An Intelligent Agent for Autonomous Lunar Exploration

Jeremy Frank
Computational Sciences Division
NASA Ames Research Center, MS 269-3
frank@email.arc.nasa.gov
Moffett Field, CA 94035

Abstract

We describe an intelligent agent that supports low-cost, highly autonomous mobile exploration of the Moon with a hypothetical spacecraft. The vehicle is designed to reuse its descent and landing system to enable surface mobility. State-of-the-art space mission software automation based on automated planning, control and intelligent system health monitoring, coupled with a traditional flight software core, enables autonomous operations and recovery from many classes of faults. We first describe the scenarios hilighting the value of autonomous operations. We then describe the software architecture, concentrating on the distinct software functions by mission phase and the architectural. Finally, we present the results of a medium-fidelity simulation of the spacecraft on these scenarios.

Introduction

The ability to explore many sites on the surface of a new planet enables a deep understanding of planetary geology using a single launch, resulting in the acquisition of considerable scientific knowledge at comparatively low cost. The Sojourner Rover mission and the ongoing Mars Exploration Rover missions of recent years have demonstrated the value of mobile robotic systems for planetary exploration. Advances in autonomy have been demonstrated on free-flying spacecraft such as Deep Space One (Muscettola *et al.* 1998), Earth Observing One (EO-1) Autonomous Sciencecraft Experiment (ASE) (Chien *et al.* 2005), and Space Technology Five (ST-5) (Stanley *et al.* 2005). Advances in computer vision coupled with autonomy have been demonstrated for descent and landing, e.g. the DIMES camera during the Opportunity landing (Johnson *et al.* 2007), and on-orbit operations, e.g. Orbital Express (Knight 2008). Low-cost future robotic exploration of the Lunar surface will depend on these new technologies, as they enable the conduct of surface operations safely and efficiently with minimal human intervention and oversight.

We describe an intelligent agent that supports low-cost, highly autonomous mobile exploration of the Moon with a hypothetical, but realistic, spacecraft. We employed a combination of traditional flight software components for control on short timescales (milliseconds to seconds), coupled with agent technology for planning, mission-management and diagnosis for control over longer timescales (minutes). We selected existing model-based technology to build our agent. We were thus faced with two fundamental problems to solve. The first problem is capturing the knowledge of the spacecraft, underlying software, and goals of each mission phase in models to drive the behavior of the agent. The second problem was one of integrating the agent software modules, both with the underlying flight software and with each other.

The paper is organized as follows. We first describe the mission, the spacecraft design, and the on-board software design. We chose model-based planning and agent software to provide high degrees of onboard automation. We describe the models, search control challenges, and software integration issues for the planning and agent technologies. We describe a series of simulations which we used to validate the on-board software design. The simulations included a spacecraft dynamics model, flight software including the system bus, mission automation software, and a medium-degree simulation of Lunar terrain. We successfully tested the design, simulating nominal operation and a variety of off-nominal conditions, including recoverable software faults and unrecoverable hardware faults in the spacecraft propulsion system.

Spacecraft Design and Mission Phases

We assumed that the spacecraft reuses its entry, descent and landing system (EDS) to enable surface mobility. We assumed the spacecraft lands within 1 km of a science objective on the Lunar surface. The EDS system consisted of a main engine and 2 cameras, of resolution 1600×1200 pixels with a 66^0 field of view, separated by 1.7 meters on the bottom of the spacecraft; this design allows the spacecraft to (hypothetically) avoid hazards on the scale of 30 cm.

In addition to the main engine, which provides $600N$ of thrust, we assumed 4 attitude thrusters of $30N$ each. We assumed at most 10 kg of fuel (divided between propellant and oxidizer) was available for mobility operations after landing. For power we assumed 4 solar panels mounted on the top of the spacecraft; these have $2.55m^2$ total area. Solar irradiance at the Moon, denoted SI, is 1367.6 watts per square meter. We assumed solar panel efficiency, denoted E, of 15%. The solar panels are oriented at an angle and arrayed around the spacecraft's vertical axis. If we denote the vector from the sun to a panel P as $\vec{L}_P$ and the normal to this vec-

tor as $\vec{N}_P$, the available solar power for panel P is calculated by $SI \times E \times \vec{L}_P \cdot \vec{N}_P$. As a backup energy source we assumed a Lithium-Ion non-rechargable battery that provides 200 Watt-Hours of power.

We focused on simulating three mission phases that demonstrate the benefits of automation. After landing, it is traditional to perform a detailed checkout of all spacecraft subsystems; this usually involves analyzing logs of activity during landing, and interrogating spacecraft subsystems after landing to ensure they are functioning properly. The next phase involves localizing the spacecraft on the surface, and planning to move from its current location to the intended destination. The next phase is executing the mobility plan, and moving to the target location.

