The Gateway Vehicle Systems Manager Planner

Michael Whitzer[1], Anthony Koutroulis[3], Hemanth Koralla[4], Chris Knight[2], Chuck Fry[3], Jeremy Frank[2], Minh Do[3], J. Benton[2], Laura Barron[1], Abiola Akanni[2]

[1]Software, Robotics and Simulation Division, NASA Johnson Space Center
[2]Intelligent Systems Division, NASA Ames Research Center
[3]KBR Wyle Services, NASA Ames Research Center
[4]Jacobs Technology, Inc, NASA Johnson Space Center

Abstract

NASA has developed an automated planner as part of the Vehicle System Manager (VSM), integrated with flight software, to control a habitable spacecraft, currently referred to as Gateway. 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.

1 Introduction

NASA plans to construct a habitable spacecraft, currently referred to as Gateway (Crusan et al. 2018), in the vicinity of the Moon and has developed and tested multiple technologies to enable its autonomous operation. These technologies include a Vehicle System Manager (VSM), integrated with flight software, to control the habitat. Building 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, we have developed an automated planner as part of the Gateway VSM. In this paper, we focus on the knowledge engineering challenges we have encountered in our work. Our automated planner inherits recognizable qualities from decades of research and development efforts for handling tasks involving time and resources. However, integrating a planner with systems designed for human spaceflight posed numerous challenges. We focus on these knowledge engineering aspects of the VSM planner and how they influence the current planner design as we maintain the heritage of model-based automated planners developed by the AI planning community.

It was essential that we considered software engineering design decisions, external interface requirements, and software quality requirements, all of which influenced the final product. For instance, the planner employs a new modeling framework specific to the VSM. Instead of using conventional flat-file storage of models, we were required to interface with an SQLite database that serves multiple VSM functions, as well as other VSM applications including fault management, resource management, and state determination. These factors influence the planner's architecture. The database stores all of the possible tasks the planner can use, which necessitates pre-processing routines such as reachability and relevance analysis to narrow down the set of relevant tasks. Additionally, the planner will operate on a PowerPC SP0-S processor[1], which offers significantly less power and memory compared to the standard desktop computers on which most research planners operate. Human spaceflight software requires the highest levels of safety and quality assurance, which imposes a variety of software testing and quality control requirements on the planner. All of these requirements drive language choice, coding standards, and algorithms that facilitate automated software testing.

The paper is structured as follows: in Section 2, we describe our planning model. In Section 3, we describe different components of our VSM Planner including preprocessing routines, core planning algorithm, heuristic-guided search algorithms, and resource handling routines. In Section 4, we describe the external interfaces between the planner and the rest of the VSM. In Section 5, we describe the testing procedure for our planner. Finally, in Section 6, we describe the software engineering decisions that influence planner development and algorithms.

2 Planning Model

Gateway, our primary operating domain, drove the development of our planning model. One unique aspect of our model is the language used to specify it, which integrates features similar to the temporal and numeric aspects of PDDL, while also introducing unique modifications derived from Spacecraft Onboard Interface Services (SOIS) compliant Electronic Data Sheets (SEDS).[2] The language has many familiar features. The model uses state variables (fluents) that can be assigned either enumerated or numeric values, but also includes first-class representation for resources and resource requirements. Another notable feature is the introduction of *prevailing constraints*, requirements which must be satisfied through the entire plan execution, based on vehicle *mode & configuration*. We will elaborate on those concepts in the

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

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

later part of this section.

