The Challenge of Configuring Model-Based Space Mission Planners

Jeremy D. Frank[**], Bradley J. Clement[*],
John M. Chachere[***], Tristan B. Smith[+] and Keith J. Swanson[**]

[*]Jet Propulsion Laboratory, California Institute of Technology, [***]SGT Inc, [+]MCT Inc., [**]NASA Ames Research Center
{FirstName.MiddleInitial.LastName}@nasa.gov

Abstract

Mission planning is central to space mission operations, and has benefitted from advances in model-based planning software. Constraints arise from many sources, including simulators and engineering specification documents, and ensuring that constraints are correctly represented in the planner is a challenge. As mission constraints evolve, planning domain modelers need help with modeling constraints efficiently using the available source data, catching errors quickly, and correcting the model. This paper describes the current state of the practice in designing model-based mission planning tools, the challenges facing model developers, and a proposed Interactive Model Development Environment (IMDE) to configure mission planning systems. We describe current and future technology developments that can be integrated into an IMDE.

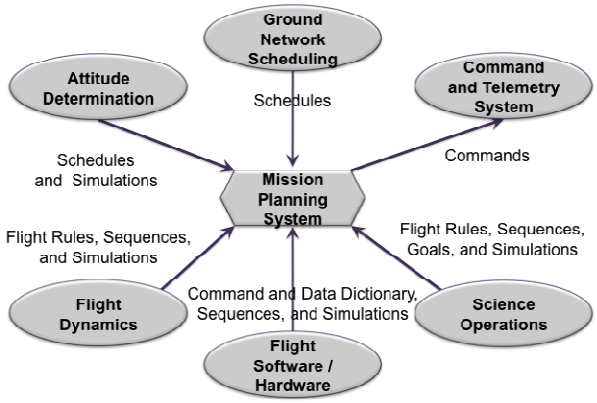
Introduction

Mission planning is central to space mission operations, and has benefitted from advanced in model-based planning software (Reddy et al. 2010, Bresina et al. 2005). A principal obstacle to fielding model-based planning systems for space missions is the complexity of domain modeling. Space mission planning has diverse information sources such as engineering specification documents (Barreiro et al. 2010), communication coverage, simulations of spacecraft subsystems (Ko et al., 2004; Yen et al., 2005), and trajectory and attitude specifications. Building a domain model requires identifying these information sources, understanding them, and often representing formal abstractions of them.

Changes to the constraints throughout the mission's development require changes to the model. Detecting and managing discrepancies between the models used to generate plans and the constraints increase mission cost, schedule, and risk. A discrepancy may indicate an error in modeling that must be fixed prior to operations. If these discrepancies are detected only during operations, they require significant manual effort to fix. If undetected, this can lead to a command sequence harming the spacecraft. These errors are difficult to avoid because models are often developed as disconnected abstractions of the system and are difficult to check against the sources of the operational constraints.

This paper first describes space mission and activity planning in the context of other mission operations system elements. We use a sample activity description to show how an activity's pieces are constructed in a declarative domain model from the various information sources. We then challenge the community to address the complexity in the context of a proposed Interactive Model Development Environment (IMDE) that simplifies the construction, validation, and maintenance of automated planning systems. The latter half of the paper describes a proposed

Figure 1. How the Mission Planning System interacts with the Mission Operations System.



IMDE from a functional and architectural perspective. We describe both current and near-term technologies that can be used to build such an IMDE. The survey of technologies to aid mission planning begins with those in use or imminently available. The survey concludes by outlining research that could produce valuable mission planning technologies.

The Mission Planning Process

A mission's planning systems reside in a context of mission planning processes. In particular, constraints are central to mission planning systems, and many of these constraints come from the mission operations system.

Mission Operations System

The *mission operations system* (MOS) is the integrated system of people, procedures, hardware, and software that executes space missions (Carraway et al. 1999); recent examples are described in (Garcia et al. 2009) and (Tompkins et al. 2010). The MOS has several planning functions. *Mission planning* decides how and when the spacecraft and subsystems will act. *Activity planning* (or, sometimes, *sequencing*) is creating or enabling specific command sequences, either onboard the spacecraft or in a ground station. Attitude determination and flight dynamics planning (which are typically distinct from mission and activity planning) determine where the spacecraft is and where it maneuvers. Communications planning (another distinct discipline) determines who to communicate with and when. Communications planning critically depends on flight design and the availability of communications assets. Mobile surface missions like the Mars Exploration Rovers include a planning system for surface operations (Ko et al., 2004; Yen et al., 2005). Science targets, science instrument or payload constraints, and preferences for science payloads and instruments are typically input to mission and activity planning. Attitude and flight dynamics, communications, surface operations, and science planning provide input to mission and activity planning, but can also be constrained by them.

Before commanding the spacecraft, mission operators typically transform plans into sequences that simulators and other tools validate (Ko et al., 2004). Finding discrepancies in this process affects cost, schedule, and risk.

