A Reconfigurable Testbed Environment for Spacecraft Autonomy

Jeffrey Biesiadecki Abhinandan Jain
Jet Propulsion Laboratory/California Institute of Technology
4800 Oak Grove Drive M/S 198-326, Pasadena, CA 91109 USA

Abstract

A key goal of NASA's New Millennium Program (NMP) is the development of technology for increasing spacecraft on-board autonomy. Achievement of this objective requires the development of a new class of ground-based autonomy testbeds that can enable the low-cost and rapid design, test and integration of the spacecraft autonomy flight software.

*This paper describes the development of an Autonomy Testbed Environment (*ATBE*) for the NMP Deep Space I comet/asteroid rendezvous mission. This simulation testbed has been designed to enable rapid design of flight modules, early identification of performance and design problems, resolution of integration issues, and thorough ground testing for reducing mission-risk.* ATBE*'s simulation requirements span a wide range of engineering platforms, functional and fidelity models, failure modes, test scenarios, and durations. The flight software modules under development include attitude control subsystem, remote agent, autonomous navigation, and flight systems control. Conventionally, such testbed functionality has been met by the expensive and time-consuming development of multiple specialized testbeds. In contrast, the* ATBE *testbed has been designed to be reconfigurable for multiple user development and test needs. The* ATBE *software will also be integrated with the support equipment for hardware-in-the-loop tests and system level integration.*

The ATBE *spacecraft simulator includes a high fidelity real-time dynamics simulation package integrated with simulation models for several of the hardware devices and interfaces on the spacecraft. The testbed incorporates existing in-house and third-party software, integrated within an object-oriented architecture. This design enables easier maintainability and usability, and perhaps most significantly this flexible design is geared to handle continual evolution in model requirements, functionality and fidelity. The simulation interfaces are highly configurable to allow swapping in and out of hardware as needed. The testbed has been instrumented from the start to provide a high degree of visibility into the simulation status with capabilities to peek/poke/checkpoint/resume model states, and includes some graphical user interfaces as well.*

1 Introduction

A key goal of the *New Millennium Program (NMP)* is the development of technology for increasing spacecraft on-board autonomy [1]. Achievement of this objective will require the development of a new class of ground-based autonomy testbeds that can enable the low-cost and rapid design, test and integration of the radically new autonomous system flight software. This paper describes the the development of such a new class of ground-based autonomy testbed – the *Autonomy Testbed Environment (*ATBE*)* – that can enable the low-cost and rapid design, test and integration of autonomy spacecraft flight software. This ATBE architecture is being designed to accommodate the large variety of autonomy flight software functions that need to be tested and validated before flight. In particular, the architecture enables: rapid design of flight modules, early identification of performance and design problems, resolution of integration issues, and thorough ground testing for reducing mission-risk. The design and implementation of ATBE is being carried out for the New Millennium Deep Space I mission [2].

ATBE's simulation requirements span a wide range of engineering platforms, hardware and environmental models, failure injection capability, test scenarios etc. Traditionally such a range of testbed functionality has been met by the expensive and time-consuming development of multiple specialized testbeds. In contrast, the ATBE testbed is being designed to be *reconfigurable* to meet these multiple user development and test needs including:

- use of the testbed by all of the various autonomy flight software sub-systems for code development and verification.
- different interfaces and fidelity levels to support the variety of testing requirements.
- use of the testbed across desktop workstations to real-time hardware-in-the-loop environments for flight software testing.
- easy evolution and maintenance of the simulation functionality to accommodate the continual change in model requirements, functionality, fidelity and interfaces.

An object-oriented simulation architecture has been designed to handle models ranging from time-critical models such as spacecraft dynamics simulators, analytical models for hardware devices, interfaces and electronics, and event-driven instrument simulators. The base model class has been designed to provide a high degree of instrumentation and visibility into the internals of the simulator. Moreover, the testbed supports the ability to turn on and off models and easily change the data flow of the simulation variables. The structure of the model database is highly modular to allow the easy change and replacement of selected parts of the simulator without global impact on the overall simulator. Indeed this aspect of the architecture allows the easy switching between software simulations of devices and real test hardware in the loop as needed. The ATBE spacecraft simulator includes the DSHELL high fidelity real-time dynamics simulation package with simulation models for several of the spacecraft hardware devices. The testbed incorporates several tools which include in-house as well as commercial software.

