Scaling Distributed Spacecraft Autonomy Operations with Simple Temporal Networks: Introducing Splinter

Michael Iatauro, Caleb Adams
NASA Ames Research Center
Mail Stop N269-3, Moffett Field, CA 94035; (650) 604-0662
michael.j.iatauro@nasa.gov

Caroline Lassiter
KBR Wyle Services
Mail Stop N269-3, Moffett Field, CA 94035; (650) 604-1731
caroline.r.lassiter@nasa.gov

ABSTRACT

The Absolute Time Sequence (ATS) is a well-established mechanism for high-level spacecraft control, executing commands at specific times. One weakness of ATSs is that the cost of making a change to the command sequence grows with the length of the sequence, and the cost paid is many repetitive-but-detailed changes to the timing of commands, which is error-prone.

This weakness becomes a liability when scaling up to spacecraft swarms or long ATS durations because any perturbation to the command sequence or schedule incurs significant cost. Fitting that cost within a small team or short sequence development timeline becomes a central challenge for mission operations.

The Distributed Spacecraft Autonomy (DSA) experiment, on-board Starling addressed these challenges by developing 'Splinter', a tool that utilizes the implicit order of a text file, a custom language for stating temporal constraints, and a Simple Temporal Network (STN) implementation to rapidly develop and update command sequences and coordinate events in time across a small satellite swarm, integrating into the core Flight System command products workflow. This paper presents the tool and discusses the specific mission challenges that motivated its development as well as the experience of using it.

BACKGROUND AND INTRODUCTION

NASA Starling[1] is a four-spacecraft CubeSat mission established to advance technology readiness for cooperative groups of spacecraft, particularly in ad-hoc cross-link networking, orbit maintenance, formation flying, and science autonomy.

The Distributed Spacecraft Autonomy project, a software payload on-board Starling, has demonstrated swarm science autonomy[2], cross-commanding[2], torrent-inspired file transfer[3], and enhanced local autonomy[1]. For the purposes of this discussion, we will focus on the science autonomy demonstration.

In brief, DSA uses GPS signals to measure plasma density in the Earth's ionosphere. The phase difference of the L1 and L2 signals is used to calculate the Total Electron Content (TEC) on the path between the measuring spacecraft and the GPS satellite. The autonomy application assigns each Starling spacecraft GPS signals it may observe at a given time. The rewards for those observations are then shared and incorporated into the next planning cycle. Each member of the swarm computes a plan for the whole swarm based on the information it has but only executes its portion of the plan.

The major parameters of the autonomy demonstration are:

1. The number of distinct GPS signals each spacecraft can observe.
2. The minimum number of spacecraft allocated to a GPS satellite for it to be considered observed
3. The relative priorities between:
    a. The number of distinct satellites.
    b. The distance to the satellites.
    c. The TEC measured along each path.

These parameters are controlled in the DSA/Starling flight software using a core Flight System table—essentially a serialized C struct.

During the initial DSA mission, the operations team controlled the demonstrations, including the values of

the autonomy parameters, by Absolute Time Sequence (ATS), loading tables with a pre-determined set of parameters at specific times.

INCITING INCIDENT

For a particular pre-launch mission rehearsal DSA Ops had a goal of stress-testing the command product authoring, assembly, and delivery process. That meant writing four 24-hour timelines which would:

- Change the autonomy parameters up to nine times each orbit
- Maintain a minimum demonstration duration with each set of parameters
- Manage a mutual exclusion between using the crosslink radio transmitters and the space-to-ground radio transmitter
- Manage a one-hour-on/one-hour-off crosslink radio cycle to satisfy a battery voltage flight rule

The rehearsal began at 09:00 with a delivery of flight dynamics products and a space-to-ground communication schedule. The DSA team immediately began working on calculating times of latitude crossings, correlating them with the comm schedule, determining which of the spacecraft would be on the crosslink network at what times, and fitting a demonstration schedule to that.

In approximately four hours, we were able to generate four command sequences spanning 24 hours totaling 1164 commands. We broke for lunch at 13:00 and returned at 14:00 to a manner of request that every operations team expects and dreads:

> Can you shift the timelines by 15 minutes?
>
> And have it done by 16:30?

The team scrambled to re-work the command sequences, determining which experiments would have to be shortened, which would become too short, which would now have different sets of spacecraft on the network, and which conflicted with radio black-outs. At 16:30, we had completed the new timelines that *probably* complied with all flight rules. Correctness of the command products wasn't a measured objective of the rehearsal, and that was fortunate. Had the timelines ever been executed, it isn't clear that they would have correctly run the experiments.