Configuring the Mission Planning System

Configuring the mission planning system involves identifying planning problems, methods to solve those problems, and ways to communicate sequences derived from the plan to the command and telemetry system (for uplink to the spacecraft or execution on the ground). We focus on the first of these issues: describing planning problems. Model-based planning experts know the 'right way' is to build a declarative planning model. However, the sources of space mission constraints present challenges to model building. Figure 1 illustrates a mission's interacting operations and planning systems. The purpose of this view is to explain how MOS components interact during the mission and how they influence the mission planning system's design. The remainder of this section details the sources of Mission Planning System constraints.

Flight Rules.
Flight rules and other operational constraint products document constraints and best practices for system operations to ensure mission safety and mission success (Barreiro et al., 2010). Instrument teams, spacecraft manufacturers, and sometimes the mission operations team create these documents. These rules provide essential planning system input, but are typically stored as human-readable (office) documents. Over time, missions have evolved a set format for these rules. A typical flight rule (below) shows features that are common in operational constraints: the rule is broken up into discrete parts, the action maps to fine-grained commands in multiple ways; the rule's criticality indicates it could be waived; the action duration is explicit; and the mission phase dependency demonstrates rules that only apply in certain contexts.

Instrument Rule 1
Rule: To power down, close the cover (Inst-Close-A or Inst-Close-B), do not issue any further CMDs, wait at least 35 seconds, and then issue the power down CMD (PDU-1-Power-Down-Inst or PDU-Power-Down-Inst).
Rationale: When not in use, the cover must be closed for protection from Sun. Instrument needs to be powered during the 35 seconds it takes to close cover.
Criticality: Category B
Mission Phase Dependency: Pre-launch, Cruise, Orbit
Commands Affected: Inst-Close-A, Inst-Close-B, PDU-1-Power-Down-Inst or PDU-Power-Down-Inst
Cognizant Individual: Instrument Operations Contact
Notes: If the cover-close command is issued when the cover is closed, the cover remains closed, and the command is rejected. Once the closure procedure is started, it is not possible to interrupt it.

Sequences.
Sequences are lists of fine-grained spacecraft commands. Operators command the spacecraft by executing sequences on the ground, by sending them to the spacecraft for immediate execution, or by storing them onboard the spacecraft to await a later event or command trigger. Simulation is used to determine sequences' time and other resource constraints. The exact simulation used depends on the sequence's origin. For instance, instrument teams may simulate their instruments (Tompkins et al. 2010). A spacecraft manufacturer or mission operations team also may build a simulator (Yen et al., 2005). Often, simulations are used solely to check sequences against flight rules (Ko et al., 2004).

Orbit Design and Communications.
Flight design may simulate orbits using a commercial product like Satellite Toolkit (Tompkins et al. 2010). Orbits provide key information for mission planning

systems. Examples include day / night times, sun angles, and the relative locations of asteroids, comets, and communications assets.

When Constraints Change

Constraints can change greatly before a mission. Mission planning systems must accommodate these changes and be validated at low cost (Carraway et al. 1999). For example, target changes (as happened on LCROSS (Tompkins et al. 2010)) may require orbit changes, which can ripple further through the planning systems. Changes in communication coverage can cascade in a similar manner. While these changes may appear to be 'mere' changes in plans, if communication windows shrink, rules or constraints governing communication coverage times may need to change as well.

Changes to vehicle configuration (specific equipment, interconnection or equipment location, or equipment performance characterization) can also cause changes in mission planning constraints. Examples include new flight rules, science instrument sequence changes, changes in maneuvers, or new power or thermal limits (Tompkins et al. 2010).

As mission planning systems mature, planners often find satisfying the constraints is too difficult. For example, science teams and spacecraft manufacturers can provide overly conservative constraints early in mission development. On examination, analysts may determine the constraints can be relaxed without compromising safety or science.

The Challenges of Configuration

Space mission planning has benefited from the use of model-based planners. However, there is a long-standing, fundamental problem in applying automated planning to physical systems. This difficulty stems from developing system models that are disconnected from the system (which lead to inaccuracy) and from modeling representation language limitations (which add to complexity). In order to make this concrete, consider the difficulty of representation of a relatively simple spacecraft activity: changing attitude.

Assumptions

We will illustrate the difficulty of modeling for space mission activity planning using the following assumptions:

- There is a spacecraft command and data dictionary. For simplicity, this section assumes the data dictionary includes orbit and attitude information.
- The simulator input includes a list of time-tagged commands from the command dictionary.
- The simulator runs deterministically.
- The simulator reports any errors (undesirable behavior);
- The system (spacecraft) and simulator are defect-free.
- The simulator is a black box (we can neither change nor inspect its code and models)
- The simulator outputs time-tagged samples of values of specific data from the data dictionary.
- Formal flight rules define mission constraints that are verifiable with the simulator output.
- A declarative modeling language configures the activity planning system, e.g. PDDL (Fox & Long, 2003).
- Every plan that the planner sends to the simulator is consistent with the planner's model.
- A plan action corresponds to a list of time-tagged commands from the command dictionary.
- The planner is sound but not necessarily complete.