This paper describes the goals, current status and future plans for the development of ATBE for the New Millennium Deep Space I mission.

2 Architecture of the ATBE Spacecraft Simulator

Reconfigurability has been a key driving requirement for the design of the ATBE architecture. Reconfigurability comes in several different flavors driven by different development and test needs. Some of these drivers for the ATBE architecture are:

Concurrent engineering s/w development process: The flight software development is following a concurrent engineering development process. The evolution of the ATBE needs to at all times support flight software development and test from the early to the more mature phases. The build-a-little test-a-little strategy also requires the architecture to be flexible and adaptable enough to support continual evolution of the simulation models and interfaces over the development period.

Multiple platform development: The most mature testbed environment for system level and flight software testing will include hardware in the loop components as well as realistic flight-like interfaces to ATBE. However this testbed will come on line only late in the software development process, and its use will be primarily for system level testing. So ATBE is being designed to run under different platforms and environments ranging from Unix desktop workstations to VxWorks/68040 real-time environments. Consequently, there are a large number of configurations of ATBE available to meet the simulation needs of flight software developers. The choice of environment is up to the flight software developer and is dictated by the development and test needs of the day. Support and development of ATBE for the multiple platforms will continue all the way until launch.

Multiple interfaces: The interfaces between ATBE and the flight software for closed-loop testing have been kept flexible to accommodate different fidelity levels. Thus the simplest interface for the 1553 bus interface treats it as merely a transport layer and uses a high-level inter-process communication mechanism to close the loop between the flight software and the simulation. Higher fidelity versions of this interface provide more detailed software simulations of the 1553 behavior, as well as actual interfaces to commercial and flight-like 1553 hardware. The ability to select the appropriate level of fidelity provides many options for flight software developers and enables significantly larger amounts of concurrent testing.

Simplified simulations: During the development process, flight software subsystems such as the attitude control system [3], autononous navigation [4], or the flight system control, etc., do not have a need for the full simulation capability. Indeed, there are definite advantages to simplifying and tailoring the simulation environment to the needs of the specific subsystem. To support this need, the ATBE architecture has been designed to support different simulation *configurations* and interfaces that are easily selectable at run-time. For instance, the attitude control system developers prefer to carry out initial performance and design tests for the run the closed-loop simulation while bypassing the 1553. The "with bus" or "bypassed bus" simulations are run-time selectable in the ATBE architecture.

Visibility: The ability to peek, poke and monitor simulation variables in real-time is essential for the test engineer. The simulation software has to be instrumented to support this type of visibility. However, due to the vast number and types of simulation data, the choice of what and when to monitor and change data is very much dependent on the test objectives. ATBE provides a rich class of commands to peek, poke and monitor virtually every significant variable in the simulation software and the test engineer has the flexibility to tailor the visibility functions as needed.

Software development: The rapid pace of the software development and the continual change in the simulation models imposes significant challenges on the software development and configuration management process. On the one hand, the flight software engineers have to be able to access different versions of ATBE compatible with their development needs. On the other hand, the ATBE software engineers internally need to be able to make changes to the different pieces of ATBE without being effected by or effecting other developers' efforts. The size and build time for the software make this a non-trivial but critical housekeeping task. The ATBE team has developed the YaM software development process (described in Section 5) to address and solve this problem.

Another important aspect of the ATBE architecture is the high performance of the critical real-time core of the simulator in addition to the several event-driven simulation models. Also, a mini-environment compatible with the ATBE architecture has been developed to support the unit development and test of simulation models. This mini-environment not only provides a convenient way for model builders to develop and test the models, but also makes it easy to migrate the modules into the ATBE environment for integration with the rest of the simulation.

The architecture design has at the outset emphasized a *tools* based approach. Given the slow maturation rates of new software, it was highly desirable to inherit and use existing and mature tools to form the bulwark of the ATBE architecture and focus the ATBE software development in knitting together these tools and implementing new feature into a usable flight software development and test environment. The architecture is also being designed for reuse in missions that follow DS1.