Anecdotally, the greatest difficulty in drafting the initial timelines was in combining the four space-to-ground communication schedules into an availability list and then fitting the experiments to that schedule. The greatest challenges of revising the timelines were in correctly propagating schedule changes to other affected commands and in maintaining awareness of why a command was at a specific time, particularly relative to exogenous events not represented in the timeline.

THE LIMITATIONS OF ABSOLUTE TIME SEQUENCES

The semantics of ATSs are simple, in principle: at time *t*, execute command *c*. However, ATSs rely on *time*, which is measured by *clocks*. So, for all the apparent simplicity of the ATS, it is limited by the simplicity of the engineering of clocks and the physics of time.

Authoring an ATS is similarly "simple": decide what command to execute, when it should execute, and write that. However, as an artifact of the process of deciding *what* and *when*, the ATS contains almost no information about the process that generated it, so -modifying it requires some external documentation or inference to determine any invariants that need to be preserved or previous decisions that need to be discarded.

Software has an analogous problem, which is instructive to consider. An ATS is very much like a program written in assembly language: as a step-by-step list of instructions, the syntax and semantics are close to those of the system that will execute them rather than those of the problem domain, and there are few-to-no abstraction mechanisms to represent intent.

As a result, authoring and especially modifying ATSs or assembly language requires maintaining externally or reverse-engineering the abstraction and intent. In an ATS, much of the intent is held in the schedule of commands. Changing a single time (say, of a transmitter-on command) can have far-reaching consequences that are challenging to determine.

Splinter was written to address these limitations in two ways. First, by allowing users to annotate their ATSs arbitrarily, including in ways that can affect the command schedule. Second, by allowing them to express the *constraints* that result in the final schedule, which are then represented in a Simple Temporal Network from which the final schedule is computed.

SIMPLE TEMPORAL NETWORKS

Simple Temporal Networks (STNs) have a venerable history in planning and scheduling applications and literature. They were first published by Dechter, et al.[4], in which they were described as a pair $S = (T, C)$ where:

- T is a set of real-valued variables called "timepoints", with a special timepoint $Z = 0$; and

- C is a set of binary constraints, each of the form $Y - X \leq \delta$, with $X, Y \in T$ and $\delta \in \mathbb{R}$. Unary constraints can be reinterpreted as binary constraints with Z.

Splinter uses an approach derived from the work of Paul Morris on implementing STNs in the EUROPA[5] and EUROPA2[6] constraint-based planning systems. In this approach, timepoints, as well as each δ ,are closed-ended integer intervals that may include $\pm\infty$, and constraints take the form $X + \delta = Y$. We use the subscripts *lb* and *ub* to denote the lower and upper bounds of an interval, e.g. X_{lb} and X_{ub}.

The graph for an STN $S = (T, C)$ is a directed weighted graph $G = (V, E, w)$ with a vertex in V for each timepoint in T. Each constraint in C becomes two edges in E, with $w(X, Y) = \delta_{ub}$ and $w(Y, X) = -\delta_{lb}$.

From this graph structure, we are interested in two properties—consistency and bounds. An STN is *inconsistent* when no possible assignment to the timepoints can satisfy the constraints. This arises in the graph structure as a negative cycle. The bounds for the timepoints can be computed using an all-pairs-shortest-path algorithm.

Splinter uses an STN implementation developed at the NASA Ames Research Center called "Tempus Fidgets", which accomplishes negative cycle detection in $\Theta(|V||E|)$ time using the Bellman-Ford algorithm and Dijkstra's single-source-shortest-path algorithm to compute the bounds in $\Theta(|E| + |V| \log |V|)$ time. Tempus Fidgets also has incremental versions of these algorithms to more efficiently recompute as constraints are added or narrowed.