The mobility phase is the most complex, as it exercises all major software and hardware spacecraft systems; as such, we describe it in more detail. Mobility operations are broken up into several distinct phases. The spacecraft first executes a main engine burn to gain altitude. It then executes a change of attitude, followed by a main engine burn to place it on a ballistic trajectory downrange towards the target location. Partway to the target location the spacecraft again changes attitude, and then executes a main engine burn to halt its down-range motion. It then uses its main engines to descend at a rate of 10 meters per second until it is 10 meters over the surface. At this point, the spacecraft hovers while onboard computer vision software finds a safe landing spot. Once the landing spot is found, the spacecraft descends at a rate of 1 meter per second until it lands. This sequence is shown in Figure 1.

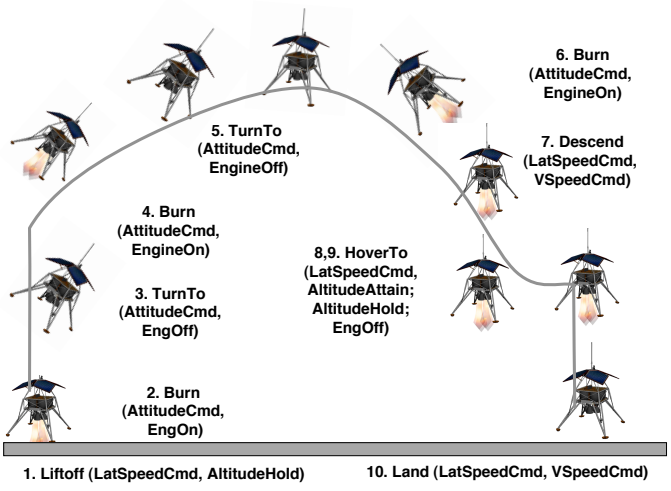

Figure 1: The different phases of the mobility operation. Each phase is labeled with the mode of the Flight Management Computer (FMC), which is in turn decomposed into the ACS and MECS modes. Note that the Hover mode of the FMC requires complex MECS mode state changes.

Spacecraft Software Architecture

We now describe the software design of the spacecraft. While high degrees of on-board automation have proven themselves in a variety of recent experiments, such software is still considered a risk in many space missions. For this reason we chose a "separation architecture", in which we distinguish between highly reliable traditional flight software functions, and higher-order functions making use of automation technologies like planning, control agents, and intelligent systems health monitoring. In the following sections we describe each major class of software on the spacecraft. Figure 2 shows the major software components, messages and data flow to and from them.

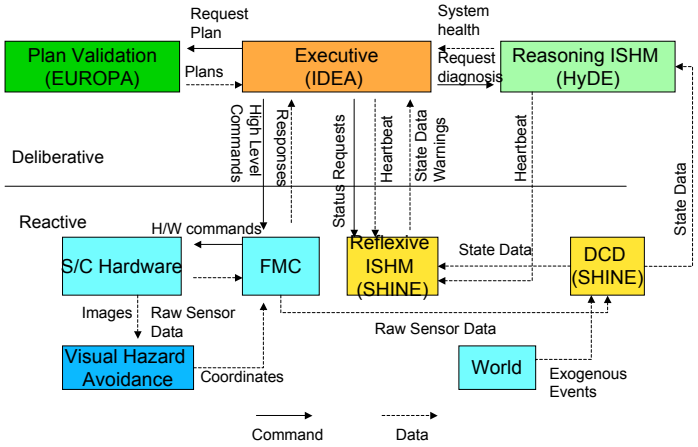

Figure 2: The software architecture. Traditional flight software components are shown at bottom, while autonomy components are shown at top; command and data paths are also indicated.

Flight Management Computer

The Flight Management Computer (FMC) comprises the software and (simulated) hardware that executes most of the low-level spacecraft commands, such as firing the main engine and attitude control system, acquiring camera images, and so on. This subsystem determines which engines are used, and for how long, in order to execute the movement actions of the spacecraft. It does so by commanding a Main Engine Control System (MECS) and Attitude Control System (ACS) to change modes, in a specified sequence, in order to move.

The ACS is a cascaded mode-based controller. The inner-loop takes altitude and attitude rate commands as input, and is based on a phase-plane controller (Windall 1971) implemented on three axes. Several outer-loop ACS modes are wrapped around the inner-loop. For instance, lateral speed commands close the loop around commanded attitude, and

lateral position control is established through lateral speed commands. The MECS controller is also a mode-based controller, whose major operating mode is descent rate commands based on a simplified predictive model. During landing, the MECS mode is set to command descent rate; the initial descent rate is constrained to a 10 meters per second descent, and the final descent rate is constrained to be 1 meter per second. These rates govern both fuel consumption and power and energy use. The modes of the FMC are combinations of MECS and ACS modes; the modes employed in the mobility sequence are shown in Figure 1.