2.1 Architecture Design

The DS1 spacecraft simulator models are roughly categorized as those belonging to the real-time core and others that are event-driven non-real-time models in order to meet the critical real-time performance requirements on the simulation software. Real-time models contain functions that are executed every simulation heartbeat while event-driven models do most of their work in response to events or commands.

Real-time models include a module for propagating the spacecraft dynamics state (Darts), models for the various attitude control sensors and actuators (Dshell models), the interfaces to the 1553 bus, models for device electronics interfaces etc. Dshell is a spacecraft dynamics simulation tool which includes a library of analytical models for actuator and sensor hardware devices typically commanded by attitude control subsystem flight software. These models have continuous states, and are tied to the Darts dynamics compute engine. Dshell will be described in more detail in Section 4.

The event-driven models get executed only occasionally and run as separate processes. The different processes in the simulator typically communicate via messages. An example of an event-driven model is a scene generator which is used to simulate the on-board camera. This model does its work in response to a "take picture" command from the flight software and may take several minutes to create an image.

The high-level categorization and decomposition of sub-systems into real-time, Dshell and event-driven models is illustrated in Figure 1. The real-time models are implemented and inter-connected using the third-party tool *ControlShell*, from Real-Time Innovations, Inc. *ControlShell* provides a C++ base class for *components*, and allows for data-flow between components. Each component has an "execute" method, which is called each tick of the simulation. Components also have inputs and outputs - each tick they set their outputs based on their inputs. The order in