SPLINTER

The Splinter tool is implemented as a plugin for the text editor Sublime Text 4[7]. Sublime Text provides both a familiar suite of text editing capabilities as well as an integrated Python environment. All the parsing of Splinter files is implemented in Python, which interfaces with Tempus Fidgets through pybind11[8]-generated bindings for building, propagating, and accessing STN information. Figure 1 is a simple Splinter file with most constraint features present to ease description.

```
1   An example Splinter file
2   [] Day start #palstart;=0m!
3   Wait an hour for the payload processor to boot and prep
4   [] ATS start #atsstart;@palstart 1h!;v0s
5   [] "/validate DS.FILTER_TBL" #v30s
6   [] "/CFE_ES_StartApp with APPLICATION 'DSA_COMM_APP'" #v10s!
7   [] "/ADMIN_XlinkON" #radios_on;@dsa2_crosslink 0!
8   [] "/DSA_LFT_PUSH with fileName ./cf/sv2_sv4_med2.txt"
9   [] "/Admin_XlinkOFF" #@radios_on 55m!
10  [] RESTART START #=2026-143T12:00:00!;reset_start;v30m!
11  [] RESTART END #=2026-143T12:30:00!;@reset_start 30m!;^30m!
```

Figure 1: Splinter File Example

Splinter considers text to be both a sequence of lines and a temporal sequence of events. Any line marked with a leading "[" or "{" is a "temporal line" and will have a corresponding timepoint t_l in the STN, as well as a constraint $t_{l-1} + [d\ \infty] = t_l$—that is, the current temporal line follows the previous by at least some default *d* time units ($d = 1$ unless otherwise configured by the user). Importantly, lines not marked as temporal are not touched by the temporal propagator, allowing users to add documentation or formatting.

Constraints are added by appending a "#" character to the end of the line and separating constraint specifications with a semicolon. Each constraint specification consists of a *reference* to some other temporal line and a *distance*.

References can be to the origin, the next or previous line, a named line, and even a named line in a different file. Absolute constraints reference the origin and use the "=" character, as in lines 2 and 10. The next or previous line are literally pointed-to by the characters "v" and "^", as in line 6, which creates a 10-second delay to line 7, and line 11, which creates a 30-minute delay from line 10. Any line can be named by having an entry among the constraints that starts with an alphabetical character, as in line 2 of Fig. 1, named "palstart". Named references use the syntax "@name", such as on line 4, which creates a 1-hour delay with line 2 by name. These references can cross files by using "@filebasename:name", as on line 7.

Distances can be written using bare integers, in which case they are interpreted as the abstract time units that are the representation in the STN, or they can have unit suffixes of microseconds, seconds, minutes, hours, days, or weeks. Suffixes can be combined, i.e. "5w3u", which would be 5 weeks and 3 microseconds. Distances with units are converted to the STN time unit using downward-rounding division by a configurable base unit described in Configuration.

The underlying interval representation of distances and timepoints allows users to express flexible constraint bounds, and Splinter provides syntax for this. A distance can consist of an explicit interval, i.e. "[5 10]", an implicitly open interval, i.e. "5" is equivalent to "[5 ∞]",

or a single value with an exclamation mark, i.e. "5!" is equivalent to "[5 5]".

The constraint created from a constraint specification is dependent on the *direction* of the constraint. A constraint on t_l with a previous line t_{l-n} is always interpreted as $t_{l-n} + \delta = t_l$. Symmetrically, a constraint with a subsequent t_{n+1} is always $t_l + \delta = t_{l+n}$. This allows users to express most constraints with distances that are 0 or positive.

For constraints between files, there is no way to treat a line in one file as having any necessary relationship to a line in another, t_o. We chose arbitrarily to treat such constraints as $t_l + \delta = t_o$.

Configuration

Each Splinter file has a number of configuration options that control details of display, export, and constraint propagation.

Display and Export

- format: The format to parse and display dates and times, using the Python datetime.strpdate format specification
- time_format: The format to parse and display times
- times_as: Select between displaying times as timepoint values or date-times
- hide_date: Toggle whether the date part of each date-time is hidden
- export_date_format: Format specification for exporting an ATS

Constraint Propagation

- origin: The date and time which corresponds to Z in the STN, formatted according to *format*
- unit: The unit of time represented by a difference of 1 between timepoints, the finest granularity of the schedule. The default is 1s, but can go as low as 1µs.
- default_duration: The default amount of time between consecutive temporal lines in the splinter file with no other constraint
- commit: For any schedule that has flexibility after propagation, commit to an earliest or latest schedule to fully ground the ATS

Propagation and Inconsistency Feedback

At a user command, Splinter will parse a file that is being edited (a variant command will parse all splinter files open in the current window), construct the corresponding STN, and attempt to propagate the bounds of all temporal lines. If the STN is consistent, any text between the leading opening bracket or brace and its closing counterpart will be replaced by the computed bounds.