Data Columnator and Decimator

The Data Columnator and Decimator (DCD) is a database capturing a subset of the spacecraft state extracted from bus messages. The DCD reads spacecraft bus messages to capture relevant spacecraft parameters at a rate of 5 Hz and performs engineering unit conversions, smoothing, and timestamping. The DCD maintains these parameters and distributed them, on request, to other software modules. The spacecraft parameters stored by the DCD are shown in Figure 3.

Name	type	Name	type
Time	long	Location known	boolean
Location	6-DOF	Bus voltage	volts
Battery S.O.C.	amp/hours	Oxidizer level	kg
Fuel level	kg	Solar power	amps
Temperature (5)	$^\circ$ K	Valve positions (15)	{on off}
Engine status (5)	{on off}	Pressure (6)	PSI

Figure 3: The spacecraft state maintained in the DCD.

Visual Hazard Avoidance

The Visual Hazard Avoidance system takes a pair of images acquired from the (simulated) cameras mounted on the bottom of the spacecraft and analyzes the scene in order to find a safe landing spot. The system first constructs a disparity map, or correlation map, between the two images using a fast block stereo correlation algorithm. In the process, the algorithm marks image regions that are in excessive shadow or exhibit excessive glare such that the surface texture cannot be accurately correlated. Next, this disparity map is converted into a digital elevation map of the landing site, taking advantage of the known geometry of the cameras and the corresponding camera models. The resulting scene is analyzed to find a region large enough for the spacecraft to land in that is sufficiently flat (in this case, slopes of $\leq 5^\circ$), has no obstacles in it, has no missing pixels in it (i.e. regions of uncertain geometry due to shadows, glare, or failed correlation), and is as close as possible to the center of the scene. We used the NASA Vision Workbench to implement this software [1].

[1]This software is publicly available; the Users' Guide can be found at http://ti.arc.nasa.gov/visionworkbench/2.0.alpha4/VisionWorkbook-2008-01-15.pdf.

Other simulated flight software functions

The flight software also contains other critical functions that were simulated at much lower fidelity. These functions include acquisition of images from the cameras, and 6 degree of freedom (6-DOF) pose estimates from a Kalman filter using data from the star tracker and radar altimeter. Finally, we simulated the spacecraft bus and telemetry system using a combination of inter-process communication (IPC) using POSIX threads, with XML formatted bus messages.

Reflexive Intelligent System Health Monitoring

The Reflexive Intelligent Health System Monitoring (ISHM) component is a simple fault detection system for the spacecraft. It monitors spacecraft parameters to look for problems that can be calculated using basic logical inference. For example, a sudden and sustained drop in available power indicates a problem with the spacecraft power system that might prevent execution of the mobility plan. In addition, the RISHM system accepts heartbeat messages from the higher level software functions, and can either detect that these software functions are permanently disabled, or attempt to restart them in the event of transient failures. The RISHM component does not perform any mode reconfiguration. We used the Spacecraft Health Inference Engine (SHINE), developed at NASA's Jet Propulsion Laboratory (James 1997) (James & Sima 2008) to implement the RISHM.

Executive

The first, and arguably most important, of the higher level software components is the Executive. Traditional spacecraft management involves manual creation and validation of simple command sequences on the ground after inspection of science or engineering data, and subsequent uplinking them to spacecraft for execution. The Executive maintains a general mission plan, including the conditions under which various parts of the plan are considered complete, and manages the execution of that plan according to a system model that describes how the system works, coupled with high-level goals to achieve.

The Executive first performs the subsystem checkout. It waits to perform mobility planning until the subsystem checkout is successfully completed and the spacecraft's location has been determined. Finally, it waits to perform the mobility operation until the mobility plan has been built. The Executive commands underlying flight software functions, and must be able to react to a variety of other events from the spacecraft, e.g. invoke planning when localization is complete, or halt execution of the plan in the event of any one of a number of failures. The Executive is charged with consulting the higher-order ISHM system at a rate of 1 Hz to ensure there are no faults, as well as checking to ensure there are no faults prior to executing the mobility operation. We used the Intelligent Deployable Execution Architecture (IDEA) software (Muscettola *et al.* 2002), developed at NASA Ames Research Center, as the Executive.

Planner

The Planner is invoked by the Executive after the position of the spacecraft is determined (referred to as localization), and

initializes the Planner with the current location and destination, and the current propellant and oxidizer levels. It estimates the propellant and oxidizer consumption for the downrange motion operation, then estimates power consumption and power available from the solar panels during the hop by simulating the controller described in Figure 1. Finally, the planner determines if there are any resource deficiencies preventing a successful mobility operation. If solar power is not available, the planner must check that total energy consumption does not exceed energy in the battery. In the worst case, there will be no place to land, and the spacecraft will have to return to its takeoff location, requiring sufficient fuel and energy reserves. We use the Extensible Universal Remote Operations Planning Architecture (EUROPA), augmented with the abstraction of the MECS, to implement the Planner (Frank & Jónsson 2003). We note here that IDEA is implemented using EUROPA, about which more later.