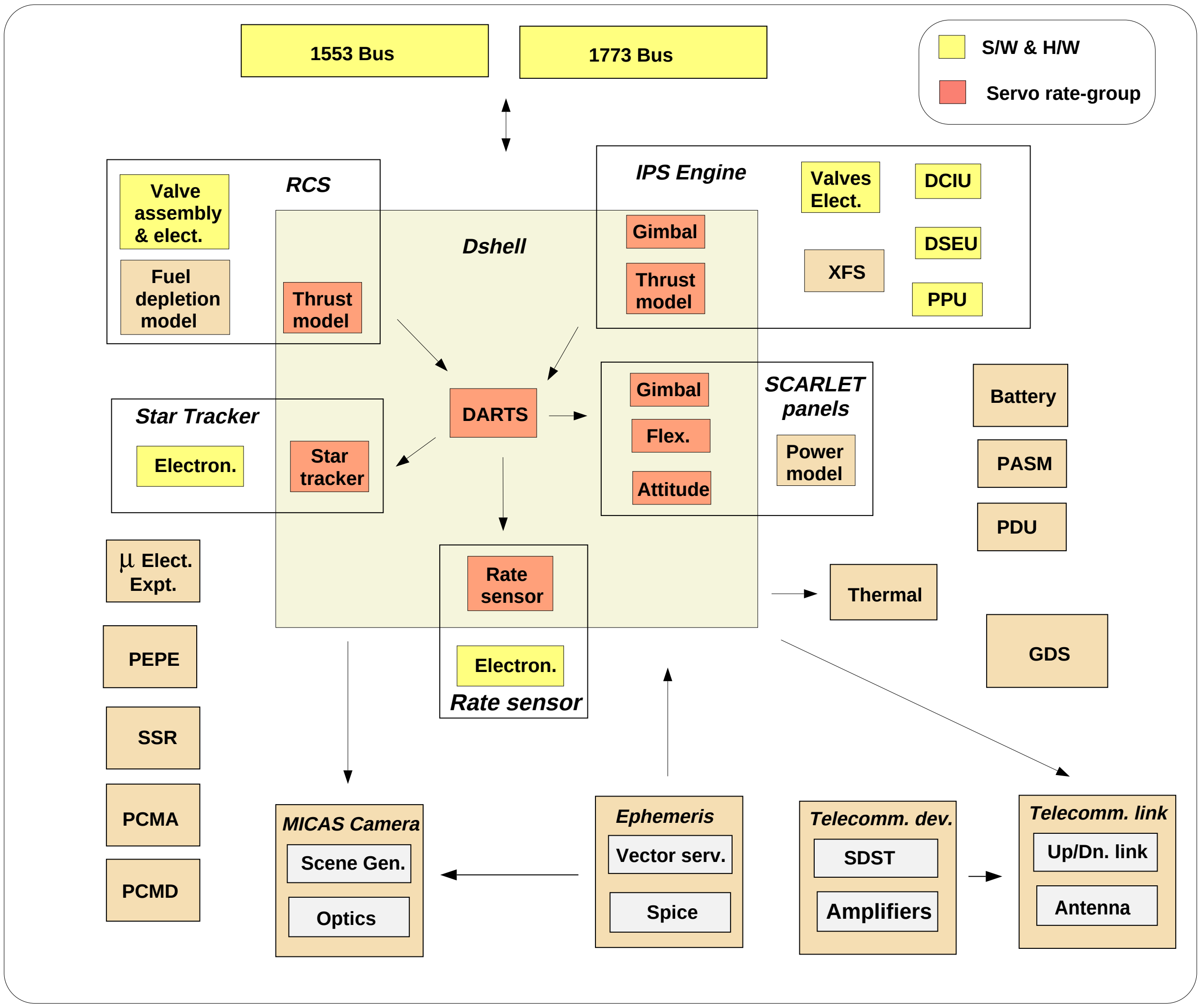


Figure 1: Simulation models categorization for New Millennium Deep Space 1 spacecraft

which the "execute" methods are called is, by default, determined from the dependencies implied by the data-flow. Graphical editors are used to define component interfaces and stub code, as well as to specify component inter-connections which are referred to as *signals*. A *Tcl* interface allows components to be activated/deactivated at run time, signal data to be manipulated and monitored, and a host of other useful interactions with the simulator.

A C++ device simulation class has been derived for all ATBE components, and includes capabilities to monitor and manipulate state data internal to the model through a *Tcl* interface. To register a state variable with the simulator, the model builder calls a function during initialization giving a name, data type, pointer, and number of elements. For this project, built-in C data types and arrays of these types are allowed for state variables, as well as integer enumerations. During the simulation, the user can call the *Tcl* **peek** command to look at the value of a state variable, and **poke** to change it. The syntax being:

peek *modelName stateName*
poke *modelName stateName value*

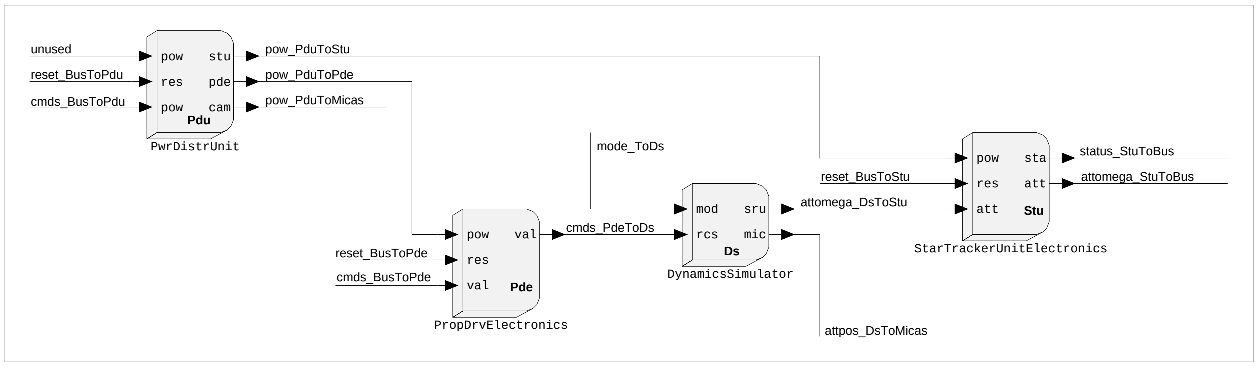


Figure 2: Example of simulator components viewed with *ControlShell*'s Data Flow Editor