If the STN is inconsistent exact bounds cannot be computed but the negative cycle in the graph provides information about which timepoints and constraints participate in the inconsistency. The below figure shows a simple example:

```
1  [0 +inf] A temporal line outside the negative cycle
2  [1 +inf] The start of the problem   #start;
3  [6 +inf] The middle of the problem  #^[5 10];v[5 10]
4  [11 +inf] The rest of the problem.  #@start [1 9]
5  [12 +inf] Another temporal line outside the cycle
```

Figure 2: Highlighting Inconsistent Constraints

The constraints on line 3 create a minimum distance of 5 from line 2 and line 4, meaning the minimum distance from line 2 to line 4 is 10. However, line 4 creates a maximum distance to line 2 of 9. This is clearly inconsistent. In the STN, the inconsistency appears as this negative cycle:

Edge	Weight
$t_2 \rightarrow t_4$	9
$t_4 \rightarrow t_3$	-5
$t_3 \rightarrow t_2$	-5
Cycle weight	-1

As seen in Figure 2, Splinter highlights in red all lines that correspond to a timepoint in the negative cycle to allow the user to focus on the problematic subset of the timeline.

Export

Once a Splinter file has been propagated and found consistent, it may be exported to a different format for compilation or execution. In its current iteration, the destination format is hard-coded.

Only lines that are understood to be commands are exported—non-temporal lines and temporal lines without commands have no representation in the exported file. Any line whose non-temporal text begins with a quote and followed by either a forward slash "/" or the characters "cmd" are commands and all text between the command marker and the final quote are copied verbatim into the output file. The time of the command is taken from between the brackets or braces at the beginning of the line.

Splinter Idioms

Through the use of Splinter, the team has arrived at a few idiomatic structures to express common features of DSA command sequences.

Command Exclusion

Lines 10 and 11 of Figure 1 represent the idiom for excluding anything that might impact the schedule. Two lines mark the beginning and end of the exclusion period. The beginning line is given a name. A constraint of exact distance is created from the beginning to the following line, the reverse constraint is created from the end line to its previous line, and then a constraint with the same distance is created to the name of the beginning line. Thus, any constraint of any other distance would be inconsistent. This is largely used to represent times during which the ATS might not be executing, such as forced processor resets. This proved so useful that we added the ability to have lines with a start, end, and duration, indicated by beginning a line with "{".

Experiment Duration

When planning for experiments that needed to run for a minimum duration for any reason, we found it useful to create lines marking the beginning and end of the experiment with a constraint between them enforcing the minimum duration. When it came time to fill in the commands for the experiments, we would create 0-distance constraints from the markers to the first and last commands, respectively.

Cross-timeline Anchors

The way to express "this line occurs at least 5 minutes after a line in another file" is to create a constraint with distance $[-\infty - 5m]$. It proved quite difficult, especially when working quickly, to correctly translate before/after across timelines into distance intervals, so whenever possible we create named lines that have a 0-distance constraint among them—that is, they will have identical times—and create local constraints with these anchor lines.

Phantom Commands

One of the capabilities of DSA is cross-commanding—one spacecraft can issue a command on the crosslink network to be executed by other spacecraft. Tracking cross-commands was made much easier by creating temporal lines that would express the command being executed and be constrained to be at the time of the cross-command but not parse as a command by the ATS export facility, creating a visual and temporal representation that would not appear in the final command products.

Additional Export Applications

Once the Splinter tool was developed enough to be useful, the DSA operations team started finding new uses for the information in the STN and in the text surrounding the ATS.

Relative Time Sequences

Splinter has proved useful in the authoring of *Relative* Time Sequences (RTSs). An RTS tags each command in the sequence with an amount of time to wait before executing the command.

Splinter allows all the same editing features it does for ATSs for RTSs. It still displays the schedule, relative to the beginning of the RTS, along the left-hand side of the RTS. However, instead of exporting that schedule, it uses a feature of Tempus Fidget to compute the distance bounds between each successive pair of timepoints to generate the RTS wait times.

```
1   [0] "/validate DSA_COMM_APP.CommDefault" #v30s
2   [30] "/activate DSA_COMM_APP.CommDefault"
3   [34] "/load /cf/dsa5u2c1a2q6T1J.tbl"
4   [38] "/validate DSA_LFT_APP.LFT_CONF_TABLE" #v30s
5   [68] "/activate DSA_LFT_APP.LFT_CONF_TABLE"
6   [72] "/DSA_LFT_START"
7   [76] "/DSA_PUSH_ENABLE"
```

Figure 3: Example Splinter RTS

Time-Jumping

Testing an ATS in any convenient way requires some ability to execute the command sequence at a rate faster than 1Hz. This can be achieved by having a high-rate simulator, but is more simply achieved by taking advantage of a flight software capability to set the clock by command—"time-jumping".