The input to our VSM planner contains three main parts: **Goals**, **Tasks**, and **Telemetry** (i.e., initial state). The output plan $P = \{\langle a_1, t_1\rangle, \ldots, \langle a_n, t_n\rangle\}$ is the list of tasks and their starting times. Tasks are allowed to overlap and may start at the same time. 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**: A relation $x \diamond y$ in which x is a variable (discrete or numeric), y is a either (i) another variable of the same type or (ii) a legal value of x, and $\diamond \in \{=, \neq, >, \geq, <, \leq\}$.
- **Variable Assignment**: An assignment $x \leftarrow y$ in which x is a variable, either discrete or numeric, and y is a legal value of x.
- **Resource**: A numeric variable r with bounded value that is changed by task T during its execution.
- **Resource Requirement (ResReq)**: A resource requirement 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 and 'releases' increases the available resources. All impacts are instantaneous.
- **Resource Kind**: In our data model, each resource is also labeled with a *resource-kind* value that can be one of the following: $\{REUSABLE, CONSUMABLE, REGENERATIVE, CLAIMABLE\}$. In Section 3.1, we will discuss in more details the relationship between *resource-use-property* and *resource-kind* values and how they are used during planning.
- **Mode & Configuration**: The system can be specified to operate in a particular mode or configuration and each combination of $\langle mode, configuration\rangle$ is associated with a particular set of *comparison constraints* that needs to be satisfied throughout the plan execution. These *prevailing constraints* are discussed in Section 2.3.

2.1 Goals

Our planner supports both *hard* and *soft* goals with preferences. While the Gateway VSM allows various methods to specify soft-goal values, the planner currently simplifies these to a single scalar *preference* value for two types of goals: (i) a set of tasks or (ii) a set of comparison constraints:

- *Task-As-Goal (TAG)*: A *Task-As-Goal (TAG)* g_T contains a specific set of tasks $T = \{a_1 ... a_n\}$. For a hard-goal g_T, a plan P is considered valid if it includes all tasks a_i in T. For a soft-goal g_T with a preference value v_g, the plan P receives a value of v_g only if it includes all tasks in T (partial completion of T does not get any value).
- *State-Target (ST)*: A *State-Target (ST)* goal g_S is defined by a set of comparison constraints: $g_S = \{\langle p_1 \diamond q_1\rangle, \ldots \langle p_n \diamond q_n\rangle\}$. For a hard-goal g_S, the plan is valid if the final state S_e satisfies all constraints in g_S. If g_S is a soft-goal associated with a value v_g, the plan P gains v_g if the final state S_e meets all constraints in g_S (like soft TAG, partial-satisfaction does not get any value).

Our planner can process any arbitrary combination of hard and soft goals of either type TAG or ST, and the main objective function is to: (1) satisfy all *hard-goals*; and (2) maximizes the accumulated values of *soft-goals* (i.e., $\sum v_g$).

2.2 Tasks

Task models in our system closely resemble the durative actions found in PDDL2.1 (Fox and Long 2003). Specifically, each task is specified by the following components:

- **Prerequisites:** a list of conditions, each a *comparison constraint*, that must be met at the task's start.
- **Invariants:** a list of conditions, each a *comparison constraint*, that must remain true for the duration of the task.
- **Effects:** a list of instantaneous *variable assignments* applied at the end-time of the task.
- **Resource Requirements:** a list of *resource requirement* incurred by the task.
- **Duration:** an estimated and worst-case durations, denoted by $\langle dur_e, dur_w\rangle$. Currently, our planner consider only the worst-case duration dur_e.

Discussion Our task model aligns with the at-start and overall durative conditions of PDDL2.1 through *prerequisites* and *invariants*. However, unlike PDDL2.1, it does not feature *at-end* instantaneous conditions and there's also no provision for *at-start* effects. The VSM language also makes resource requirements explicit and integral to task modeling. Task models in our framework are fully grounded with fixed task durations and resource utilization properties. TAGs can be modeled as classical goals using existing AI planners, but are included as first-class goals in the Gateway model.

2.3 Initial State

At the onset of the planning process, the planner receives current system state information via *telemetry* messages from software components that monitor various hardware modules. This system information comprises the *initial state* for the planner, detailing the initial values of state variables and the *profiles* of resource availability. Specifically:

- **Initial Variable Assignments**: initial values of all state variables.
- **Resource Profiles**: each resource r is given a profile defined by: $P_r = \langle Max_r, min_r, A_r\rangle$ in which:
  - ◇ Max_r and min_r are the maximum and minimum values of r that the planner needs to keep the resource amount within (i.e., $Max_r \geq value(r) \geq min_r$) throughout the plan.
  - ◇ $A_r = \{\langle v_1, t_1\rangle \ldots \langle v_n, t_n\rangle\}\}$ is a step-function that describes the amount of r that is available for the planner

to use over the planning horizon where: (i) $t_1 = 0$ denotes the initial availability of r, and (ii) each pair $\langle v_i, t_i \rangle$ indicates that v_i is the amount of r available from time t_i until t_{i+1}.