Deliberative Intelligent System Health Monitoring

The final software component is Deliberative Intelligent System Health Monitoring (DISHM). This is a more sophisticated technology than the RISHM; it is capable not only of detecting faults, but of performing a search over possible diagnoses to determine what caused the fault based on a combination of observations of the spacecraft and an internal model of the spacecraft subsystems. For our work, this system was primarily used to monitor for faults in the main engine and attitude control system; inputs include pressure, temperature and on/off readings from the tanks, valves, and regulators. We use the Hybrid Diagnosis Engine (HyDE) developed at NASA Ames Research Center as the DISHM system (Narasimham & Brownstone 2007).

Software Function by Mission Phase

In this section we will describe the interactions between the software components during each of the simulated mission phases. We will also describe how the higher level reasoning components are configured in order to accomplish their principal functions.

Post-Landing Checkout

The post-landing checkout phase is accomplished by simulating the checkout of the MECS and ACS hardware, and checkout of the software components. This mission phase exercises IDEA, SHINE and HyDE (while the DCD is also involved, we postpone the description to the next section.) IDEA first simulates commands to the hardware, then requests the current snapshot of the spacecraft from the DCD; the return message is then parsed to ensure that the MECS and ACS hardware status is as expected. In order to check out HyDE and SHINE, IDEA commands HyDE and SHINE to perform their internal self-tests and report status back to IDEA.

IDEA is a model-based executive. IDEA models capture concurrent threads of execution, possible states each thread of execution can take on, and rules governing the allowed transitions between states that control what the agent does in response to events. Essentially, the spacecraft operating plan is transformed into rules that govern the legal behavior of the spacecraft. IDEA rules state that a thread is allowed to start or end a specified state S if some condition C is true. This can be thought of as a "blocking wait" semantics; for example, S is "waiting" to start until the relevant part of C is true. When writing the IDEA model, we first describe the concurrent threads of execution for the IDEA agent, and the possible states this thread of execution can take on. Part of IDEA's model is described below:

```
enum Nominal_Modes{
  checkout,
  post_checkout,
  ...
}
enum StepStatus{
  ok,
  failed,
}
class Agent_Mode{
  predicate Agent_Standby{}
  predicate Agent_Nominal{
   Nominal_Modes mode;
   StepStatus status;
  }
  predicate Agent_Shutdown{}
}
```

Thus, the IDEA agent mode timeline `Agent_Mode` can be in one of a set of states defined by the predicates `Agent_Standby`, `Agent_Nominal` and `Agent_Shutdown`, coupled with the ground values of the parameters of `Agent_Nominal`. Similarly, we declare the allowed states of the MECS, ACS, FSW, HyDE and SHINE. Next we write the rules governing the legal transitions. For example:

```
Agent_Mode::Agent_Nominal{
  if (mode==Nominal_Modes.post_checkout){
   starts_during(HyDE_Commands.HyDE_Ready);
   starts_during(SHINE_Diag.SHINE_Diag);
   starts_during(Agent_SelfTest.SelfTest_Idle);
   starts(FSWSimulator.StartScenario);
  }
}
```

This rule says that the thread `Agent_Mode` can transition to state `Agent_Nominal` with `mode=Nominal_Modes.post_checkout` when `HyDE_Commands` is in state `HyDE_Ready`, `SHINE_Diag` is in state `SHINE_Diag`, and `Agent_SelfTest` is in state `SelfTest_Idle`, and thread `FSWSimulator` has just transitioned to `StartScenario`. Also, thread `Agent_Mode` can transition out of `Agent_Nominal` with `mode=Nominal_Modes.post_checkout` unconditionally. The temporal relations (`starts_during` and `starts`) qualify the states that hold on the other timelines in the conditions, governing when a mode may start and when it may end.

IDEA uses a reactive planner (implemented in the EUROPA framework) to enact mode transitions. The planner is woken up every clock tick, checks every thread to see if a transition is allowed, and "eagerly" performs every legal transition. The rules act as constraints on the reactive

planner that IDEA uses to issue commands to different concurrent threads of execution to change state. Generally, the planner must be augmented with a variety of heuristics. For our application, the bulk of the heuristics specify that plan variables will be assigned when IDEA receives system messages. A small number of the heuristics predict that commands that could fail will succeed. Finally, a small number of heuristics stipulate the order that transitions are enforced; in this case, IDEA prefers to handle transitions on the `Agent_Nominal` thread, since that is the driver of most of the other threads in the system.

IDEA requires custom software to determine when states on threads require messages to be set to other software components. This software is collectively known as the AgentRelay. In our work, we mapped certain states to messages to the Flight Software (including SHINE), and other states to messages sent to EUROPA or HyDE.