An Example

```
(:durative-action slew
 :parameters (?from - attitude
              ?to - attitude)
 :duration (= ?duration 5)
 :condition
   (and
     (at start (pointing ?from))
     (at start (cpu-on))
     (over all (cpu-on))
     (at start (>= (sunangle) 20.0))
     (over all (>= (sunangle) 20.0))
     (at start (communicating))
     (over all (communicating))
     (at start (>= (batterycharge) 2.0)))
 :effect
   (and
     (at start (decrease (batterycharge)2.0))
     (at start (not (pointing ?from)))
     (at end (pointing ?to))))
```

The PDDL above configures the mission planning system for a spacecraft attitude change activity. This activity model is more abstract than simulators of the spacecraft's command set, lighting conditions (a function of the orbit), dynamics of slewing the spacecraft, communication asset locations, spacecraft power utilization, and battery performance. Typically, extracting knowledge to configure the planning system is manual, inefficient and error-prone. Questions often include:

- How do planner model attitudes relate to real spacecraft operations' continuous attitudes? For example, a deep spacecraft with camera directional sensors may require a discrete set of target attitudes only. Often, there are designated pointing attitudes. Examples are to Earth (for deep-space), Earth-nadir (for Earth orbits), and sets of navigation guide stars.
- How does the planner estimate battery discharge? Evaluating the spacecraft components is essential. Planners must account for all components that are active during the slew operation, which may require consulting the simulation.

- What drives slew duration? One factor is angular slew distance. Another factor is the attitude control system (reaction wheels, thrusters, torque rods). The spacecraft manufacturer characterizes that system in performance tests. A third factor is spacecraft sequence (see below). Flight rules may govern slew sequences, and simulation may characterize the sequence duration and resource use.
- How does the planner model communication coverage? Coverage is a function of the spacecraft orbit, communication assets, spacecraft antenna type and configuration, many of which simulators analyze pre-flight. Flight rules could require spacecraft communications because ground systems must monitor spacecraft activities or command the slews.
- How does the abstract slew correspond to one or more sequences of spacecraft commands? Some spacecraft decompose attitude changes into a sequence of rotations in each major body frame (pitch, roll, yaw). 'Arbitrary' slewing is possible, but limiting slews to one axis at a time is simpler and hence safer. Mission planning can't easily check some flight rules (e.g. voltage limits in the power system) and are checked by simulation. It is especially important to check for unexpected interactions between concurrent sequences.

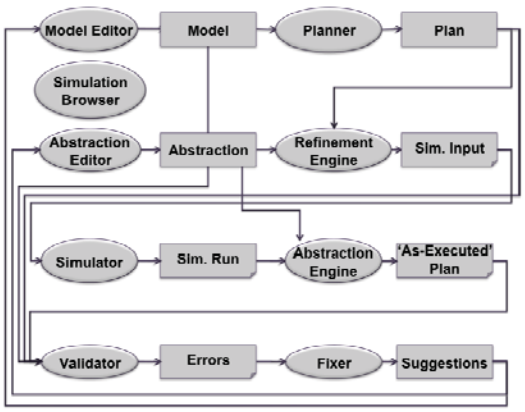


Figure 2: Hypothetical IMDE architecture.

Before flight, the orbit, attitude, engineering subsystem specification and simulations can change frequently. These changes require reconfiguring the activity planner efficiently. For example:

- New targets or navigation aids require updating the set of discrete attitudes.
- Analysis or changes in sequences can cause a change in attitude control system performance, leading to activity changes.
- Any power-using subsystem that changes performance (e.g., attitude control system or communication) will change the power consumption. If planning determines mission objectives can't be met, a need to slew faster could also increase power consumption
- Changing orbit, communication coverage plan, or antenna configuration may change the activity.
- Changing flight software, or the uses of major spacecraft operating modes, might require changing the commands that affect attitude

Clearly, configuring the mission planning system with even the one activity described here requires much effort. The effort includes extracting knowledge from the flight rules, command and data dictionaries, and simulation APIs and output. Currently, those data (and input from the mission operations system orbit, trajectory and communications elements) often reside in documents that are difficult to extract planning knowledge from.

Interactive Model Development Environments for Space Mission Operations Applications

The activity planning system's main goal is to verify activity plans against the relevant constraints. However, validating the planning system also requires validating the constraints themselves.

Academic planning languages and algorithms originally used Boolean state variables only. Such variables are generally impractical for representing time, location, and other numerical states. Planning languages are more expressive now (Howey et al. 2004, Fox and Long, 2003) but their limitations still force inelegant workarounds that make system models complex. Models strongly influence the performance of automated planning, so revising the model to improve performance can increase the complexity further. This complexity can combine with human error and lack of information about the modeled system's behavior to produce inconsistencies (with the model and the modeled system). Finding the inconsistencies can require significant work, and fixing these inconsistencies can require significant changes.

To explain the challenge problem intuitively, this paper describes how a hypothetical Integrated Model Development Environment (IMDE) could integrate planning and simulation. This integration could simplify validation of models within the development cycle, thereby making modeling for space mission planning more efficient. A more specific goal is integrating a planner and simulator to help automate model development and validation. The goal is not conventional model checking, where the model by itself is checked for potential problems. Instead, the model is being checked for consistency with the simulator.

The hypothetical IMDE design features and architecture are described in the next section. It then describes two use cases: developing a mission planning domain using

simulation and operating constraint specifications, and checking the specification for errors.

IMDE design features

The hypothetical IMDE differs from programming language IDEs. It aids some tasks that are specializations of software development tasks. The IMDE also includes features for validating a planning domain model against a simulation or operations constraints.