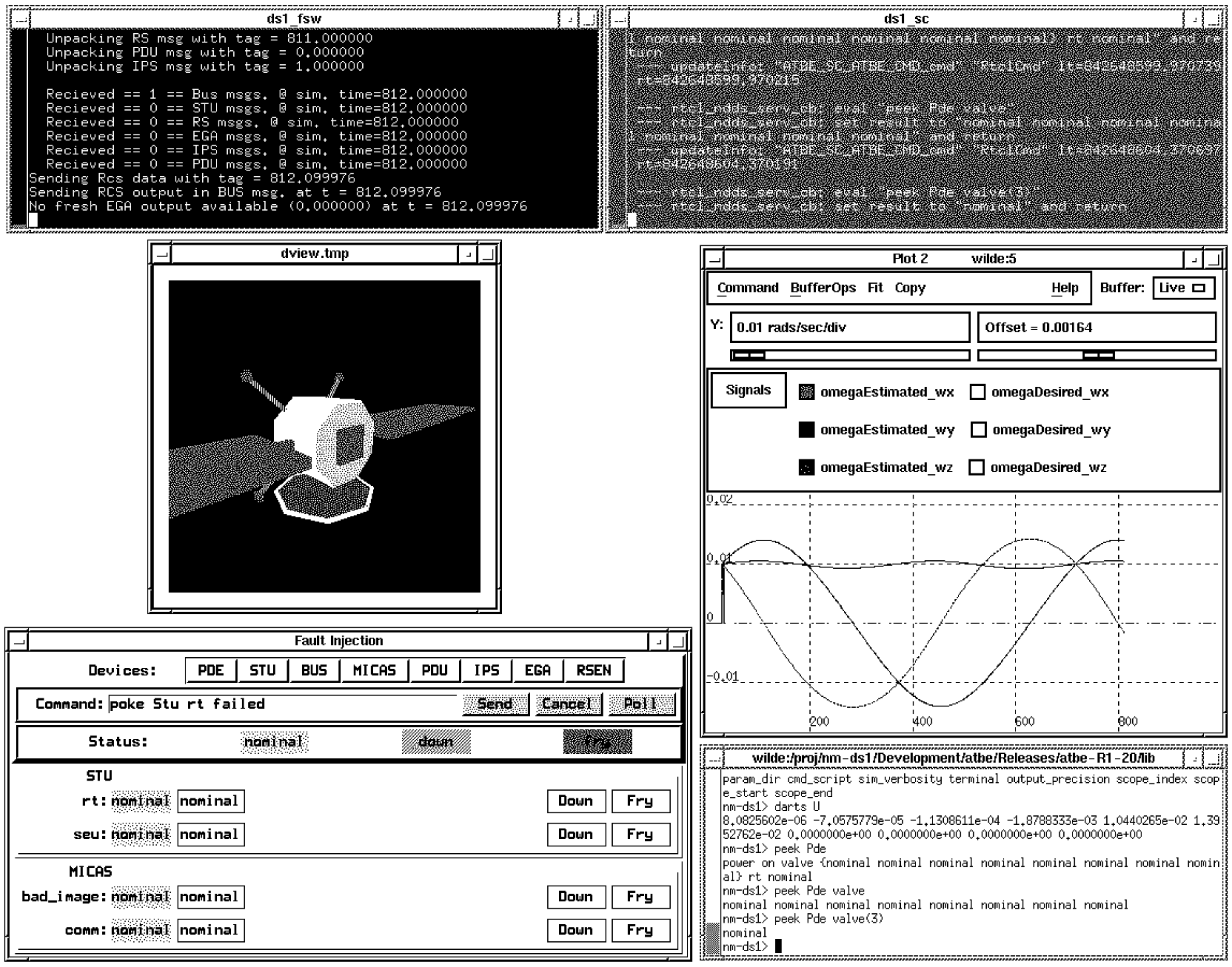
The simulation can be checkpointed, saving the values of all registered states as a series of **poke** commands in a file which can be sourced for a future run.

For components that correspond to actual devices on the spacecraft, we have a further refined subclass. It has support for a *power* line, device *reset*s, and failure states. From the model's "execute" method, functions can be called to determine whether or not the device is receiving power and whether or not a reset has occurred.