SHINE is designed to be a very fast spacecraft fault detection system. It compiles mathematical inferences into highly optimized code designed to run onbard spacecraft. For our purposes, SHINE was used only for fault detection. IDEA receives fault detection messages from SHINE; there is no other content in the messages, so the interface to SHINE is simple. Additionally, IDEA sends heartbeat messages to SHINE; the only content besides the originator is a timestamp. In the event that one of the higher-order software modules, like IDEA, fails and ceases sending heartbeat, SHINE detects this and can restart these software functions. We simulated a failure of IDEA, which was detected by SHINE; SHINE then simulated the restarting of IDEA by sending IDEA a message to resume operations.

If IDEA receives notification of a hardware fault, it requests a diagnosis from HyDE. HyDE is a very powerful and general diagnosis engine. HyDE uses a model of system operations that captures the probability of various faults given that system commands are issued in particular system states. HyDE can generate one or more fault candidates, generate only the most likely fault candidates, and has a variety of potential means of interfacing with other systems. For this application, IDEA requests that HyDE return a single, most probable diagnosis should there be a problem. IDEA can, in principle, take contingent action based on any diagnosis of any component, but in practice the only response is to transition to a fault state that precludes mobility planning, or if the plan is done, precludes taking off. HyDE's return states (except for unknownFault) are enumerated in Figure 4.

Component	HyDE modes
Valve	open,closed, stuckOpen,stuckClosed
Regulator	nominal, regulating,regulatingHigh
Pressure Sensor	nominal, shortCircuit

Figure 4: The possible failure modes of ME and ACS components.

We model all the possible return values from HyDE in IDEA, and then use IDEA's rules to map return values to behaviors. Recall that this requires representing every failure for every component of the system.

Mobility Planning

Mobility planning commences when localization and checkout are complete and successful. We did not model localization, but simulated the process by sending a message informing IDEA that the spacecraft position had been determined. In this mission phase we describe the interactions of IDEA, EUROPA, SHINE and the DCD. Part of IDEA's rule for initiating mobility planning is shown here:

```
ACS_Expert::HAL_Planning{
  met_by(Spacecraft_State.Request_Frame s0);
  eq(s0.status, ReturnStatus.OK);
  eq(s0.oxidizer_level, oxidizer_level);
  eq(s0.propellant_level, propellant_level);
  eq(s0.battery_charge, battery_charge);
  ...
```

The thread `ACS_Expert` can transition to state `HAL_Planning` when thread `Spacecraft_State` transitions out of `Request_Frame`. When `Spacecraft_State` is in mode `Request_Frame`, IDEA sends a message to the DCD to get the current snapshot of the spacecraft. After the message is sent, IDEA waits until it receives a return message, containing the current spacecraft state from the DCD (Figure 3). This message is processed to fill out the parameters of the `Request_Frame` predicate. If the variable `status`, mapped from the DCD variable "Location Known", has value `failed` then localization has not occurred yet; IDEA waits 10 seconds, then polls the DCD again. If the status is `OK`, then the `HAL_Planning` predicate on the `ACS_Expert` class is allowed to start. This, in turn, causes a request for a plan to be sent from IDEA to the mobility planner. The spacecraft state from the DCD is sent to the planner from IDEA to initiate planning. Once the mobility planning is complete, as long as the plan was built successfully, the spacecraft is now ready to move.

We now describe the mobility planning in more detail. In order to estimate fuel consumption and hop duration, the planner makes use of an *abstraction* of the FMC. This estimate uses the initial and final positions to calculate the hop distance, which in turn is used to calculate the hop time, power and energy consumption, and total propellant and oxidizer consumption using the mode controllers shown in Figure 1. For example, fuel consumption for the ballistic trajectory (phases 4-6 of Figure 1) is calculated assuming the initial ballistic burn and braking burn are performed when the spacecraft is at $\theta = 45^o$ to the gravity gradient. The spacecraft is assumed to be stationary at some arbitrary height after an initial vertical burn, and is assumed to be stationary at the same height after the braking burn is completed. Lunar gravity $g = 1.62\frac{m}{s^2}$. Let $T = 600N \sin\theta$ (lateral translation velocity for the initial and braking burns). Let K be the estimated downrange distance to be traveled during the mobility operation; for our purposes this is $\leq 1km$. Then the total duration of the hop is: $2A + B$ where $A = \sqrt{\frac{Km}{T\left(\frac{2T}{gm}-1\right)}}$ and $B = \frac{2}{g}\left(\frac{T}{m} - g\right)A$. The fuel consumption can be calculated using the engine performance characteristics coupled with the initial and braking burn times. For the MECS, for

example, the fuel flow rate is $0.258\frac{kg}{s}$ the spacecraft thrust duty cycle is 100% and the thrust duration is A. Similar calculations can be done for the ACS. To calculate the fuel consumption for hovering (phase 9 of Figure 1), assuming a $65kg$ spacecraft mass (i.e. the worst case). The engine duty cycle to maintain a hover is 17.5%. Thus, one second of hovering comsumes $.0453kg$ of fuel. The hover duration depends on image processing time and plan validation time. Timing estimates of the analysis of the scenes from the camera and selection of a safe landing spot were used to estimate total duration during which spacecraft hovers over the intended landing spot. A similar calculation is performed to calculate the fuel consumed by the initial ascent and the descent modes.

