

The Challenge of Grounding Planning in Simulation with an Interactive Model Development Environment

Bradley J. Clement^{*}, Jeremy D. Frank^{**},
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

A principal obstacle to fielding automated planning systems is the difficulty of modeling. Physical systems are modeled conventionally based on specification documents and the modeler's understanding of the system. Thus, the model is developed in a way that is disconnected from the system's actual behavior and is vulnerable to manual error. Another obstacle to fielding planners is testing and validation. For a space mission, generated plans must be validated often by translating them into command sequences that are run in a simulation testbed. Testing in this way is complex and onerous because of the large number of possible plans and states of the spacecraft. Though, if used as a source of domain knowledge, the simulator can ease validation. This paper poses a challenge: to ground planning models in the system physics represented by simulation. A proposed, interactive model development environment illustrates the integration of planning and simulation to meet the challenge. This integration reveals research paths for automated model construction and validation.

Introduction

A model-based system is software typically for analyzing or optimizing another system that it models in some representation language. There are several applications that have benefited from using model-based planners. But, there are long-standing, fundamental problems in applying automated planning to physical systems: models are abstractions that are disconnected from the physical system (reducing accuracy) and limited in representation (increasing complexity). Brooks remarked, "Explicit representations and models of the world simply get in the way. It turns out to be better to use the world as its own model" (1991). The successes of model-based applications dull this point, but developing a model that is sufficient for a real application can be a painful struggle, especially if the modeling language is missing basic features like numeric state variables, in which case the language can seem to "get in the way." While languages have become more expressive, algorithms that parse them do not scale well, and detailed modeling may require too much effort. Abstraction has its advantages! So, can existing planning systems somehow use "the world" as their model?

That is the challenge: to ground an automated planner in system physics ("the world") and thus simplify model development, verification, and validation. Space missions often develop simulation testbeds that serve as ground truth for the system and can be automated to evaluate test cases in batches. So, one approach to meeting the challenge is to help automate model development and testing by integrating the planner and simulator.

We first describe a sample activity to show the complexity of translating diverse information sources into different elements of a declarative domain model. We then explain how prior work in verification and validation does not address this problem. We propose an Interactive Model Development Environment (IMDE) to simplify the construction, validation, and maintenance of automated planning systems with help from a simulator. The majority of the paper describes IMDE functions and architecture. We discuss both current and near-term technologies that can be used to build such an IMDE and mention progress on a proof-of-concept implementation. We conclude with research goals that could help produce valuable mission planning technologies.

Model Development Challenges

As discussed above, fielding model-based planning applications is challenging because typical modeling processes are complex and error-prone and because the combinations of possible test scenarios can seem as overwhelming as testing the entire system being modeled.

Originally planning languages and algorithms used only Boolean state variables. 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. Modeling choices can strongly influence the performance of automated planning. So, performance requirements can spur model revisions that increase complexity further. This complexity compounded by human error and lack of information about the modeled

system's behavior can produce inconsistencies with the model and the modeled system. Identifying and fixing these inconsistencies can require significant work.

To make the discussion concrete, consider the difficulty of representing an activity for changing a spacecraft's attitude (its 3-dimensional orientation).