- **Mode & Configuration:** special state variables x_m and x_c represent the system's operating mode and configuration for the duration of the planning process. The mode and configuration are retrieved with the other telemetry values and can be thought of as variable assignments of the form $(x_m \leftarrow y_m) \wedge (x_c \leftarrow y_c))$.
- **Prevailing constraints:** are comparison constraints that must be satisfied through the entire plan execution and take 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, when *mode* x_m and *configuration* x_c take on specific values, a corresponding set of comparison constraints the plan must adhere to for its entire execution are enabled. These resemble the plan-trajectory constraints, modeled as Linear Temporal Logic (LTL) formulas, in PDDL3.0 (Gerevini and Long 2006).

3 Gateway VSM Planner

The Gateway VSM Planner combines goal achievement reasoning and scheduling to resolve potential resource conflicts. This section concentrates on the planner's internal working while leaving the discussion on interfacing with other software components to Section 4.

3.1 Planning Model Building

The Onboard Data Model (ODM) is an SQLite database containing the variables and tasks that capture the whole VSM system configuration, and all possible potential goals that the planner may have to plan for. Clearly, in any given planning scenario, with its specific goals and initial conditions, much of the ODM may be unnecessary or irrelevant. Therefore, the VSM Planner runs pre-processing steps to narrow down the set of tasks, goals, and the associated variables to make the planning model as compact as possible. We first load the ODM task library from the SQLite database. *Conflict Analysis* pre-detects and stores information on conflicting task pairs. This helps speedup the subsequent phase of planning search where task conflicts lead to *task-orderings* and *causal-link threats*. *Relevance Analysis* finds the tasks and variable-assignments that are 'relevant' to the goals. Finally, *Reachability Analysis* finds the tasks and variable-assignments that are 'reachable' from the initial state.

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

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 links back 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 change 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.

Filtering disabled tasks: 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 request are read from the ODM.

Acquire Initial Values: Projected initial state variable values S_{init} are requested from the *State Determination* cFS app, and resource availability projections RP_{init} are obtained from the *Resource Manager* app. Prevailing constraints C_{prev} are read from the ODM, based on the projected mode (i.e., mission phase) and configuration of the assembled spacecraft in S_{init}.

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 those which either achieve the goals directly or support the conditions that enable other tasks to achieve the goals. Specifically, let T_R and C_R be the relevant set of tasks and comparison-constraints. A sketch of the Relevance Analysis routine 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 support a comparison-constraint in C_R.
3. *Relevant Variable-assignments Update*: add to C_R all prerequisites and invariants of new tasks added to 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 variable-assignments relevant to the set of TAG and state-target goals given to the planner. Also note that in those two steps, any task or variable assignments 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 finds tasks and state-assignments that can be *reached* from the initial state through executing a set of tasks that are applicable. A sketch of the Reachability Analysis is as follows:

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.

Tasks and variable assignments absent from the final sets T_I and S_I can be excluded from the planning model. If there is a TAG T_g or a state-target goal ST_g that is not entirely in T_I or S_I, respectively, then: (i) If T_g or ST_g is a *soft*-goal, it's removed from the planning model; (ii) If T_g or ST_g is a *hard*-goal, then we can declare the planning problem unsolvable.

Resource Type Inference: As outlined in Section 2, each resource R is classified as one of 4 types, *REUSABLE, CLAIMABLE, REGENERATIVE, CONSUMABLE* that the Planner relies upon during planning (refer Section 3.5). Also as outlined in Section 2, each task can have a resource-requirement on R of one of the six different types. 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 remaining tasks in T_I. Therefore, the pre-processing phase includes a routine to *infer* the resource-kind to be consistent with how resources are utilized by the final set of relevant & reachable tasks.

Specifically, let T be the set of task 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 exist $t \in T_r$ that 'release' (i.e., generate) r at any point during its execution, then r is classified as REGENERATIVE.
- *Line 4-5*: If exist 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.