The simulation specification (which is synonymous with the command and data dictionary) gives the domain modeler a place to start. In a perfect world, the domain model would map 1-1 to the dictionary. More often, though, the mission planner is an abstraction of lower level commands. Also, different types of mismatches between the planner model and source information indicate different types of links between planner, simulator, and constraints.

Figure 2 shows the system architecture of a hypothetical IMDE. The Model Editor provides traditional IDE functions. The Simulation Browser lets users access the command and data dictionary while creating models. The Abstraction Editor enables describing how plan model building blocks (objects, states, timelines, actions, constraints) relate to the simulation specification. The Abstraction and Refinement Engines integrate the planner and simulator. The Refinement Engine transforms a plan into simulator input. The Abstraction Engine transforms simulator output into an 'as-executed plan' to be compared with the original plan. The Validator compares the two plans and assesses discrepancies. The Validator also takes as input any constraints not explicitly checked in the simulator (e.g., flight rules) that are part of the planning model. The Plan Viewer (not shown) visualizes (e.g. in a Gantt chart) the plans generated for a planning problem, as well as simulator output that should correspond to the plans. The Plan Viewer (and/or a separate Error Viewer) shows mismatches between the plan and simulation (as described earlier). Finally, the Fixer analyzes mismatches between the plan and simulation output and identifies model and abstraction elements for possible change. (Checking simulation output against flight rules is not shown but also a useful function.) Since the simulator is a black box, there is no guarantee that affecting the suggested repairs will fix all problems.

IMDE use cases

The hypothetical IMDE's user first creates a model. Doing so requires deciding on the objects types and sets, fluents, actions, and action conditions and effects (and, for modeling languages that support them, timelines). In this process, the user can access the simulation interface as well as output produced directly from the simulation. A typical workflow for this phase of modeling is as follows:

1. User opens new model
2. User browses simulation interface
3. User either:
   a. Copies variables to the model (e.g. `sunangle`)
   b. Abstracts variables in the model (e.g. an x,y,z attitude as a member of the planner object `attitude`)
   c. Copies one command to model as an action (e.g. turn on CPU)
   d. Abstracts a command sequence to model an action (e.g. command sequence to slew spacecraft)
4. Repeat

When the user performs these operations, they can document the relationship between the planning model and elements of the simulation using the Abstraction Editor, as shown in Figure 2. This provides traceability so that elements of the model are 'grounded' in the simulation, and as we will see below, provides a means of detecting problems when things go wrong.

Adding a new variable or timeline to a domain model requires informing the Plan Viewer. The Plan Viewer also must maintain a consistent view of the plans. It is impractical to regenerate all of the plans from first principles every time the underlying models change. So, an established policy must address stored plans generated using older models. A typical plan repair strategy might work well for 'scheduling' errors in older plans. But it may take a lot of work to indicate what must be fixed when an older plan's timelines, semantics, and state and object names change.

The hypothetical IMDE user next compares planner-generated plans and predictions to corresponding simulator output. This phase could follow the following workflow:

1. User edits initial state in planner
2. IMDE generates and tests possible plans
   a. Translates planner initial state into simulator input
   b. Translates the plan into simulator commands
   c. Runs simulation
   d. Translates simulator output to an 'as-executed' plan
   e. Checks for errors against constraints
   f. Checks for discrepancies between planner's plan and 'as-executed' plan
3. User views plan, 'as-executed' plan, and errors
4. IMDE suggests model changes based on test results
5. Repeat

Translating plan information as abstraction and refinement

The abstractions captured during initial plan authoring are heavily used in the previous workflow. For example, one `slew(?from,?to)` action in the plan may translate into

three ordered subsequences in the simulator that change different spacecraft attitude dimensions. Another abstraction type specifies how state variables in the planning model relate to those in the simulator output. One abstraction could define the mapping from spacecraft environment to attitudes: $\{px,py,pz\} \rightarrow$ `p` $\in$ `domain(attitude)`. Another abstraction could map the `at(?to)` predicate to indicate whether the simulated attitude of the spacecraft is near a target attitude. Spacecraft attitude control systems often have dead-band modes. Different modes trade pointing accuracy for propellant, power, and computer usage. Different targets (e.g., communications and science) require different pointing accuracies. So, the mapping from simulator output $\{x,y,z\}$ to the desired attitude $p=\{px,py,pz\}$ would be $\|\{x,y,z\},\{px,py,px\}\| < d \rightarrow$ `at(?to=p)`[1]. In general, an abstraction could be any function of a set of time-varying variables to a time-varying variable.

The Refinement Engine translates initial state information using these abstractions in Steps 2a and b of the process. The abstractions captured by the user are 'reversed' in order to do so. For example, translating a plan's `slew(?from,?to)` action into simulator commands would translate the symbol `?to` to the corresponding simulator object's coordinates.

The Abstraction Engine translates simulation results into 'as-executed' information using the abstractions. There are two plan model references: one for generating the prospective plan and another for comparing the planning constraints to simulation results that are transformed into the 'as-executed plan'. The planner then compares the 'as-executed' to the original plan predictions, possibly revealing numerous errors. The next section explains this process.

