and Cudmore 2016) and spacecraft flight software standards impose additional constraints[11]. Stack space is statically allocated at system startup. Recursion is prohibited. Applications can allocate system memory only at startup. Real-time routines may not use dynamic memory allocation. Non-real-time applications, such as the planner, may use a standard cFS dynamic allocator library, but have to reserve the allocation region statically, or at startup. Memory management strategies are informed by object lifetimes. Domain data is read in at startup from the ODM database, and persists with only modest updates throughout the application's lifetime. Search memory usage is dynamic and ephemeral. At the end of each planning run, search memory is completely reset, so that each subsequent run starts from a clean slate. Fixed size objects are allocated from statically sized arrays. List and set representations use fixed size "buckets", also allocated from static arrays, with simple, low-overhead allocation policies. A simple bump allocator suffices for ODM text data; search does no string manipulation.

[11]See CFE User's Guide, 1.6.16 Memory Pool

These memory management policies have implications for several elements of the Planner design; the search algorithms, the Simple Temporal Network (STN) maintained during search, and anytime search performance, which we describe below.

The planner's search algorithm iterates over a priority queue (i.e. no recursion). Its STN propagation algorithms are incremental and iterative. The STN algorithms are derived from the NASA open source EUROPA constraint based reasoning system (Frank and Jónsson 2003). The partial plan representation incorporates the STN for that plan. The simple greedy scheduler is also incremental and iterative. Notably, neither algorithm uses backtracking (a choice described in Section 4.4), and there is no need for retraction of constraints during either the planner or scheduler search.

The planner's search algorithm necessarily generates many equivalent search states, as is common in 'best-first' type search. Duplication detection and removal has proven to be a classic space-time tradeoff. In the current approach, each partial plan is retained in a hash table when it is enqueued. New partial plans are checked against the entire search history prior to being enqueued. Duplicate states are dropped. The result has been a significant reduction in the number of steps required for some sample problems, but search history storage dominates the memory pool for larger problems, causing memory exhaustion after a few hundred search steps.

Variable size structures in the plan representation and the STN are dynamically allocated from a 1 MiB pool. This has proven sufficient for planning problems requiring up to hundreds of search steps. STN representations have been optimized for the relatively small problem sizes. 2-byte array indices are used in place of 4-byte pointers. This imposes an upper bound of 64k nodes and 64k edges, which is more than adequate for the problems the planner will be required to solve. A node in the STN occupies only 32 bytes – the size of an L1 cache line on the flight CPU – and an edge only 16 bytes. Nodes and edges are stored in contiguous, homogenous arrays. The result is a very compact representation with a minimum of pointer chasing. The plan representation has also been tuned for space efficiency, though there remains room for improvement.

Search termination criteria and anytime search are influenced by available memory when it runs out. Search is terminated at that point. If at least one complete plan has been generated, the best plan is returned.

7.3 Coding Standards and Software Quality Assurance

VSM has coding standards and test requirements covering syntax, symbol naming, coverage testing, and functional testing. The planner team implemented a number of these specifically for planner code to ensure integration into the flight software went with minimal effort. Tools leveraged include GitLab CI[12] jobs to ensure code quality include formatting (yamllint[13], clang-format[14], pyfmt[15], shfmt[16]) and code static analysis (flawfinder[17], shellcheck[18], and internally-developed database and code checking tools.) Functional testing includes the cFS Test Framework (CTF)[19], cFS unit tests, and a standalone test framework. Critical to support development was the ability to monitor code quality specifically on changed code so that code quality would improve incrementally and would reduce the effort to meet code compliance later in the development cycle.

8 Related Work

The Remote Agent (Jónsson et al. 1999) was the first in-space autonomous agent employing AI techniques. The Remote Agent included a planner, executive, fault management, and Mission Manager. The Remote Agent Planner (Jónsson et al. 1999) reasoned about time, resources, and causality. The Remote Agent Experiment set the stage for many successive innovations in AI planning in space.

The Autonomous Sciencecraft Experiment (ASE) from JPL (Tran et al. 2004), (Sherwood et al. 1998) demonstrated automated planning for a low-earth orbiting remote sensing satellite. This planner focused on optimizing science, and did not incorporate fault management.

MEXEC (Troesch et al. 2020), a planner/execution system evaluated for future deep space missions on the ASTERIA Cubesat mission, was integrated with fault management, but only to the extent that execution would terminate and clear the current plan in the event of a fault.

