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 benefited from advances in model-based planning software, but developing a planning model still remains a difficult task. Mission planning constraints arise from many sources, including simulators and engineering specification documents. Ensuring that these constraints are correctly represented in the planner's model is a challenge. As mission constraints evolve, planning domain modelers must add and update model constraints efficiently using the available source data, catching errors quickly, and correcting the model. We describe the current state of the practice in designing model-based mission planning tools and the challenges facing model developers. We then propose an Interactive Model Development Environment (IMDE) to configure mission planning systems by integrating modeling and simulation environments to reduce model editing time, generate simulations automatically to evaluate plans, and identify modeling errors automatically by evaluating simulation output.

Introduction

Mission planning is central to space mission operations, and has benefited from advances in model-based planning software (Chien et al., 2005, Bresina et al., 2005, Knight et al., 2009, Reddy et al., 2010). 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. A principal obstacle to fielding model-based planning systems for space missions is the complexity of their configuration, typically through domain modeling. Building a domain model requires identifying these information sources, understanding them, and often abstracting them.

Throughout the mission's development, changes to the constraints require changes to the models used to generate plans. Detecting and managing discrepancies between the constraints, plans generated from the model, and spacecraft behavior increases mission cost, schedule, and risk. A discrepancy may indicate an error in modeling that should be fixed prior to operations in order to avoid harm to the spacecraft and laborious re-testing at the subsystem and system levels. Such errors are difficult to avoid because models are often developed as disconnected abstractions of the system and are difficult to validate against spacecraft behavior and the sources of operational constraints. For example spacecraft behavior is often captured in a simulation testbed, which is used to validate command sequences that correspond to a plan. Today this process is performed by hand; the modeler must be familiar enough with the simulator output in order to characterize, for example, power usage, activity duration, and activity mutual exclusions.

Validating the planning system (not just a plan) is a central challenge to automating model development. An automated planning system is (in part) a plan verification system because it checks constraints on system states that the plan's activities affect. However, *validating* the plan additionally requires validating that the constraints and effects in the model are consistent with the simulation testbed. Validating the planning *model*, therefore, validates all plans that the planner generates or accepts as feasible.

Model checking software can automatically check a plan or model for *foreseeable* problems described in another formal language (e.g., Howey et al., 2004, Brat et al., 2008, Long et al., 2009, Raimondi et al., 2009, Cesta et al., 2010). From the point of view of system validation, model checking approaches only provide *verification* of plans/models in isolation.

In contrast, the challenge we pose is to automate the *validation* of the planning system application and the identification of modeling errors. Instead of model checking rules, we propose to use the simulator for checking the planning model. This approach avoids having to specify model checking rules and provides a direct path for finding problems that may be *unforeseeable*.

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 Earth Observing 1 (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, for an edit to the model, the simulator can be invoked to identify, avoid, and fix modeling errors. This continual testing could make the reviews simpler since plans have already been validated for a documented set of constraints by the simulators. The reviews could then focus on what is modeled instead of how.

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, and describe the various challenges in ensuring this activity description is valid. We then propose an Interactive Model Development Environment (IMDE) that simplifies the construction, validation, and maintenance of automated planning system models. The latter half of the paper describes the proposed IMDE from a functional and architectural perspective. We then describe both current and near-term technologies that can be used to build such an IMDE, and conclude with a description of open research problems.

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 (Clement and Johnston, 2005). 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. Flight dynamics, communications, surface operations, and science planning provide input to mission and activity planning but can also be constrained by it.

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.

Challenges of Configuring the Mission Planning System

Academic planning (modeling) 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.

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 that the "right way" is to build a declarative planning model. However, the sources of space mission constraints present

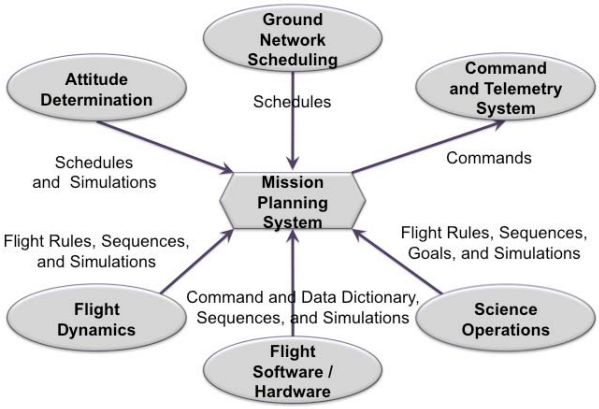

Figure 1: How the Mission Planning System interacts with the Mission Operations System.

challenges to model building. Figure 1 illustrates 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, describes how constraints change, and explores the difficulty of modeling an activity.