EUROPA's planner invokes the FMC abstraction that returns the total resources consumed, checks to see if the fuel and oxidizer use are possible given available resources, then generates the appropriate number of discrete plan steps to represent the estimated duration of the mobility operation. The planner uses a discretized representation of time and the estimates of main engine burn times to calculate when power is needed to open and close valves. The planner must allocate energy consumed to one of two energy sources; the solar panels, or the battery. Since the battery is not rechargable, the correct policy is to maximize energy used from the solar panels prior to draining the battery. An approximation of power available from the solar panels based on estimated spacecraft attitudes is used to generate a profile of available energy from the solar panels. This information is used to define the initial state of a planning problem. The EUROPA planner uses a declarative representation that is similar to that used by IDEA. The planner applies the abovementioned heuristic to ensure that energy is used from the solar panels first, and checks the constraints imposed by the total energy used and the total available battery energy. A simplified fragment of EUROPA's model is shown below. In the model below, the class `Smart_Spacecraft_Data` is populated with data from the FMC abstraction.

```
class Smart_Spacecraft_Data
  predicate Step_Data{
   int energy;
   int availSolarEnergy;
   int num;
  }
}
class Smart_Spacecraft
  predicate Step{
   int num;
   int solarEnergy;
   int batteryEnergy;
  }
}
Smart_Spacecraft::Step{
  meets Step next;
  addeq(num, 1,next.num);
  equals(StepData s);
  eq(s.num num);
  addeq(solarEnergy, batteryEnergy,s.energy);
  leq(solarEnergy, s.availSolarEnergy);
  any(object.m_batteryPowerSystem.consume tx);
  eq(tx,batteryEnergy);
}
```

Mobility Plan Execution

Every software component is exercised in mobility plan execution. During the first several phases of the mobility operation `Agent_Mode` takes on the predicate `Agent_Nominal` with parameter `mode` taking on value `hop_to_wpt_1`. IDEA's rule is:

```
Agent_Mode::Agent_Nominal{
  if (mode==Nominal_Modes.hop_to_wpt_1){
   equals(GNC_Commands.GNC_Command s0);
   eq(s0.mode, GNC_Command_Mode.HopToPosition);
   meets(Agent_Mode.Agent_Nominal s1);
   eq(s1.mode, Nominal_Modes.hop_take_images);
  }
}
```

This rule says that on the `Agent_Mode` thread, `Agent_Nominal` with mode `mode==Nominal_Modes.hop_to_wpt_1` can start when `GNC_Command` starts, and can end when `GNC_Command` ends (specified by the `equals` temporal relation. It also specifies that the next state on the `Agent_Mode` thread must be `Agent_Nominal` with `mode=Nominal_Modes.hop_take_images`. The predicate `GNC_Command` on class `GNC_Commands` with parameter `mode` taking on value `HopToHoverPosition` results in IDEA sending a message to the FMC. This command, in turn, results in the execution of the first part of the mobility plan shown in Figure 1. The FMC terminates this command at Step 9, with the FMC in mode HoverTo, the spacecraft hovering and awaiting further instructions. The FMC sends a message back to IDEA, which then ends this predicate. `Agent_Mode` then takes on the predicate `Agent_Nominal` with parameter `mode` taking on value `take_images` after completion of the previous mode. IDEA's rule is:

```
Agent_Mode::Agent_Nominal{
  if (mode==Nominal_Modes.hop_take_images){
   equals(Image_System.Take_Image s0);
   meets(Agent_Mode.Agent_Nominal s1);
   eq(s1.mode, Nominal_Modes.hop_check_wpt_2);
  }
}
```

The predicate `Take_Image` on thread `Image_System` causes IDEA to send a message to the flight software to acquire the camera images, perform the subsequent image analysis, and generation of the actual waypoint at which to land[2]. This process terminates when the image analysis finds a safe place to land; the coordinates of the landing location are then inserted into the DCD via the telemetry stream (in this case simulated by having the image analysis system directly insert the coordinates into the DCD). The simulation sends a message to

[2]We did not simulate the scenario in which there was no safe place to land.

IDEA, which ends the Agent_Nominal predicate with mode==Nominal_Modes.hop_take_images, and starts a new Agent_Nominal predicate with parameter mode taking on the value hop_check_wpt_2. The final IDEA rule we discuss is:

```
Agent_Mode::Agent_Nominal{
  if (mode==Nominal_Modes.hop_check_wpt_2){
   equals(ACS_Expert.Validate_Waypoint s0);
   meets(Agent_Mode.Agent_Nominal s1);
   eq(s1.mode, Nominal_Modes.hop_to_wpt_2);
  }
}
```

In this case, the initiation of the new Agent_Nominal predicate starts a predicate Validate_Waypoint on class ACS_Expert. This predicate causes IDEA to send another message to the planner to ensure that the spacecraft can land given available resources. While we do not discuss this rule in detail, the Validate_Waypoint predicate results in a request to the DCD to get the current state of the spacecraft, and the relevant parts of the state (including the specified landing position) are passed to the planner for validation. This planning problem is much like the previous problem, only much simpler to solve. Once the planner is done, IDEA is notified and commands the FMC to descend.