Example activity model of spacecraft slewing

```
(: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 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 may have a lot of questions:

- 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.

Verification, validation, and model checking

Validating the planning model (not just a plan) is a central challenge to automating model development. The planning system is partly 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 effects are realized as expected. Validating the planning *model* requires validating all plans that the planner generates or accepts as feasible.

Model checking may detect violations of formal properties by the planning model or individual plans "in isolation" (e.g., Howey et al., 2004, Brat et al., 2008, Long et al., 2009, Raimondi et al., 2009, Cesta et al., 2010), but our goal is to validate the model against the simulator. Simulation of activities in the planning model can directly indicate problems, for example, an unrealized effect of an activity or a system fault. Model checking cannot substitute for this functionality without using a complete model of the simulator. Current model checking systems have the same representation and scaling problems as

planning, so a detailed model (which rarely exists anyway) would likely be unusable.

Interactive Model Development Environment

We now describe an approach that integrates planning and simulation in order to help automate model development and validation in the context of a hypothetical IMDE. We make assumptions that simplify the discussion, describe IMDE design features and architecture, and outline a concept of operation for modeling with the IMDE.

Assumptions

The following assumptions simplify stating the challenge in the IMDE context 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 is sound but not necessarily complete.

IMDE design features

The hypothetical IMDE could share 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 1 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 abstractions are the semantic glue connecting the planner

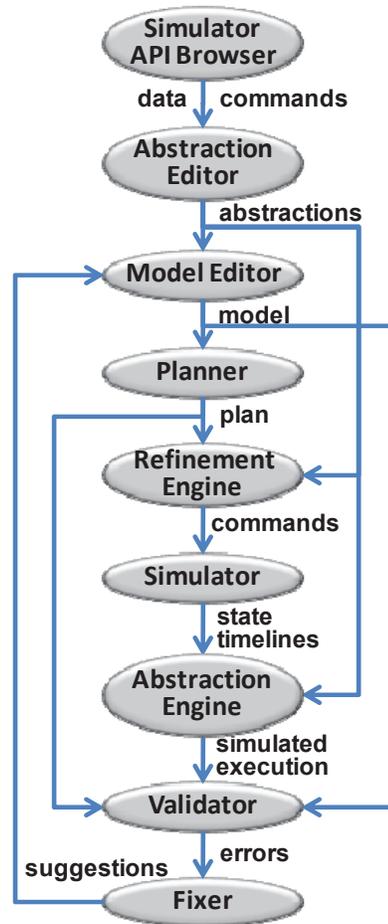


Figure 1: Hypothetical IMDE architecture.

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 explain how to detect errors and make suggestions after describing the IMDE concept of operation.

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.

Translating plan information 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 (Clement et al., 2007).

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 for each axis is within 1 degree of pointing the transceiver to the Earth's center. In general, an abstraction could be any function of a set of time-varying variables that calculates the time-varying values of some variable.

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 Abstraction Engine 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. 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 translate back to an execution where (`pointing Earth`) was never achieved, failing the goal. This could be the result of the plan containing another overlapping `slew` command 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 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). However, the discrepancy might indicate an efficiency improvement opportunity: a more detailed battery depletion model could enable scheduling more activities.

These discrepancies of inefficiency could be detected by running plans that have constraint violations through the simulator and seeing if the violation occurs in the simulated execution.

Generating plans to validate the model

The reason for generating different plans to test (step 3) 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. In general, there may be an infinite number of possible 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 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 `(pointing ?to)` at

the end? The set of valid solutions to this planning problem is the test suite.

Now, it is expected that multiple simulations could map to 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 the model has just been started, 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 simulator 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 combinations of actions.

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 the safing of the spacecraft 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` or adding a simulator command to the slew abstraction/decomposition to `bootCpu`. These suggestions from the IMDE Fixer component (see Figure 1) could include changing constraints on an action, adding state variables, or creating new actions. Similar suggestions could be made to fix the abstractions.

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 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. A distinct difference from the IMDE approach is the assumed access to the simulator model (white-box simulation).

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 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 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. 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 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.

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 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 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 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 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 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 and the IMDE, summary information relies on an accurate, detailed model and, thus, applies only to white box simulation, similar to model checking approaches.

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 semantics of the simulation language and may not be feasible for all simulation approaches.

Preliminary Proof of Concept

We have implemented a simple simulator and planner to explore the challenges of building an IMDE. The system provides a two-dimensional slew example and a simple

surface explorer (e.g., rover) example. The system manages a simulator (implemented with ASPEN, Chien et al., 2000) and a planner (EUROPA, Frank and Jónsson, 2003) using an enhanced Eclipse IDE. Simple file formats are used for initial state, simulator commands, output, and executions. A Java library translates these files, supports abstraction specifications, and fulfills the Refinement and Abstraction Engine roles. Currently, the system only detects model errors from single plan simulations.

Conclusion

The maturation of model-based planning provides an opportunity to improve the state of the art in planning applications. But, the improvement requires spacecraft engineers to build and validate planning models that represent complex constraints derived from diverse information sources. This paper hypothesizes that an Interactive Model Development Environment could overcome many of the associated challenges, providing features to prevent, catch, and repair model errors. While the technologies described above support the described IMDE features, 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

We gratefully acknowledge the assistance and comments of members of the MER, LCROSS and LADEE mission operations and flight software teams, as well as members of the Johnson Space Center Mission Operations Directorate, in formulating this work. Some of the research described in this paper was performed by the Jet Propulsion Laboratory, California Institute of Technology. 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, California Institute of Technology. © 2011 California Institute of Technology. Government sponsorship acknowledged.

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 *ALAA Infotech@Aerospace Conference*, 2010.
- Barrett, A. and Dvorak, D. A Combinatorial Test Suite Generator for Gray-Box Testing, IEEE SMC-IT 2009.
- Scott Bell, David Kortenkamp, Jack Zaiantz. 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.
- Brooks, R. A. Intelligence without representation. *Artificial Intelligence*. 47, pp. 139–159, 1991.
- 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.
- Chien, S., Rabideau, G., Knight, R., Sherwood, R., Engelhardt, B., Mutz, D., Estlin, T., Smith, B., Fisher, F., Barrett, T., Stebbins, G., & Tran, D. ASPEN - Automating space mission operations using automated planning and scheduling. In *Proc. SpaceOps*, 2000.
- Clement, B., Durfee, E., Barrett, A. Abstract Reasoning for Planning and Coordination. *Journal of Artificial Intelligence Research*, vol. 28, 453-515, 2007.
- Cresswell, S.; McCluskey, T. L.; and West, M. M. 2012. Acquiring planning domain models using LOCM. *Knowledge Engineering Review*, to appear.
- Fox, M. & Long, D. (2003), PDDL2.1: An extension of PDDL for expressing temporal planning domains, *Journal of Artificial Intelligence Research* 20, 61–124.
- Frank, J. and Jónsson, A. Constraint-Based Interval and Attribute Planning. *Journal of Constraints, Special Issue on Constraints and Planning*, 2003.
- 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.
- Izygon, M., Kortenkamp, D., Molin, A., “A Procedure Integrated Development Environment for Future Spacecraft and Habitats,” *Space Technology and Applications International Forum*, 2008.
- 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.
- 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.