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-2-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-2-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 (Barrett et al., 2009; 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).

Flight dynamics and communications

Flight dynamics may simulate orbits and trajectories 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, such as 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, constraints governing communication coverage times may also need to change.

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, dust on solar panels, frozen robotic arms or wheels, or new power or thermal limits (Barrett et al., 2009; Tompkins et al. 2010).

As mission planning systems mature, planners often find that satisfying constraints is too difficult. Science teams and spacecraft designers can provide overly conservative constraints early in mission development. Analysts may determine that the constraints can be relaxed without compromising safety or science (Barrett et al., 2009).

Example Activity Model of Spacecraft Slewing

There is a long-standing, fundamental problem in applying automated planning to physical systems. In order to make this concrete, consider the difficulty of representing a relatively simple spacecraft activity for changing attitude.

```
(: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 difficulty stems from developing system models that are disconnected from the system (leading to inaccuracy) and from modeling representation language limitations (adding to complexity). The PDDL above specifies a spacecraft attitude change activity. The activity model is more abstract than a typical simulator's, which would use 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 from the simulator to configure the planning system is manual, inefficient and error-prone. The modeler must address the following issues:

- How do planner model attitudes relate to real spacecraft operations' continuous attitudes? For example, does it suffice to represent a deep-space craft with camera directional sensors using a discrete valued attitude variable with values such as to-Earth (for deep-space), Earth-nadir (for Earth orbits), Sun-pointing (for solar power generation), and others for sets of navigation guide stars? How does data from the inertial measurement unit map to these discrete directions?
- How does the planner model battery discharge? How can the model conservatively estimate the battery energy consumed by subsystems for different possible system states? For example, does temperature affect power usage? How is a cap on battery capacity modeled to avoid overfilling? How is solar recharging modeled?
- What drives slew duration? Is it proportional with angular slew distance? Will a slew always follow the shortest rotation? Must it avoid pointing instruments at the sun? What determines the choice of control system (reaction wheels, thrusters, or torque rods)?
- How is reaction wheel momentum dumped?
- Along what axes can the spacecraft slew while communicating? conducting science measurements? recharging the battery? changing trajectory?
- What are the communication coverage requirements? What information is needed about the spacecraft orbit, availability of ground communication assets, and the spacecraft antenna type and configuration? When do ground stations require communications to monitor trajectory changes or other related activities?
- How does the abstract slew correspond to one or more sequences of spacecraft commands? Are there setup and teardown activities? Is the slew for each axis performed separately to avoid risk of concurrent interactions?

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

- New targets or navigation aids require updating the set of discrete attitudes.
- 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 power consumption. If planning determines mission objectives are infeasible, 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

In this section we describe how an Integrated Model Development Environment (IMDE) could integrate planning and simulation to address the challenge of model development. This integration could simplify validation of models within the development cycle, thereby making modeling for space mission planning more efficient. The challenge is to integrate a planner and simulator to automate model development and validation.

The next sections state assumptions that simplify the discussion, describe IMDE design features and architecture, outline a concept of operation for modeling with the IMDE, characterize discrepancies that indicate modeling errors, and describe how these modeling errors may be identified and fixed.

Assumptions

The following assumptions simplify in the description of the IMDE and also indicate additional challenges addressed toward the end of the paper.

- The simulator input includes a list of time-tagged commands.
- 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 (the user can neither change nor inspect its code and models)
- The simulator outputs time-tagged value samples of system state variables.

- Formal flight rules define mission constraints that are verifiable with the simulator output.
- Every plan that the planner sends to the simulator is consistent with the planner's model.
- An action in the plan corresponds to a list of time-tagged commands.
- The planner generates plans that conform to model constraints or else identifies all constraint violations.

IMDE Design Features