Because the text on a temporal line is simply text and any text not formatted like a command is excluded from the final ATS, Splinter is free to attach additional semantics to other text. For time-jumping, a temporal line consisting of the text "SPLINTER" is interpreted as a marker that the time between the preceding line and the next can be safely skipped in testing. These pairs of times are exported to a time-jump file. If multiples of these skippable periods exist between parallel timelines, only the intersection of the periods is kept.

When running the ATSs in test, the test-runner script can monitor the spacecraft clocks for the start of a time-jump and, when that time arrives, immediately set the clocks forward to the end of the time jump. Thus, it is possible to see the execution of every command of a 24-hour ATS in much less time.

Additional Features

Command Name Aliases

Naming commands is surely the most challenging task in flight software design, if measured by how frequently it is done well. Very often, names are either unusably long and verbose, incomprehensibly abbreviated and terse or, worse, both.

Splinter allows aliasing of command names, which will then be de-aliased on export. In Figure 3, line 4, "validate" is an alias in DSA/Starling for "CFE_TBL_Validate_t".

This allows mission operators to attempt the challenge of naming commands themselves.

State Analyzer

Splinter contains a pure-Python implementation of an extremely simple discrete simulator of certain aspects of flight software and hardware state for the Starling mission, as well as a model of the preconditions and effects of a subset of the commands available.

When the user runs the analyzer, it begins with an initial state and steps through the Splinter file, applying each command to the state and then moving on to the next command. If the text of an RTS is available, it can descend into RTS invocations and analyze those.

As the state evolves with each command, the analyzer annotates the line of the command with an abbreviated representation of the state which can then be moused over to inspect in more detail. Any line with an error is annotated in red. In Figure 4, we see an example of executing RTS 139, which starts DSA's applications. The abbreviated indicator for **C**omm, **T**EC, **A**uto, and **L**FT go from gray (off) to yellow (idle).

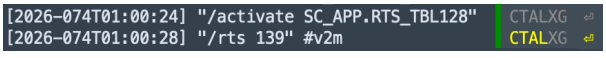


Figure 4: State Analyzer Starting Applications

Figure 5 shows part of an expanded error annotation on a line that calls RTS number 220. The error indicates that the RTS executed on that line wasn't known to be registered with the stored commanding system when it was run, so a prerequisite of running it was not met by the ATS.

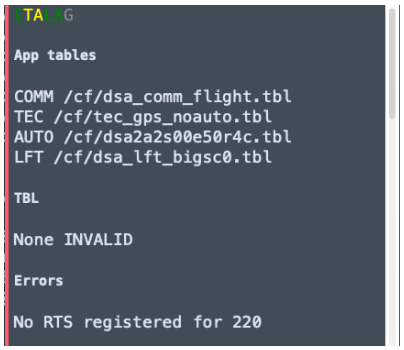


Figure 5: Expanded Error Annotation

The simple checks in the state analyzer have proved invaluable, most notably in catching sequencing errors where the precondition of a command is not satisfied when it executes.

Alternative Tools

The DSA team found time between autonomy experiments to evaluate two other tools for mission planning, APRES[9] and Aerie[10]. In both cases, we found that the problem was ill-suited to the tool.

Both APRES and Aerie are designed for problems where there are many high-level actions with many constraints, and goals are easily stated as a state to plan towards. The behavior of the system is understood and represented in the model.

The DSA autonomy experiment problem has relatively few different actions, all of them low-level, and constraints that are few but mighty. Moreover, the behavior of the system *is the experiment*. This presented a significant impedance mismatch between the way the DSA operations team thought about our problem and the way the tools wanted to represent them. It is possible that this could have been overcome, but there wasn't time.

Possible Future Work and Conclusion

Resources

Many of the constraints that are currently being represented in Splinter arise from some problem or condition of resource availability—radios, power, applications. A model of resource availability, even a simple claimable unit resource, would bring the constraints in Splinter closer to the constraints of the system. This would allow us to be able to make even clearer why a schedule looks a particular way, or why it fails.

Schedule Repair

The STN in Splinter can identify an inconsistent schedule, but it is up to the user to try to understand exactly what is wrong and fix it. An algorithm to suggest a repair step would greatly speed up timeline development.