Evaluation and Discussion

In order to evaluate the system, we use a medium-fidelity physics simulation of the spacecraft. We use Apollo SPI camera images of the Aristarchus Plateau area of the Moon at 200 cm resolution to exercise the vision system; because of this, we could only simulate one lighting condition. Due to an unusual feature of the simulated image capture, we could simulate a variety of lost pixel conditions, thereby changing the set of legal landing sites. Faults in the MECS and ACS are injected using software simulation. Finally, as described above, we also simulate IDEA failing and being restarted.

In the presence of hardware faults, IDEA reports detection of the relevant fault, and the spacecraft remains idle. When IDEA fails, SHINE simulates the restart of IDEA, and execution of the scenario proceeds. When all proceeds normally, the spacecraft executes the mobility plan under complete control of the higher level automation functions. We tested every fault scenario, failing every component, including IDEA, during checkout and prior to mobility operations [3], and ran the nominal scenario with the spacecraft traveling 1 km downrange with several missing pixel scenarios. These experiments were conducted with one computer running all autonomy software, while a second computer ran the FMC, vision system, all kinematics and dynamics simulations as well as a visualization of the spacecraft.

In the following paragraphs we discuss the knowledge capture, modeling and software integration issues encountered during this work. The IDEA model consisted of 22 classes, 46 predicates, 55 rules. Roughly $\frac{1}{3}$ of this is a replica of the HyDE models, $\frac{1}{3}$ governs the invocation of the mobility planner, and the rest represents the actual spacecraft plan. This may seem like overkill due to the size of the model and the fact that IDEA simply stops the execution of the plan in the event of any off-nominal hardware behavior. However, we showed that IDEA is capable of replicating the HyDE model. Further, interpreting HyDE and SHINE responses must be captured either in special-purpose software between HyDE and IDEA, or it can be accomplished directly by IDEA, which is the course of action we chose. As mentioned, IDEA's reactive planner heuristics are fairly simple, and consist primarily of ensuring IDEA variables set by messages are not set by the planner, and additional heuristics to transition states of the overall spacecraft operating plan prior to other, lower level threads. Roughly 2000 lines of IDEA code were needed to send and recieve messages to the other software components.

The EUROPA model for the mobility planner consisted of 2 classes, 2 predicates, and 6 rules. The balance of custom software was tilted in the other direction, with 10000 lines of planner code (most of which replicated the FMC), with a little more for heuristics to tune the EUROPA planner. The bulk of the knowledge needed to build this planner concerned the FMC abstraction, with the rest concerning the interaction of the planner and the visual hazard avoidance system. The overall planning algorithm can be thought of as a 3-phase algorithm. First, the FMC is simulated to calculate the propellant and oxidizer consumption, power consumption, and power availability. Second is a simple resource check on propellant and oxidizer, and finally, the power resource assignment and power and energy resource check. The fuel and energy calculation formulas could be represented in a declarative way by EUROPA (or other planners), but we simply integrated the FMC abstraction software with EUROPA and only used it to solve the power allocation problem. The decisions to make in this planning problem, whether to use battery or solar panel power during the hop, are resolved completely by the heuristic (since the battery is not rechargeable). EUROPA's explicit representation of time, concurrent threads (for power availablity vs resource assignments) and resources in the model proved helpful while developing the planner, but as mentioned above, provided no leverage when handling the FMC abstraction. It should be apparent that the planning problems were not very challenging from a computational complexity point or heuristics of view; again, the principal challenge was integrating the EUROPA software with the FMC simulation software.

Our intention was to prototype an intelligent agent that could be fielded on a spacecraft. We close with a discussion of system resource requirements for spacecraft. A typical conservative spacecraft design requires 50% CPU and memory margin; for comparison, JPL's ASPEN planner flying on EO-1 was allocated 16 M. The software we designed required 64 M and was simulated on two high-end desktop machines. The fastest processor flying is 800 Mhz (e.g. BAE), so we estimate that the current software would require 2 processors onboard with 64 M each. In short, it is plausible that

[3] The dynamics simulation was not up to the task of simulating the physics of failures during mobility, so we did not test these cases.

such a system could run a mission such as the one we describe, but there is considerable work left to integrate all of the components more seamlessly and field them on a flight processor running a real-time operating system.