The hypothetical IMDE shares many features of a traditional programming language Integrated Development Environment (IDE); An IMDE's model corresponds to an IDE's code, plans correspond to test cases, and the simulator corresponds to the computer. One distinctive IMDE function is the generation of test cases to aid model validation. Another is the generation of suggestions on how to fix modeling errors. In the traditional IDE, this is similar to suggesting code fixes for program run failures. Following sections discuss validation and model fixes.

Figure 2 shows the system architecture of the proposed IMDE. The Model Editor provides traditional IDE functions. The Simulation API Browser provides model creators access to the simulation API. With the Abstraction Editor a user documents how plan model building blocks (objects, states, timelines, actions, constraints) relate to data and commands in the simulation API, thus providing traceability for detecting model problems. These

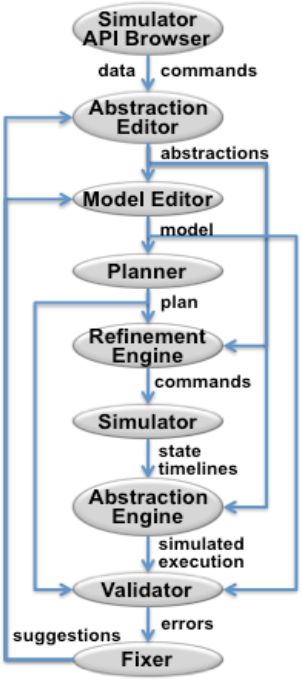

Figure 2. Hypothetical IMDE Architecture and data flow.

abstractions are the semantic glue connecting the planner to the "the world"/simulator. The Abstraction and Refinement Engines integrate the planner and simulator. The Refinement Engine transforms a plan into simulator command input. The Abstraction Engine transforms simulator output into an actual/simulated execution for comparison with the expected/planned execution. These executions are time-tagged actions and state variable values in the language of the planner. The Validator identifies discrepancies between the two executions, errors reported by the simulator, and any planning model constraint violations, some of which the simulator may not check (e.g., flight rules). A Plan Viewer (not shown) comparatively displays the simulated and planned executions (e.g., in a Gantt chart). The Plan Viewer (and/or an Error Viewer) visualizes discrepancies between the executions and highlights those that indicate modeling errors. Finally, the Fixer suggests model changes that may eliminate one or more errors seen in the current and past simulations of different plans. We later explain how to detect errors and make suggestions.

Concept of Operation

An IMDE user may start with an empty or existing planning domain model. Depending on the modeling language, an edit to the model may add, change, or remove actions, state variables, constraints, and effects (or their associated object types and sets). The user may create and edit abstractions to ground the model in simulator elements. A simulation interface exposes these elements, including commands and system state variables. These edits initiate the following basic workflow:

1. The user edits the model, or
2. the user edits abstraction by either
   a. copying variables from the simulator interface to the model (e.g. `sunangle`),
   b. abstracting variables in the model (e.g. `(pointing Earth)` is true in the planner if the simulator's *xyz* attitude is for each axis within 1 degree of the attitude to point directly at Earth),
   c. copying a simulator command to the model as an action (e.g. turn on CPU), or
   d. abstracting a command sequence to model an action (e.g. command sequence to slew spacecraft)
3. The IMDE generates possible initial states and plans and tests each by
   a. translating the initial state and plan into simulator commands,
   b. running the simulator with those commands,
   c. translating simulator output to an execution,
   d. checking the execution for violations of constraints in the planning domain model, and

e. checking for discrepancies between planned and simulated executions.
4. The IMDE analyzes test results to suggest changes to the planning model that could fix discrepancies.
5. The user assesses planned and simulated executions, their constraint violations, their discrepancies, and suggested fixes.
6. Repeat.

The idea is that when the user edits the model, in the background the IMDE generates and simulates different plans to search for discrepancies indicating modeling errors. The user can be made aware of these errors even while editing (much like syntax errors in an IDE), and when the user is ready to see what is in error, the IMDE may already have suggested fixes for the user to select.

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 may be impractical to regenerate all of the plans every time the underlying model changes. 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.

Translating Plans as Abstraction and Refinement

The previous workflow uses abstractions heavily. For example, in step 3a the Refinement Engine may translate one `slew(?from,?to)` action in the plan into three ordered subsequences of simulator commands to rotate the spacecraft around each of its three axes. In step 2c and 2d, the user specifies this abstraction as an action decomposition, similar to hierarchical plan decomposition.

Another abstraction type for data specifies how state variables in the planning model relate to those in the simulator output. For example, an abstraction could map the simulator *xyz* spacecraft attitude to a discrete `(pointing ?target)` planner predicate, with `?target` either `Earth`, `Sun`, or `SomewhereElse`. An abstraction function could specify that `(pointing Earth)` is true if the simulator *xyz* attitude is within 1 degree of pointing the transceiver to the Earth's center[1].

When the Refinement Engine translates initial state and plan information into simulator commands using these abstractions in steps 2a and b, some data abstractions may need to be reversed. For example, translating a plan's `slew(Sun,Earth)` action into simulator commands would translate the `Sun` and `Earth` symbols to the corresponding *xyz* attitudes for pointing to the targets.