3.2 Planner Setup

Algorithm 1 captures the key steps described in this section. Specifically, when the Planner app is starting up, it loads all

	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.

tasks from the ODM (line 1) and conducts *Conflict Analysis* to identify potential effect-effect and effect-condition conflicts between all pairs of tasks in T_{all} (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. It first invokes the *pre-processing* routine to assess the current task-status flags and exclude all tasks that are currently disabled from consideration (line 17), and extract the specified goal information (line 18) from the ODM.
2. The planner retrieves (lines 21-25) from other cFS apps: (1) the initial values for state variables from *State Determination*; and (2) resource allocation profiles from *Resource Manager*. It will then fetch applicable *prevailing constraints* (line 24) from the ODM based on vehicle mode and configuration.
3. With the concrete set of goals and tasks, initial values for state variables, resource allocation profiles, and the prevailing constraints, the Planner calls *Relevance Analysis* (line 27-31) to find all tasks, variables, and resources relevant to the goals.
4. It then calls *Reachability Analysis* (line 33-36) to further narrow the set of relevant tasks to only those tasks reachable from the initial state.
5. The final planning model is built (line 38-44) from tasks, goals, variables & variable assignments, and resources that are both relevant and reachable.

After the planning-model is built, the planner will try to find a valid plan using the Partial Order Causal-Link (POCL) planning algorithm (Penberthy and Weld 1992), and when found will send the plan to the *Dispatcher* cFS app for execution (line 12). Simultaneously, the planner will request that the *Goal Tracker* cFS app tracks the achievements of the given goals (line 13). After that, the planner returns to the idle loop waiting for the new plan-request.

3.3 POCL Planning Algorithm

Our POCL planning algorithm starts from an initial partial-plan P_{init} containing only tasks that belong to 'hard' TAG goals, and continues to expand it by generating 'child' partial-plans that add causal-link support for goals and resolve unsupported action conditions.

Algorithm 1: Gateway VSM Planner as cFS Application

```
1: T_all ← 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 AcquireInitialValues(V, R)
8:     Call RelevanceAnalysis(G)
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 ← retrieve from ODM using goal IDs in PR
19: end procedure
20:
21: procedure AcquireInitialValues(V, R)
22:    S_init ← values for V from the StateDetermination cFS app
23:    RP ← resource profiles for R from the ResourceManager cFS app
24:    C_p ← applicable prevailing constraints from ODM
       using mode & configuration from S_init
25: end procedure
26:
27: procedure RelevanceAnalysis(G, S_init, RP, C_p)
28:    T_rel ← tasks in T_all that are relevant to G given
       S_init, RP_init, and C_prev
29:    V_rel ← variables appear in conditions of T_rel
30:    R_rel ← resources utilized by tasks in T_rel
31: end procedure
32:
33: procedure ReachabilityAnalysis(S, T)
34:    T_rec ← tasks in T_rel that are reachable from S_init
35:    S_rec ← effects of T_rec
36: end procedure
37:
38: procedure BuildFinalPlanningModel
39:    V_final ← variables appear in both V_rel and S_rec
40:    T_final ← T_rec
41:    R_final ← resources used by T_final
42:    G_final ← goals supported by tasks in T_rec
43:    Run ResourceTypeInference.
44: end procedure
```

Each causal-link connects a task effect e to another task's condition or a state-target goal, for which the comparison constraint representing it is satisfied by the variable-assignment representing e. Since task effects can conflicts with each other or violate other tasks' prerequisites/invariants, the POCL algorithm also manage all conflicts between tasks in the (partial) plans and maintain the task orderings resulted from conflicts. Each partial-plan integrates a Simple Temporal Network (STN) (Dechter, Meiri, and Pearl 1991) that captures all temporal relations imposed by causal-links and task-orderings to resolve conflicts, allowing us to detect temporal violations (e.g., circular causal dependency between two tasks) early. A partial-plan is a complete plan if: (1) all 'hard' goals are supported; (2) there is no task-condition that is not supported; (3) no temporal or resource constraint violation. While each plan P is represented by a set of causal-link supports, P's STN can give the *earliest* time at which each task can execute. The Planner then send the set of tasks and the associated earliest-start-time to the *Dispatcher* cFS app (refer Figure 1) for execution.