Identifying modeling errors

Unexpected simulator behavior can indicate modeling errors. For instance, commands may fail due to gaps in the mapping from action to command. To illustrate, `slew(?from,?to)` could fail because its decomposition lacks necessary power-up commands, because commands are improperly ordered, or because it occurs in an unexpected and incompatible system configuration. Checks on the simulation output (e.g. those assessing flight rule violations) can also detect these problems.

Modeling errors can also result in simulator output having constraint violations that the original plan lacks. For example, in testing the `slew(?from,?to)` action, suppose that a violation occurs because the simulation achieves `at(?to)` too late. The modeling error may be that the spacecraft turns slower than expected along one axis.

In another error detectable at this stage, the power system simulator predictions of power consumption may not match the planner's power consumption model. A planning constraint, such as a minimum battery state of charge, might appear violated when comparing the simulation and the planner model. Yet, unlike in the previous example, no action precondition or effect fails.

Finally, a host of errors may arise in checks of simulation output against flight rules. For example, power bus voltage limits, dynamics constraints, structural load limits and other properties may be specified in flight rules or other engineering documentation, but must be checked against the simulation output variables describing those quantities as a function of the plan.

Discrepancies between the planner and simulator state value predictions need not be modeling problems. Defining the planning states as abstractions of the simulator's states could naturally lose information. For example, the planning model could represent battery depletion as instantaneous while the simulation represents depletion as gradual. Discrepancies will probably manifest between the planned and simulated battery levels. But, planning the battery levels conservatively could avert simulation failures. The user may choose to omit specific discrepancies from reporting (as with waiving constraint violations in mixed initiative planning systems – see Aghveli et al. 2007). However, the discrepancy might indicate an efficiency improvement opportunity: a more detailed battery depletion model could enable scheduling more activities.

An alternate workflow can check a plan against pre-existing simulator output. This method needs not translate planner initial states and plans to generate simulation runs. Instead, the method compares a plan's commands and predicted state with corresponding, pre-existing simulation output. In this case, errors may result from different commands or orderings in the plan and the simulation.

Generating plans to validate pieces of the model

Abstractions generally lead to loss of information. There may be many possible valid simulations that can be abstracted to the same plan. The reason for generating different plans to test is to validate that the model will work for all situations. Validating the model requires validating all possible plans that can be constructed from the model. There may be a manageable number of simulations that is enough to validate a single part of the model. For example, if the user wants to ensure that the `at(?to)` effect is always satisfied at the end of `slew(?from,?to)` then a complete space of plans to test would combine all possible attitudes (slew from each

[1] Strictly speaking, evaluating the at(?position) abstraction on simulation ouput requires first evaluating the ?position abstraction.

attitude to each other attitude) and all possible initial states for `slew(?from,?to)`. It is possible to generate all of these plans with special purpose code, but the planner itself may be leveraged to accomplish this. Instead of generating all combinations, incorporate this parameterization into a planning problem: what initial state and ordering of instantiations of `slew(?from,?to)` will achieve `at(?to)` at the end? The set of valid solutions to this planning problem is the test suite. Unfortunately, this kind of test coverage problem is known to be quite difficult and, thus, part of the challenge.

A single plan may lead to many (possibly infinite) sequences to simulate. This situation is apparent from the first flight rule described, in which several command sequences could be used to power down an instrument. Again, it may be possible to cleverly scope the validation to reduce the number of sequences tested.

Suggesting changes to the model

When the IMDE runs a batch of plans through the simulator, some may result in simulator errors and some may result in planning constraint errors. These indicate that there are modeling errors, but the actual mistake made by the modeler may not be obvious, especially if the relevant system variables are not sampled very often or not part of the simulator output at all. The IMDE can suggest methods to fix inconsistencies between the plan test cases and corresponding simulator output. For example, suppose a spacecraft failed to reach its destination attitude whenever the z-axis angle change exceeded 120 degrees. This condition could be added to the planning model. IMDE suggestions could include changing constraints on an action, adding state variables, or creating new actions, as described by the Fixer component in Figure 2.

For example, suppose the `slew(?from, ?to)` action never achieved `at(?to)`, but the simulation output indicates that the spacecraft never stopped moving and in fact oscillated around a deadband (as described earlier). In this case, there was never a value reported in the results that the spacecraft attitude was in the vicinity of `?to` because the deadband mode was larger than the modeler expected. The fix may not actually be to the model but to how simulator results are interpreted in the abstraction (the modeler chose the wrong deadband to map attitudes to the `at(?to)` value), or the action abstraction was wrong (the attitude controller invoked was the wrong one for the chosen attitude and the data abstraction was correct), or that all the abstractions are correct but the modeler chose the wrong action (i.e. there are multiple `slew` actions).

Can we automatically identify where the problem is? We could try to classify the conditions under which the errors occur. If we look at this as a machine learning problem with the simulations of plans as the sample set, we could try and learn a function of plan and simulation output to whether an error was found. Part of that function may be that whenever the rotation around the z-axis is greater than 120 degrees, `|deltaZ|` > 120, `slew` fails. A direct fix to the model would be to bring `deltaZ` into the planning model and to add a precondition to the drive action that `|deltaZ| <= 120`. The user could accept this fix and replace the `deltaZ` inequality with a Boolean `longSlew` abstraction in order to avoid numerical states. Note this approach works regardless of whether there are simulation errors, post-simulation checks to determine flight rule violations, or discrepancies between the simulation and the plan. The learning problem is difficult because simulation output may vary in size for the same plan, and because it may require learning complex functions of the simulation data and abstractions.