The Validator checks for discrepancies with the planned execution and helps identify modeling errors by translating simulation results into execution information in the planner language using the abstractions in Step 3c. The abstractions provide the time-varying planner state values, but another step is needed to construct the execution that explains these values, using the Abstraction Engine. Our assumptions make this relatively simple, but in general it can be a difficult state estimation optimization problem.

Identifying Modeling Errors

Modeling errors are indicated by errors explicitly reported by the simulator and by plan constraint violations on the simulated execution that do not occur in the planned execution (a discrepancy in constraint violations). For example, in testing the `slew(Sun, Earth)` action, the simulator might report an error from the fault management system because the computer had not yet been booted when commands were sent to the reaction wheels (a flight rule violation). This is an error in the planning model because the slew action lacked a necessary precondition that the computer be booted. As another example, the plan test case might include a goal (or constraint) `(pointing Earth)` to check that the effect of a `slew` is achieved. The simulator output could be error-free, and yet translate back to an execution state where `(pointing Earth)` was never achieved, failing the goal. This could be the result of the plan containing an overlapping `slew` that commanded the spacecraft to retarget the slew. In this case the modeling error was in allowing overlapping slews.

This simple specification for identifying modeling errors applies generally to different kinds of errors. For example, how would an error in the timing of a slew be detected? If the model specified a fixed duration for `slew`, the test plan still only needs a constraint that `(pointing ?to)` be true at the end of the `slew` activity. If the slew takes longer than expected, then the constraint will be violated in the simulated execution.

Discrepancies between the planner and simulator 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

[1] An abstraction could be any function of a set of time-varying variables that calculates the time-varying values of another set of variables such that the magnitude of the domain is smaller than the magnitude of the range.

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, Aghveli et al. 2007).

At the same time, the discrepancy might indicate an opportunity to improve efficiency: a more detailed battery depletion model could enable scheduling more activities. These discrepancies of inefficiency could be detected by finding plans with constraint violations that do not occur in the simulated execution.

Generating Plans to Validate the Model

Different test plans are generated (step 3) in order to validate that the model will work for all situations. This requires validating all possible plans acceptable by the model. In general, this may be an infinite number of plans, but there may be a manageable number that is enough to validate a single part of the model.

For example, if the user wants to ensure that the `(pointing ?to)` effect is always satisfied at the end of `slew(?from,?to)`, then a complete space of plans to test would combine all possible initial attitudes, slews for all target attitudes (slew from each attitude to each other attitude), all possible additional actions (slews from each target to each other target), and the different temporal orderings of those other actions with respect to `slew(?from,?to)`.

It is possible to generate all of these plans with special purpose code, but the planner itself may 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 `(pointing ?to)` at the end? The set of valid solutions to this planning problem is the test suite.

It is expected that multiple simulations could be necessary to validate a single plan. For example, there are an infinite number of *xyz* attitudes that translate to `(pointing Earth)`. So, why not test all possible simulations instead of all possible plans? If plans are meant to be the only mechanism for generating command sequences for the spacecraft, the other simulations will never occur because a plan only translates to one set of commands resulting in one deterministic simulation. On the other hand, the initial state is not dependent on actions in the plan, so the complete space of test cases would include the infinite number of attitudes that translate to `(pointing Earth)`. In this case, conventional test coverage techniques may still be necessary.

Another reason to generate simulations instead of plans may be that a model is in its infancy, and many actions have yet to be modeled, so the necessary plan-based test cases to validate the first modeled action would be insufficient. Thus, generating simulations based on the simulator interface specification (using simulator commands instead of planner actions) would be useful and more robust to model changes. It may also be better to generate simulator-based test cases when there are many actions in the planning model. If activities are defined for many combinations and orderings of simulator commands, then the space of plans necessary to validate an action could be greater than the space of simulations due to repetition of commands in a combination of actions.