We use a variant of Best-First Search, with the heuristic function deciding the 'best' node briefly explained in Section 3.4, to navigates the set of partial-plans generated by the POCL algorithm. To compensate for the heuristic quality, and to handle the combination of hard and soft goals described in Section 2.1, we have 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 generating plans as long as there are still not-visited search nodes, potentially finding a better solution. Thus, our algorithm may find multiple solutions with gradually better plan quality, and at any point $t_{stop} \leq t^s_{limit}$, we can return the best solution found by t_{stop}. For example, the first complete plan P_1 that it returns at $t_1 < t^s_{limit}$ may satisfy all hard-goals without addressing any 'soft' optional goals that could add 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.

Implementation details: 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). While our POCL algorithm handles *consumable* resources directly, it defers handling of *reusable* resources (refers Section 2) to a 'scheduling' phase after a complete plan is found. This phase employs an incremental and iterative heuristic algorithm. Notably, neither STN propagation or reusable resource handling algorithm uses backtracking, and there is no need for retraction of constraints during either the planner or scheduler search.

The planner's best-first graph search algorithm can generates and visit many equivalent search states, leading to increase planning time, and requires additional queue space. We employ a variant of duplicate-detection, in which the entire search history is recorded in a *hash table* of partial plans. Prior to being enqueued, new partial plans are checked against this history using first the plan's hash value, and then the full partial-plan structure if the hash is matched. Unique partial plans are enqueued and recorded in the table; duplicates are dropped from consideration. The space-time tradeoff inherent in this approach is discussed in Section 6.

3.4 Heuristics

Heuristics are crucial in providing estimates to rank search nodes and decide which partial-plans are the 'best' to ex-

plore next. At the high-level, heuristics help guide the planning search algorithm towards finding a *valid good quality plan efficiently*, leading us to design a multi-factor heuristic function that is geared toward ranking partial plans based on the following factors:

1. Find a *valid* plan: Thus, it first prioritizes partial-plans nearing the state of a complete, valid plan, which means supporting all 'hard' goals.
2. Find a *good quality* plan: Since plan quality is determined by the sum of 'soft' goal values, the heuristic then gives precedence to plans that accumulate the highest goal values.
3. Find a plan *efficiently*: Next, our heuristic prioritizes partial-plan that is closer to a complete plan, thus require adding fewer new actions and having fewer goals and open-conditions that need to establish new support.

Specifically, we rank partial-plans by these heuristics, ordered by their preferences: (1) fewer unsatisfied mandatory 'hard' goals; (2) higher total potentially achievable soft goal value; (3) higher accumulated soft goal value; (4) higher number of goals supported; (5) lower planning-graph additive heuristic value; (6) fewer unsupported conditions; and (7) fewer actions. Several of those heuristics are inspired by the ones used in existing POCL planner such as VHPOP (Younes and Simmons 2003) or UCPOP (Penberthy and Weld 1992).

3.5 Resource Handling

As described in Section 2, there are 4 different resource types in our planning model: *REUSABLE* (with *CLAIMABLE* being a special case), *CONSUMABLE*, *REGENERATIVE*. Currently, our planner handles *CONSUMABLE* and *REUSABLE* resources.

Consumable resources are managed during the planning phase. When a task is added to the plan, its resource consumption is assessed against projected availability. If the total consumption surpasses availability, the planner identifies a resource conflict. Reusable resources are handled after a complete plan is found by a greedy resource repair algorithm. 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). Specifically, the repair algorithm progresses from the start of the plan, addressing each resource conflict sequentially. It employs multiple greedy heuristics: a 'minimum time perturbation' heuristic for direct conflict resolution and a 'minimum resource usage' heuristic to mitigate the risk of future conflicts when adjusting task timings. There is no iterative feedback loop between the planner and scheduler; instead, the algorithm relies on the planner's generation of alternative task sequences that facilitate effective heuristic application. Our current scheduling algorithm for reusable resources runs greedily in polynomial time and doesn't guarantee completeness.