Conclusion and Future Work

We have presented an intelligent agent for Lunar exploration, constructed from modern automation technologies. The agent is centered around a model-based execution system that controls an automated planner and ISHM subsystem, and commands lower level flight software functions to execute a mobility plan. We have designed realistic scenarios and tested the complete system in software simulations of medium fidelity. The integrated system was shown to perform as expected in all of our scenarios. Since we used existing model-based technology to build the agent, the principal challenges were knowledge capture and software systems integration. The bulk of the modeling challenge was in building IDEA's models, with roughly equivalent effort spent modeling the diagnosis system, activity plan and mobility planner. The IDEA software proved to be easily integrated with the other software modules in the system. By contrast, the mobility planning required significant augmentation of EUROPA with the FMC abstraction, but modeling was much simpler.

While the higher level autonomy architecture resembles that used in the Remote Agent (Muscettola *et al.* 1998), overall the control strategy is more nuanced. Nominally, IDEA is in control, and invokes the planner and other experts, in a manner similar to APEX (Freed 1998). However, the lower level fault recovery software can take control at critical times, such as in the event of IDEA's failure. This is not a usual feature of the three-layer architecture, but is inspired by a more traditional flight software perspective, emphasizing control by simple functions that are highly trusted.

One future goal is to test traverses of shorter distances than 1 km. Very short traverses are actually quite difficult, and traverses with differences in takeoff and landing sight elevation also caused problems during development of the MECS and ACS. Due to the size of the Lunar terrain patch available, we could not vary the lighting effectively. Doing so would be essential in order to demonstrate the viability of the mission concept throughout different phases of the lunar day, and to ensure that the integrated system is robust to performance variations of the vision system, both in computation time and landing site selection variability. Another future goal is to test the system on a very realistic failure scenario, in which one (or more) solar panels fails. Since the panels provide primary power during the mission, a panel failure imposes a non-trivial constraint on the spacecraft orientation upon landing. When coupled with the fixed orientation of the science instrument on the spacecraft body, this also imposes a significant constraint on the landing location. Finally, creating more sophisticated behaviors to robustly guide the spacecraft in the event no safe place to land will require more sophisticated planning. Using prior knowledge of the terrain leads to choices of scouting contingent landing sites prior to traveling to the goal, which drive more interesting planning problems.

Acknowledgments

This project was the work of a large team of people: E. Balaban, L. Brownston, K. Greene, K. Gundy-Burlet, M Hancher, C. Ippolito, M. James, P. Jarvis, G. Limes, R. Mackey, C. Plaunt, A. Sweet, R. Tikidjian, R. Watson, and B. Vijaykumar. We also gratefully acknowledge the support of K. Hicks, S. Uckun, as well as the NASA Jet Propulsion Laboratory's Team-X Mission Designers.

References

Chien, S.; Sherwood, R.; Tran, D.; Cichy, B.; Rabbideau, G.; no, R. C.; Davies, A.; Mandel, D.; Frye, S.; Trout, B.; Shulman, S.; and Boyer, D. 2005. Using autonomy flight software to improve science return on earth observing one. *Journal of Aerospace Computing, Information and Communication*.

Frank, J., and Jónsson, A. 2003. Constraint-based interval and attribute planning. *Journal of Constraints Special Issue on Constraints and Planning*.

Freed, M. 1998. Managing multiple tasks in complex, dynamic environments. In *Proceedings of the* 15^{th} *National Conference on Artificial Intelligence*.

James, M., and Sima, H. 2008. An introspection framework for fault tolerance in support of autonomous space systems. In *Proceedings of the IEEE Aerospace Conference*.

James, M. 1997. Spacecraft health inference engine. New Technology Report NTR-20166, NASA Jet Propulsion Laboratory.

Johnson, A.; Wilson, R.; Cheng, Y.; Gougen, J.; Leger, C.; Sanmartin, M.; and Matthies, L. 2007. Design through operation of an image-based velocity estimation system for mars landing. *International Journal of Computer Vision* 74(3):319 – 341.

Knight, R. 2008. Automated planning and scheduling for orbital express. In *Proc. International Symposium on Artificial Intelligence, Robotics, Automation and Space*.

Muscettola, N.; Nayak, P.; Pell, B.; and Williams, B. 1998. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence* 103(1-2):5 – 48.

Muscettola, N.; Dorais, G.; Fry, C.; Levinson, R.; and Plaunt, C. 2002. Idea: Planning at the core of autonomous reactive agents. In *Proceedings of the 3d International NASA Workshop Planning and Scheduling for Space*.

Narasimham, S., and Brownstone, L. 2007. Hyde - a general framework for stochastic and hybrid model - based diagnosis. In *Proceedings of the 18th International Workshop on the Principles and Practices of Diagnosis*, 162 – 169.

Stanley, J.; Shendock, R.; Witt, K.; and Mandl, D. 2005. A model-based approach to controlling the st-5 constellation lights out using the gmsec message bus and simulink. In Arabnia, H., and Reza, H., eds., *Proc. Intl. Conf. Software Engineering Research and Practice*, 29 – 35. CSREA Press.

Windall, W. S. 1971. Lunar module digital autopilot. *Journal of Spacecraft and Rockets* 8(1):56–62.