Again, it may be possible to cleverly scope the validation to reduce the number of sequences tested. For example, test cases including two slews following the slew to be validated should find the same errors as those test cases with only a single following slew. Thus, a tractable number of test cases may be identified for validating an action in a model. This test coverage problem is known to be quite difficult and, thus, part of the challenge.

The tractability of validating the entire model depends on that of individual actions. Validating each action in isolation is enough to validate the entire model since the soundness of the planner guarantees action combinations.

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 modeler may not be able to deduce the actual mistake by looking at any one execution. For example, suppose the slew was never executed because the CPU was never turned on, resulting in a simulation error flag. There would be a violation of `(pointing Earth)` in the simulated execution, but no information in the output ties these errors with the state of the computer. So, the modeler would have to know the spacecraft (and simulator) very well to guess the problem after seeing it in a single run.

By finding relationships between plan/state attributes and simulator/discrepancy errors, the IMDE can generate plausible suggestions for fixing the model. For example, if a complete set of test cases showed that the slew failed every time that the computer was not booted, a machine learning classifier or data mining algorithm could identify the pattern. Then, the IMDE could suggest abstracting the *computerMode* variable in the simulator interface to a `cpu-on` predicate in the planning model and add the predicate as a precondition to `slew`. Other suggestions include adding a constraint that a `turnOnCpu()` action always precedes `slew(?from ?to)` or adding a simulator command to the slew abstraction/decomposition to *bootCpu()*. These suggestions from the IMDE Fixer

component (Figure 2) could include changing action constraints, adding state variables, or creating new actions.

The challenge of generating suggestions may be in framing the learning problem. Plans have variable numbers of actions, so there is not an obvious feature set over which to learn. In addition, the modeler may want suggestions in terms of complex functional relationships of multiple variables. For example, the desired fix may be to avoid exhausting memory storage by adding a constraint that the sum of durations of all communications activities in a day must be greater than the sum of data collected multiplied by a particular constant. The number of functional relationships that may be part of a feature set of a learning algorithm could easily be intractable. On a positive note, the modeler may be able to deduce the needed fix with the help of overly-specific suggestions learned from a limited set of features.

Technology Foundations

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

The itSimple tool (Vaquero et al., 2007) is a plan domain modeling environment similar to the proposed IMDE. Users of itSimple can build static models of objects, actors, and relationships between them in a specialization of UML. Users specify dynamic (simulation) models of how states of the objects can change using Petri Nets (an encoding of state charts). itSimple automatically translates these models to PDDL. A difference from the IMDE approach is the assumed access to the simulator model (white-box simulation).

PAGODA uses black box simulations to learn activity preconditions in an interactive model environment (desJardins, 1994). This is a basic feature of the Fixer (Figure 2) used to identify modeling errors. Another interactive environment helps the user identify model errors by constructing an invalid plan (that the user knows should be valid) by relaxing constraints (Chien, 1996).

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; 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 that 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, there is no abstraction mapping, and procedures lack formalisms needed for planning.

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 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 editing data abstractions for the IMDE, but it would need to be extended to capture transformations of plan actions into simulator commands.

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. We explore the goals of validating that all plans and suggesting fixes to the model, not just the plan. Furthermore, the approach in VAL would have to apply to simulated executions.

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 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 may require learning abstractions from simulated command sequences, which plan domain learning systems presently do not do, except PAGODA (desJardins, 1994).

Techniques for ordering test cases to expose errors more quickly can also be leveraged. Instead of generating test plans by systematically trying each permutation of plan features, test cases may be chosen that are believed to more likely discover a flaw based on results of past cases. The Nemesis test system has had success with this by using a genetic algorithm to smartly choose test cases (Barltrop et al., 2010). A complementary strategy is to use coverage techniques to quickly sweep across the landscape of test cases and learn combinations of features to more quickly converge on a formula describing the conditions under which a flaw appears (Barrett, 2009). This can be used to converge quickly on suggestions to fix modeling errors.

Challenges in relaxing the assumptions