Schedule repair is a well-studied problem with many possible algorithms, but in our experience simply moving a group of related and inconsistent commands needed to achieve a state before or after the nearest absolute time constraint would be sufficient for most of the inconsistencies we see.

Generalizing state analyzer

The state analyzer's model of the spacecraft and command behavior is implemented entirely in Python and currently only describes parts of the Starling/DSA hardware and flight software. A more data-driven model of the behavior and checks would reduce the effort to apply Splinter to other missions.

Adding this feature, however, might bring Splinter close to becoming a planning application all its' own.

Conclusion

Splinter made possible many of the successes of the DSA 1.0 mission and DSA 1.5 extension. The DSA operations team was often called upon to develop, test, and deliver 48-hour experiment timelines in a day. Less often, but still with significant frequency, we had to rearrange, drop, or otherwise re-plan, re-test, and re-deliver experiment timelines to cover a change in schedule. A tool *like* Splinter would have been necessary to accomplish these tasks in the allotted time and achieve mission success.

The performance of STNs in general and Tempus Fidget in particular is difficult to overstate. Tempus is able to propagate a fairly routine set of timelines with 378 constraints in ~4µs and a simulated timeline with 49153 constraints in ~7µs on an M4 Macbook Max. This is fast enough to fully reconstruct and repropagate the STN in real-time while the user is editing without appreciable input lag.

Acknowledgements

First and foremost, the authors would like to thank the late Paul Morris for his work on STNs in planning, scheduling, and execution, without which none of this would be possible. We would similarly like to thank the indefatigable Chuck Fry, the developer and maintainer of Tempus Fidget, whose commitment to tool development enabled Splinter to be developed on such a short timeline.

Our gratitude goes also to Aaron Woodard and Ethan Massey. Their joint work on the DSA swarm testing tool and time-jumping enabled the Ops team to deliver many command sequences with confidence and on time.

Lastly, we would like to acknowledge the Starling operations team: Randon Grashuis, Fahrin Kabir, and Jen Murphy. We could hardly ask for better partners in operations, even when they asked us to shift the timelines by 15 minutes.

References

1. Hunter, Roger, et al., "NASA's Starling Mission and Space Situational Awareness Implications", Proceedings of the Advanced Maui Optical and Space Surveillance (AMOS) Technologies Conference, held in Wailea, Maui, Hawaii, September 16-19 2025.
2. Adams, Caleb, Michael Iatauro, et al., "Advancing Autonomy in Distributed Space Systems: Results From the Distributed Spacecraft Autonomy Experiment on Starling 1.0", Proceedings of the Small Satellite Conference, held in Logan, UT, August 3-8 2024.
3. Adams, Caleb, Michael Iatauro, et al., "A Torrent-Inspired Large File Transfer System for Small Satellite Swarms", Proceedings of the Small Satellite Conference, held in Logan, UT, August 10-13 2025.
4. Dechter, Rina, Itay Meiri, Judea Pearl, "Temporal Constraint Networks," *Artificial Intelligence*, Vol. 49, 1991, pp. 61-95
5. Frank, Jeremy, Ari Jonsson, "Constraint-Based Attribute and Interval Planning", *Constraints*, Vol. 8, 2003, pp. 339-364
6. Barreiro, Javier, Matthew Boyce, Jeremy Frank, Michael Iatauro, Paul Morris, Tristan Smith, and Minh Do, "EUROPA: A Platform for Timeline-Based AI Planning, Scheduling, Constraint Programming, and Optimization", Proceedings of the International Conference on Automated Planning and Scheduling, held in Atibaia São Paulo, Brazil, Jun. 25-29, 2012
7. "Sublime Text", https://www.sublimetext.com/, accessed Jun. 8, 2026.
8. "pybind11 (v3) — Seamless interoperability between C++ and Python", https://github.com/pybind/pybind11, accessed Jun 8, 2026.
9. Bresina, John, Vijayakumar Baskaran, J Benton, et al., "APRES Prototype Mission Planner System Demonstration", Proceedings of the International

Conference on Automated Planning and Scheduling, held in Prague, Czech Republic, Jul. 8-13, 2023.

10. Dailis, Matthew, Eric Ferguson, Chris Camargo, and Adrien Maillard. “Aerie: A Modern Multi-Mission Planning, Scheduling, and Sequencing System”, Proceedings of the International Workshop on Planning and Scheduling for Space, held in Prague, Czech Republic, Jul 7-8, 2023.