Technology Foundations

While the ultimate vision of the IMDE has yet to be achieved, many component technologies have been built. In this section we describe some of these technologies as well as research activities that enable this goal.

To understand better how the proposed automated model development and validation may be used, consider the validation of the CASPER automated planning system for onboard commanding of NASA's EO-1 spacecraft. This validation process involved tabletop model reviews with EO-1 engineers and operators, safety reviews to elicit potential hazards, and automated tests stochastically generated as perturbations to nominal scenarios and executed on simulation platforms of varying fidelity where spacecraft, operations, and safety constraints were checked (Cichy et al., 2004). The automation proposed here may not be able to eliminate any of these steps. However, tests could be generated and executed for each edit to the model to identify, avoid, and fix modeling errors. This testing could make the reviews simpler since plans have already been validated for a documented set of constraints by the simulators. The reviews could focus on what rules have been checked instead of how they are being modeled.

The itSimple tool (Vaquero et al., 2007) is a plan domain modeling environment very similar to the proposed IMDE. Users of itSimple can build 'static' models of objects, actors and relationships between them in a specialization of UML, and 'dynamic' models of how states of the objects are allowed to change using Petri Nets (an encoding of state charts); the Petri Net model acts as a simulation. The resulting models are automatically translated by itSimple to PDDL, after which the users can continue refining the resulting models. This ensures commonality between the primitives used in the simulation and in planning, and automates the translation from well-

known engineering formalisms used in the software world to declarative planning models. In order to be used for space mission planning, the itSimple approach must be able to create planning models with time, resources, complex numerical state variables, and complex resource constraints; the simple Petrie Net formalism is not expressive enough to do this. Furthermore, the assumption that the simulation is captured by the plan domain modeler does not always match the space mission domain, where simulations are often black-box (although the approach can be quite valuable when a white-box simulation serves as the root of a planning model.)

The Procedure Integrated Development Environment (PRIDE) (Izygon et al., 2008) is a procedure authoring technology prototype that can be used to create procedures for execution by flight controllers and crew. PRIDE presents procedure authors with a command and telemetry database characterized in XTCE (Simon et al., 2004); users can drag commands and telemetry references into a plan directly from the command and telemetry database GUI. PRIDE provides access to either state-chart simulations or high-fidelity simulations the procedure writer can use to manually check procedures for correctness. Procedures can also be automatically verified by means of translation to Java and the use of model checking software (Brat et al., 2008). The use cases for creating procedures are quite similar to the assumptions made here. However, procedures use the command and telemetry dictionary directly, with no abstraction, and 'models' are limited to including verify steps for post-conditions that can be checked by the procedure writer. There is also no notion of automatically proposing fixes to plans.

The Constraint and Flight Rule Management System (ConFRM) (Barreiro et al. 2010) is a flight rule authoring environment. ConFRM provides features to create and maintain links between related operational constraints, between constraints and source data such as simulations or engineering analysis documentation, as well as numerous IDE-like features to reduce the effort in creating and authoring operational constraints. Like PRIDE, ConFRM is intended to allow mission operators to create flight rules and operating constraints while browsing the command and telemetry specification of the spacecraft. ConFRM prototypes included the ability to export declarative constraints to tools such as a planning system. ConFRM's link management functionality is similar to capability needed by the proposed IMDE. Providing tools like ConFRM to the organizations providing source data to the planning system can further reduce the effort in ensuring flight rules and related constraints are created for easy integration with the planning system.

The Data Abstraction Architecture (DAA) (Bell et al., 2010) is designed to address the problem of transforming spacecraft or space system telemetry into useful information for operators (be they flight controllers or crew). The system allows system operators to write common data transformations using a GUI; the transformations are then executed by an engine that accepts telemetry as input, and produces more intuitive information as output. The DAA framework is well suited to specify a number of abstractions needed for transforming simulation variables and values into plan domain model variables and values; however, it would need to be extended to capture transformations of commands to plan domain model actions.

VAL takes steps toward the Fixer IMDE element by validating that a specific plan is indeed a solution to a planning problem that may be specified with continuous effects, including limited forms of time-dependent change on numerical state variables (Howey, et al., 2004). VAL can also advise modelers how to fix a plan. The goal explored here is how to validate that all plans execute as intended and suggest fixes to the model, not just the plan. Furthermore, the approach in VAL would have to act on the 'as-executed' plans abstracted from the simulator.

The LOCM system (Cresswell et al. 2009) learns planning domain models from sets of example plans. Its distinguishing feature is that the domain models are learned without any observation of the states in the plan or about predicates used to describe them. This works because of some restrictive assumptions about the form of the model describing the domain. In particular, the objects are grouped into sorts, and the behavior available to objects of any given sort is described by a single parameterized state machine. LOCM is the latest in a number of plan domain learning systems that could be employed to abstract black-box simulations into domain models as part of the Fixer in our proposed IMDE. However, doing so requires plans to begin with, and hence requires interacting with simulators to create those plans. Furthermore, learning directly on the simulation representation will not abstract the lower level commands, and will also not account for constraints manifested in flight rules. While learning could be the mechanism to determine how to design and implement the Fixer as proposed in the IMDE, it is unclear precisely how to approach this aspect of the problem.