The usefulness of the described IMDE may still be insufficient because of limiting assumptions. We describe those we deem important and their associated 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 translates 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 duration value to use? Moreover, constructing the simulated execution from state values may not be obvious and, in general, can be a difficult state estimation problem!

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 mapping of plan actions to simulator command sequences) and parameter dependency functions, how can errors and fixes be automatically identified? Does this similarly extend to errors in abstraction specifications?

Relaxing other assumptions may not pose difficult research challenges but can change the nature of the system capability. For example, if the simulator or system (e.g. spacecraft) does have defects, then discrepancies that are inconsistencies between planned and executed behavior may now be (in addition to modeling errors) indications of those system defects. So the IMDE now can identify simulator and system defects and 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 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. Since the model of the system is complete, SLATE can prove system-level properties as model checking does.

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 prove that a plan abstraction is 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 and the IMDE, summary information relies on a detailed, accurate simulation model (white box).

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 an IMDE as a means of reducing up-front errors by catching and repairing errors during model development. 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 case plans 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?
- How can the architecture be adapted to suggest changes for plan quality?

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

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.

Barltrop, K., Clement, B., Horvath, G., and Lee, C. Automated Test Case Selection for Flight Systems using Genetic Algorithms. Proceedings of the *AIAA Infotech@Aerospace Conference*, 2010.

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

Barrett, A., Bass, D., Laubach, S., and Mishkin, A. A Retrospective Snapshot of the Planning Processes in MER Operations After 5 Years. In *Proceedings of the International Workshop on Planning and Scheduling for Space.* 2009.

Barrett, A. and Dvorak, D. A Combinatorial Test Suite Generator for Gray-Box Testing, *IEEE SMC-IT* 2009.

Bell, S., Kortenkamp, D., and Zaientz, J. A Data Abstraction Architecture for Mission Operations. In *Proc. of the International Symposium on AI, Robotics, and Automation in Space*, 2010.

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.

Brat, G., Gheorghiu, M, , Giannakopoulou, D., "Verification of Plans and Procedures," In *Proc. of IEEE Aerospace Conf.*, 2008.

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.

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, p. 587-620, 1999.

Cesta, A., Finzi, A., Fratini, S., Orlandini, A., Tronci, E. Validation and Verification Issues in a Timeline-Based Planning System. *Knowledge Engineering Review*, 25(3): 299-318, 2010.

S. Chien, R. Sherwood, D. Tran, B. Cichy, G. Rabideau, R. Castano, A. Davies, D. Mandl, S. Frye, B. Trout, S. Shulman, and D. Boyer. Using Autonomy Flight Software to Improve Science Return on Earth Observing One. *Journal of Aerospace Computing, Information, and Communication*, 2005.

Chien, S. A. Static and Completion Analysis for Planning Knowledge Base Development and Verification. In *Proc. of 3rd Int'l Conf. on Artificial Intelligence Planning Systems*. 1996.

Cichy, B., Chien, S., Schaffer, S., Tran, D., Rabideau, G., Sherwood, R. Validating the Autonomous EO-1 Science Agent In: *Int'l Workshop on Planning and Scheduling for Space*. 2004.

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

Clement, B. and Johnston. M., The Deep Space Network Scheduling Problem. In *Proceedings of the 17th Innovative Applications of Artificial Intelligence Conference*, 2005.

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

desJardins, M. Knowledge Development Methods for Planning Systems. In *Proceedings of the AAAI Fall Symposium on Planning and Learning*, New Orleans, 1994.

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

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

Howey, R. and Long, D. and Fox, M. VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *Proc. ICTAI*. pp. 15—17, Nov 2004.

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

Knight, R., Chouinard, C., Jones, G., and Tran, D. Planning and Scheduling Challenges for Orbital Express. In *Proceedings of the Inte'l Workshop on Planning and Scheduling for Space*. 2009.

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. In *Proceedings of the AIAA Space Conference*. 2004.

Long, D., Fox, M., and Howey, R. Planning Domains and Plans: Validation, Verification and Analysis. In *Proc. Workshop on V&V of Planning and Scheduling Systems*, 2009.

Raimondi, F., Pecheur, C., and Brat, G. PDVer, a Tool to Verify PDDL Planning Domains. In *Proc. Workshop on Verification and Validation of Planning and Scheduling Systems*, ICAPS, 2009.

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.

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. In *Proceedings of AIAA Space* Conference(SpaceOps) 2010.

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

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