4 Planner Interfaces

As outlined in Section 3 and demonstrated in Algorithm 1, the Planner's integration with numerous VSM applications

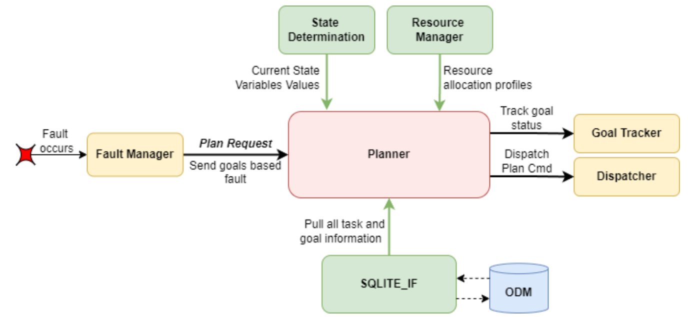

Figure 1: Planner interfaces to other VSM applications.

is vital for its operation. These applications include *SQLite Interface* (SQLITE_IF), *Fault Manager*, *Resource Manager*, *State Determination*, *Dispatcher*, and *Goal Tracker*, each playing a unique role:

- **SQLITE_IF**: As mentioned previously, the Gateway Onboard Data Model (ODM) stores all of the onboard data required for the operation of the vehicle. Like other VSM applications, the planner utilizes the SQLITE_IF app 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.
- **Resource Manager** (RM): The RM app is specifically responsible for calculating and reporting current & projected state, and trending information for resources such as power, bandwidth, and storage capacity. These reports and calculations 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.
- **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** (GT): The GT app 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. GT 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 1 describes these interfaces between the Planner and the other apps. We explain this figure, referring back to Algorithm 1 in Section 3, in the remainder of this section. In the event that a fault occurs, FM may initiate a recovery process after Gateway has achieved a safe state. When the Planner receives a replan command from FM (Algorithm

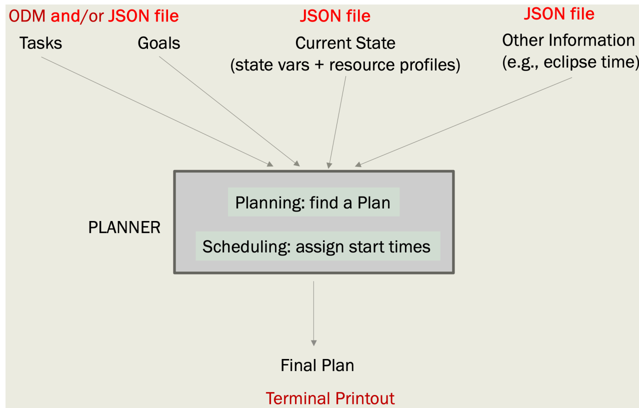

Figure 2: Planner running in the Offline Mode.

1, line 5), it filters the task set it previously retrieved from the ODM using SQLITE_IF for any disabled tasks, then uses SQLITE_IF to retrieve goal data from the ODM for the goal(s) it received from FM (line 6). Next, the Planner obtains projected state variable values at the planning horizon start from the SD app, and resource projections from the RM app (line 7). It then performs Relevance Analysis (line 8) and Reachability Analysis (line 9) to eliminate irrelevant and unusable tasks from the search space. The analysis results are used to build the final planning model (line 10). With this data, the Planner has all of the initial conditions needed to initiate the planning algorithm (line 11). 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. The green arrows in Figure 1 represent data inputs to the Planner for the problem setup. The figure also contains interactions between the planner and other apps during the planning process.

5 Verification & Testing

Given its critical role within the Gateway VSM, rigorous testing is imperative to ensure the Planner's reliability and defect-free operation. Defects arise in different parts of the planner; inside any of the numerous constituent planner 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 *planning model*, *interfaces*, and *algorithmic* errors are critical to ensuring a quality software product.

Algorithm 1 outlines our planner running in a continuous closed-loop interacting with other cFS apps (as shown in Figure 1). While this is how the planner is deployed, it is difficult to effectively test solely on the deployment architecture. 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, the ODM contents (the planning model and goals) are configuration managed, with multiple VSM stakeholders depending on its contents.

ODM Checker: 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 3.5, 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 planner's inputs prior to 'scenario-level' testing using the ODM itself.