Bonasso et al. (Bonasso et al. 2010) describe a means of eliciting plan domain information from a human expert. They assume a domain ontology (things and their relationships) is available; they then analyzed common goals in an International Space Station Extra Vehicular Activity (EVA) domain, and constructed an interactive questionnaire that users could fill out in order to construct an action description that achieves a goal. The structure of the questionnaire conforms to the structure of the action modeling language but does not force users to create descriptions in an unfamiliar format. Such an approach can be important as an adjunct to building flight rule

documents that drive plan domain modeling. Important gaps still to be resolved include deriving the ontology, and whether and how the system can be used to revise previously created actions. Apropos to the validation issues described, the approach allows users to distinguish the purpose of an action (or constraints on its success) from side effects that are known to take place and may be of interest to others.

Challenges in relaxing the assumptions

The usefulness of the described IMDE may still be insufficient because of some limiting assumptions we made. We describe those we believe are important and present additional challenges.

It is possible that an action may correspond to multiple commands, a loop, or any arbitrary function generating commands. As long as this function is a legitimate simulator input, then this is not a difficult problem.

Many systems have uncertain behavior, for example, stemming from attitude and temperature control. If the simulation testbed can be invoked in a way that explores different outcomes, then a single plan now corresponds to multiple (possibly infinite) test cases for which the model should be validated. This presents an additional difficulty in determining a tractable number of test cases sufficient for validating the model. It also presents a problem of how to model the activity correctly; if the action duration varies between 30 and 40 seconds, what is the best value to use?

In addition, the spacecraft may be able to execute sequences conditioned on the perceived system state. This requires simulations that incorporate all possible perceived states that could influence the plan outcomes.

We have discussed some basic examples of modeling errors on preconditions and effects. For expressive language elements such as activity decomposition (as opposed to the abstractions mapping plan actions to sequences of simulator commands) and parameter dependency functions, how can errors and fixes be automatically identified?

Relaxing other assumptions may not pose difficult research challenges but can change the nature of the system capability. For example, if the simulator or spacecraft does have defects (which we must always assume is true), then discrepancies that are inconsistencies between planned and executed behavior may now (in addition to modeling errors) also be indications of those defects. So the system now can identify simulator and system defects and help validate them against the planner. Thus, the IMDE may more generally be designed for validating multiple systems against each other. This validation is especially important for interactions between autonomous spacecraft subsystems (such as an onboard planner or a guidance, navigation, and control system).

Another assumption was that the simulator is a black box. One option is to treat the effects of an action as properties that are input to a model checker, which is used to directly analyze the simulation model. The System-Level Autonomy Trust Enabler (SLATE) validates a complete model of the system and its operation, incorporating device, control, execution, and planning models (Boddy et al., 2008). The conventional approach of building a model only at an abstract level requires much extensive testing of different scenarios and could only be guaranteed to work if all possible scenarios are tested. SLATE only requires testing of individual behaviors whose performance envelopes are incorporated into the model. Thus, since the model of the system is complete, it can be used to prove system-level properties similar to model checking. Another strategy for validating plan abstractions (in particular, those of hierarchical plans) is to summarize the potential constraints and effects of the potential decompositions of each abstract action in the model (Clement et al., 2007). A planner can use this summary information to create a plan whose actions are detailed to different levels necessary to conclude that all further refinements of the plan are either valid or invalid. Like SLATE, summary information validates higher level actions composed of more detailed validated actions. Summary information differs in that abstract actions retain choices of refinement for flexibility of execution, while abstract actions in SLATE are robust to uncertain system behavior. Instead of validation through testing like SLATE, summary information relies on an accurate, fully detailed model and is, thus, valuable for validating an action model based on its decomposition into a white box simulation model that can be summarized. For example, the `slew` action's effect `at(?to)` would be validated if its summarized effects included `must at(?to)`, meaning that `at(?to)` is an effect of all possible ways to execute the slew.

A more aggressive approach is to automatically abstract the simulation model to create the planner model, i.e. augment the approach of itSimple (Vaquero et al. 2007) to translate more expressive models to declarative planning languages. Automating such translations requires a deep understanding of the simulation modeling language, and may not be feasible for all simulation approaches.

Conclusion

The maturation of model-based planning provides an opportunity to improve the state of the art in space mission planning. However, doing so will require planning models to represent complex constraints derived from many sources of information, and for spacecraft engineers to be able to build and validate these models. We have

described the challenges in doing so, and described the Interactive Model Development Environment as a means of reducing up-front errors as well as providing numerous tools to catch and repair errors in building models. While the technologies described in the previous section support features of the described IMDE, there remain significant research challenges to achieve the overall vision.

- How can a complete but tractable space of test cases be identified for activity model validation?
- Can a single test case contribute to the validation of multiple model elements?
- How can errors in different modeling language features, command refinement, and data abstraction be clearly identified based on simulation output of these tests?
- What are the features of a learning problem for classifying an error?
- How can suggested fixes be generated for these errors?