An onboard scheduler has been implemented for use onboard the Perseverance Mars rover (Chi, Chien, and Agrawal 2020). The problem of scheduling Mars rover activities is similar to the problem we must solve for the Gateway VSM, and is also constrained to run on a limited flight processor. Similar design constraints drove the implementation of the algorithms, as described in (Gains et al. 2022). Unlike the planning problem we solve, scheduling Perseverance activities is driven by resources and temporal constraints; there are no 'planning-style' causal constraints to solve.

The Gateway VSM planner differs from these prior efforts in several significant ways. First and foremost is its role in a human spaceflight mission, as opposed to for robotic science missions. Second, the Planner uses a planning model that is similar in spirit to classical AI planning, while most of the prior efforts use different representations, and often are more focused on scheduling problems than planning. Nevertheless, all of the efforts above, and the Gateway VSM, share the complexities of a resource-limited computational environment, mission critical software, in a harsh space environment. As a result, they have all used similar strategies, ranging from algorithm choice to software design to the need for critical testing. The Gateway VSM will ultimately be the latest in a long and rich heritage of enabling autonomous missions through AI planning.

Compared to academic planning research, the modeling language used in our planner for VSM Gateway share a lot of feature with the standard PDDL2.1 (Fox and Long 2003) planning language with a couple of features such as soft state-target goals and prevailing constraints are similar to goal preferences and plan-trajectory constraints introduced in PDDL3.0 (Gerevini and Long 2005). We also natively use multi-value discrete, and not boolean, variables, which was introduced in PDDL3.1. Our planner use the POCL algorithm that is used my many planners such as UCPOP, VHPOP (Younes and Simmons 2003), and CPT (and recently in combination with forward-search planners such as POPF). POCL is more flexible and tends to be able to handle complex temporal constraints easier than state-space search based planners. Our planning-graph based "add" heuristic introduced by the HSP planner and the "open-condition count" heuristic has been used by earlier planners such as UCPOP.

9 Future Work

While our work is on track with current project requirement, looking forward there are features that that we are planning to add to our planner:

Timed Initial Literals: there are temporal constraints in our planning domain such as "eclipse windows" that affect when certain tasks can or can not execute. Those temporal constraints can be modeled similar to "timed initial literals" concept in academic planning research. Given the fact that we store a complete STN inside each search node, we believe that handling such temporal constraints will be straightforward in our planner.

Stronger Resource Handling: there are several directions where we want to future develop our resource handling algorithms. First, it currently lacks support for *generative resource* and our current plan is to extend the existing post-processing routine handling *reusable resource* to also handle generative resources, this allow adjusting the starting times of actions using the same resource r so that the accumulated usage at any given point not only within

[12]https://docs.gitlab.com/ee/ci/

[13]https://github.com/adrienverge/yamllint

[14]https://clang.llvm.org/docs/ClangFormat.html

[15]https://github.com/Psycojoker/pyfmt

[16]https://github.com/mvdan/sh

[17]https://dwheeler.com/flawfinder/

[18]https://github.com/koalaman/shellcheck

[19]https://github.com/nasa/ctf

the allocated amount for r but also within the maximum value given by the resource profile of r. Second, while our current "back-track free" greedy algorithm is fast and so far always produce good quality schedule for our test scenarios, we also want to look into other complete algorithm as a backup in case the current algorithm runs into scenario where it make a wrong choice and not find a solution when one exist. Third, while the reusable-resource handling algorithm is run as a post-processing step, in case where the resource availability is very constrained we may want to detect infeasibility earlier during the search process and thus we can add option to invoke it when each partial-plan is generated.

Comparison Constraint Extension: comparison constraint of the form $x \diamond y$ on variable x is one of the key concept in our planning model that is used to represent the action's prerequisites and invariants, prevailing constraints, and state-targets in goal. In our current planner implementation, y is a value in a domain of variable x but the domain language specifications indicates that y can also be a variable of the same type with x. While we have never encountered a scenario where y is not a variable-value, we do expect in the future it may changes and the planner needs to reason about the comparison constraints $x \diamond y$ in which both x and y are variable. That will requires us to make fundamental changes to key routines in establishing causal-link and goal supports, threat protections, and heuristic computation.

Acknowledgement We acknowledge the support of the Gateway Program, and the Gateway Vehicle System Manager (VSM) Design Team.

References

Aaseng, G.; Do, M.; Frank, J.; Fry, C.; and Planning, A. S. I. 2023. Diagnosis, and Execution for Vehicle Systems Management. In *Proceedings of the Workshop on Integrating Planning and Execution*.