Offline Testing with JSON Planning Model: 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 allows us to test it without compiling or running other cFS apps providing its input such as Fault Manager, State Determination, or Resource Manager. We can also test any planning scenarios that may involve tasks/goals/variables that currently are not in the ODM, or are slightly different from their configuration managed definition in the ODM. This 'offline' test mode supports rapid development and allow us to work on future-looking features of the planner.

To support offline testing, we developed a modeling framework based on the JSON format that mimics all planner inputs described in Section 2. 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, can also be specified in the JSON format. Figure 2 shows the input and output information flow diagram of our planner running in the offline testing mode.

Preliminary Results: our Gateway VSM planner is fully implemented and integrated with other cFS apps as shown in Figure 1 and has been tested with small but realistic scenarios. The online integration testing has been done both by us and other groups working on Gateway VSM. For offline testing, we have an automated test suite currently consist of 66 scenarios that test different technical capabilities of our planner (e.g., different goal combination, different resource types with different availability profiles, task conflicts etc). The entire test suite runs in under 5 seconds on current desktop computers, despite the search algorithm generating in excess of one hundred partial-plans in several of these scenarios.

6 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 design decisions.

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, enabling it to quickly down-select task and goal information as needed using the SQL language.

Memory Management: Gateway VSM runs as part of the flight software load on the HaLO module. The Aitech SP0-S flight computer is a COTS single board computer with significant spaceflight heritage. The SP0-S is based on a PowerPC system-on-a-chip introduced in 2005. It has one 32-bit CPU core, which runs at a 1 GHz clock rate, and 1 GiB of DDR2 RAM. The COTS real-time operating system doesn't support virtual memory paging, meaning all active code and data reside in physical memory. The file system is loaded from flash storage at startup, and also resides in main memory during execution. The Core Flight Software system (McComas, Wilmot, and Cudmore 2016) and spacecraft flight software standards impose additional constraints[3]. Applications may use a standard cFS dynamic memory allocator library, but the allocation region must be reserved statically at build time. Process stack space is also set statically at startup. Recursion is prohibited by flight rule. These constraints have influenced the design of Planner elements such as the anytime search algorithm, search state duplicate detection, and the Simple Temporal Network (STN).

Fixed size objects are allocated from static arrays. Variable size structures are dynamically allocated from a static pool. A 1 MiB pool has proven sufficient for planning problems requiring up to hundreds of search steps. The STN propagation algorithm is iterative with a fixed size queue. STN structures have been optimized for the relatively small problem sizes. 2-byte array indices are used in place of pointers. The resulting network size limits of 64k nodes and 64k edges have proven more than adequate for the problems the Planner will be required to solve. Despite representing time as an 8-byte integer representation, 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. STN 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.

The anytime search algorithm retains the best complete plan found so far, and updates it when a better plan is found. Search history storage for duplicate detection dominates memory pool usage for larger problems, causing memory exhaustion after a few hundred search steps. If memory runs out before search times out, search is terminated immediately. If at least one complete plan has been generated, the best plan is returned.

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[4] jobs to ensure code quality, code formatting (yamllint[5], clang-format[6], pyfmt[7], shfmt[8]), and code static analysis (flawfinder[9], shellcheck[10], and internally-developed database and code checking tools). Functional testing includes the cFS Test Framework (CTF)[11], cFS unit tests, and a standalone test framework. Critical to support development is the ability to monitor code quality on changed code so that code quality would improve incrementally and reduce the effort to meet code compliance later in the development cycle.

7 Conclusions and Future Work

Future requirements for VSM planner capability will include: replanning, handling of complex resource availability profiles, generalization of resource types, timed initial literals and constraints, and mode and configuration changes. We also expect changes in the domain model and increases in the size of plans we must generate. These changes will require evolving the planner algorithms and heuristics, especially in light of the deployment environment.

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

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

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

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

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

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

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

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

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

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*.

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*.

Gerevini, A.; and Long, D. 2006. Preferences and Soft Constraints in PDDL3. In *Proceedings of the ICAPS Workshop on Preferences and Soft Con- straints in Planning*.

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*.

Penberthy, J.; and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, 103–114.

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.

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