Acknowledgements

The authors would like to thank members of the LCROSS and MER mission team for their insights into planning and operations for those missions. Some of the research described in this paper was performed by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government, or the Jet Propulsion Laboratory.

References

Boddy, M., Carpenter, T., Shackleton, H., Nelson, K. System-Level Autonomy Trust Enabler (SLATE), In Proc. of the U.S. Air Force T&E Days, AIAA, Los Angeles, CA, Feb, 2008.

Cichy, B., Chien, S., Schaffer, S., Tran, D., Rabideau, G., Sherwood, R. Validating the Autonomous EO-1 Science Agent International Workshop on Planning and Scheduling for Space (IWPSS 2004). Darmstadt, Germany, June 2004.

Howey, R. and Long, D. and Fox, M. VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL. In: 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17, Nov 2004.

Cresswell, S.; McCluskey, T. L.; and West, M. M. 2012. Acquiring planning domain models using LOCM. Knowledge Engineering Review, to appear.

Bonasso, P. and Boddy, M. Eliciting Planning Information from Subject Matter Experts. Proceeding of the Workshop on Knowledge Engineering for Planning and Scheduling, 2010.

Tompkins, P.D., Hunt, R., D'Ortenzio, M., Strong, J., Galal, K., Bresina, J., Foreman, D., Barber, R., Munger, J., and Drucker, E. Flight Operations for the LCROSS Lunar Impactor Mission. Proceedings of AIAA Space (SpaceOps) 2010.

Ko, A., Maldague, P., Page, D., Bixler, J., Lever, S., and Cheung, K. M. Design and Architecture of Planning and Sequence System for Mars Exploration Rover (MER) Operations. Proceedings of the AIAA Space Conference 2004.

Yen, J., Cooper, B., Hartman, F., Maxwell, S., Wright, J., Leger, C. Physical Based Simulation for Mars Exploration Rover Tactical Sequencing. Proceedings of the IEEE Conference on Space Mission Challenges, 2005.

Reddy, S., Frank, J., Iatauro, M., Boyce, M., Kurklu, E., Ai Chang, M., Jonsson, A. Planning Solar Array Operations on the International Space Station. Special Issue on Applications of Automated Planning, ACM Transactions on Intelligent Systems and Technology, 2011 (in press).

Bresina, J., Jónsson, A., Morris, P., and Rajan, K. 2005. Activity planning for the Mars Exploration Rovers. In Proceedings of the 15th International Conference on Automated Planning and Scheduling, Monterey CA, USA, June 5-10, 2005, S. Biundo, K. Myers, K. Rajan, Eds. AAAI, Menlo Park, CA., USA 40-49.

Vaquero, T., Romero, V., Sette, F., Tonidandel, F., Reinaldo Silva, J. ItSimple 2.0: An Integrated Tool for Designing Planning Domains. Proceedings of the Workshop on Knowledge Engineering for Planning and Scheduling, 2007.

J. Barreiro, J. Chachere, J. Frank, C. Bertels and A. Crocker. Constraint and Flight Rule Management for Space Mission Operations. International Symposium on Artificial Intelligence, Robotics, and Automation in Space, 2010.

Carraway, J., Squibb, G., Larson, W. Mission Operations. In Wetz, J. and Larsen, W. Space Mission Analysis and Design (3d edition). Microcosm Press, El Segundo, CA 1999, p. 587-620.

Garcia G., Barnoy, A., Beech, T., Saylor, R., Cosgrove, J., Ritter, S. Mission Planning and Scheduling for NASA's Lunar Reconnaisance Orbiter. Proceedings of the Ground Systems Automation workshop, 2009.

Izygon, M., Kortenkamp, D., Molin, A., "A Procedure Integrated Development Environment for Future Spacecraft and Habitats," *Space Technology and Applications* International Forum, Albuquerque, NM, 2008.

Brat, G., Gheorghiu, M, , Giannakopoulou, D., "Verification of Plans and Procedures," IEEE Aerospace Conference, IEEE, Big Sky, MT, 2008.

Scott Bell, David Kortenkamp, Jack Zaientz.A Data Abstraction Architecture for Mission Operations. Proceedings of the International Symposium on Artificial Intelligence Automation Robotics in Space 2010.

Simon, G. Shaya, E. Rice, K. Cooper, S. Dunham, J. Champion, J. "XTCE: A Standard XML-Schema for Describing Mission Operations Databases," *IEEE Aerospace Conference*, IEEE, Big Sky, MT, 2004.

Aghevli, A., Bachmann, A., Bresina, J.L., Greene, J., Kanefsky, R., Kurien, J.,McCurdy, M., Morris, P.H., Pyrzak, G., Ratterman, C., Vera, A., Wragg. S., Planning Applications for Three Mars Missions. Proceedings of the International Workshop on Planning and Scheduling for Space. Baltimore, MD, 2007.

Fox, M. & Long, D. (2003), PDDL2.1: An extension of PDDL for expressing temporal planning domains, Journal of Artif cial Intelligence Research 20, 61–124.

Clement, B., Durfee, E., Barrett, A. Abstract Reasoning for Planning and Coordination. Journal of Artificial Intelligence Research, vol. 28, 453-515, 2007.