Aaseng, G.; Frank, J.; Iatauro, M.; Knight, C.; Levinson, R.; Ossenfort, J.; Scott, M.; Sweet, A.; Csank, J.; Soeder, J.; Loveless, A.; D, C.; Ngo, T.; and Greenwood, Z. 2018. Development and Testing of a Vehicle Management System for Autonomous Spacecraft Habitat Operations. In *Proceedings of the AIAA Space Conference*.

Badger, J.; Strawser, P.; and Claunch, C. 2019. A Distributed Hierarchical Framework for Autonomous Spacecraft Control. In *Proceedings of the IEEE Aerospace Conference*.

Blum, A.; and Furst, M. 1995. Fast Planning Through Planning Graph Analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, 1636–1642.

Chi, W.; Chien, S.; and Agrawal, J. 2020. Scheduling with Complex Consumptive Resources for a Planetary Rover. *Proceedings of the International Conference on Automated Planning and Scheduling*, 30(1): 348–356.

Crusan, J. C.; Smith, R. M.; Craig, D. A.; Caram, J. M.; Guidi, J.; Gates, M.; Krezel, J. M.; and Herrmann, N. 2018. Deep Space Gateway Concept: Extending Human Presence into Cislunar Space. In *Proceedings of the IEEE Aerospace Conference*.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal Constraint Networks. *Artificial Intelligence*, 49: 61–94.

Fox, M.; and Long, D. 2003. PDDL 2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20: 61 – 124.

Frank, J.; and Jónsson, A. 2003. Constraint-Based Attribute and Interval Planning. *Journal of Constraints Special Issue on Constraints and Planning*.

Gains, D.; Chien, S.; Rabideau, G.; Kuhn, S.; Wong, V.; Yelamanchili, A.; Towey, S.; Agrawal, J.; Chi, W.; Connell, A.; Davis, E.; and Lohr, C. 2022. Onboard Planning for the Mars 2020 Perseverance Rover. In *Proceedings of the* 16th *Symposium on Advanced Space Technologies in Robotics and Automation*, 1040 – 1045.

Gerevini, A.; and Long, D. 2005. Plan Constraints and Preferences in PDDL3. Technical report, Department of Electronics for Automation, University of Brescia, Italy.

Gerevini, A.; and Schubert, L. 1996. Accelerating partial-order planners: some techniques for effective search control and pruning. *Journal of Artificial Intelligence Research*, 5(1): 95–137.

Jónsson, A. K.; Morris, P. H.; Muscettola, N.; and Rajan, K. 1999. Next Generation Remote Agent Planner. In *Proceedings of the Fifth International Symposium on Artificial Intelligence, Robotics and Automation in Space*.

McComas, D.; Wilmot, J.; and Cudmore, A. 2016. The Core Flight System (cFS) Community: Providing Low Cost Solutions for Small Spacecraft. In *Proceedings of the* 30th *AIAA /USU Conference on Small Satellites*.

NASA. 2018. Spaceflight Demonstration of a Power and Propulsion Element (PPE). https://www.fbo.gov/spg/NASA/GRC/OPDC20220/80GRC018R0005/listing.html. See Amendment 6: Unique Requirements.

Nguyen, X.; and Kambhampati, S. 2001. Reviving Partial Ordering Planning. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, 459 – 464.

Sherwood, R.; Govindjee, A.; Yan, D.; Rabideau, G.; Chien, S.; and and, A. F. 1998. Using ASPEN to Automate EO-1 Activity Planning. In *Proceedings of the IEEE Aerospace Conference*.

Srivastava, B.; Kambhampati, S.; and Do, M. 2001. Planning the project management way: Efficient planning by effective integration of causal and resource reasoning in RealPlan. *Artificial Intelligence Journal*, 131: 73–134.

Tran, D.; Chien, S.; Sherwood, R.; Castaño, R.; Cichy, B.; Davies, A.; and Rabbideau, G. 2004. The Autonomous Sciencecraft Experiment Onboard the EO-1 Spacecraft. In *Proceedings of the* 19th *National Conference on Artificial Intelligence*, 1040 – 1045.

Troesch, M.; Mirza, F.; Hughes, K.; Rothstein-Dowden, A.; Bocchino, R.; Donner, A.; Feather, M.; Smith, B.; Fesq, L.; Barker, B.; and Campuzano, B. 2020. MEXEC: An Onboard Integrated Planning and Execution Approach for Spacecraft Commanding. In *Proceedings of the Workshop on Integrated Execution and Goal Reasoning (InTEX/GR)*.

Younes, H.; and Simmons, R. 2003. VHPOP: Versatile Heuristic Partial Order Planner. *Journal of Artificial Intelligence Research*, 20: 405–430.