Failure states are implemented as a special kind of integer enumeration. All failure states may be set to a built-in nominal value. The model builder defines the values for mutually-exclusive failure modes. For example, a thruster might have a failure state which can be set to **nominal**, **stuckClosed**, or **stuckOpen**. This allows the inclusion of models that can monitor conditions and set failure states on its own. The manual injection of failures is carried out using the **poke** command. Likewise, a model may be written to fix a failure (i.e., set to **nominal**) on its own, or the user may correct it with **poke**. As a convenience, however, it is possible to define options such that the failures are automatically corrected when the corresponding device is power cycled, reset, or a time-out occurs.

In addition to **peek** and **poke** commands, lists of all components and their states can be obtained via the *Tcl* interface. This forms the basis for a GUI tool for injecting failures into the simulation. At start up the fault-injection tool queries the simulator about the implemented failure modes and dynamically generates panels and interfaces based upon the information. This allows developers to add or change failure modes in their simulation models without having to be concerned about making compatible changes to the fault-injection tool. Figure 3 shows the look and feel of a typical stand-alone run of the ATBE software.

A mini-environment has been developed to aid in the development and unit testing of simulation models outside the main ATBE simulation environment. The mini-environment allows the model developers to compile and run their models in a stand-alone mode with interfaces similar to those in the ATBE environment. Models are implemented by specifying initialization and tick functions to implement its functionality. When the model is to be added to the actual simulator, a *ControlShell* component for it is created and input/output lines to other components are added. The methods in the *ControlShell* component only need to call the initialization and tick methods of the model. The mini simulation environment is implemented as a library for the model developer. This environment has the same **peek** and **poke** commands, which also work for model inputs and



```
  Unpacking RS msg with tag = 811.000000
  Unpacking PDU msg with tag = 0.000000
  Unpacking IPS msg with tag = 1.000000

  Recieved == 1 == Bus msgs. @ sim. time=812.000000
  Recieved == 0 == STU msgs. @ sim. time=812.000000
  Recieved == 0 == RS msgs. @ sim. time=812.000000
  Recieved == 0 == EGA msgs. @ sim. time=812.000000
  Recieved == 0 == IPS msgs. @ sim. time=812.000000
  Recieved == 0 == PDU msgs. @ sim. time=812.000000
Sending Rcs data with tag = 812.099976
Sending RCS output in BUS msg. at t = 812.099976
No fresh EGA output available (0.000000) at t = 812.099976
```

```
l nominal nominal nominal nominal nominal nominal} rt nominal" and re
turn
 --- updateInfo: "ATBE_SC_ATBE_CMD_cmd" "RtclCmd" lt=842648599.970729
 rt=842648599.970215

 --- rtcl_ndds_serv_cb: eval "peek Pde valve"
 --- rtcl_ndds_serv_cb: set result to "nominal nominal nominal nomina
l nominal nominal nominal nominal" and return
 --- updateInfo: "ATBE_SC_ATBE_CMD_cmd" "RtclCmd" lt=842648604.370697
 rt=842648604.370191

 --- rtcl_ndds_serv_cb: eval "peek Pde valve(3)"
 --- rtcl_ndds_serv_cb: set result to "nominal" and return
```

```
Command: poke Stu rt failed
Status:        nominal        down        fry
STU
       rt: nominal  nominal
      seu: nominal  nominal
MICAS
bad_image: nominal  nominal
     comm: nominal  nominal
```

```
param_dir cmd_script sim_verbosity terminal output_precision scope_index scop
e_start scope_end
nm-ds1> darts U
8.0825602e-06 -7.0575779e-05 -1.1308611e-04 -1.8788333e-03 1.0440265e-02 1.39
52762e-02 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
nm-ds1> peek Pde
power on valve {nominal nominal nominal nominal nominal nominal nominal nomin
al} rt nominal
nm-ds1> peek Pde valve
nominal nominal nominal nominal nominal nominal nominal nominal
nm-ds1> peek Pde valve(3)
nominal
nm-ds1>
```

Figure 3: DS1 simulator and assorted tools

outputs. Additionally, simulation time can be advanced, power to the device turned on and off, the device can be reset, etc. Thus, the developer can **poke** values for model inputs, advance a step, **peek** at model outputs, run *Tcl* scripts to enable unit testing. While these capabilities are available in the full-up simulator, the mini environment has the advantages that it involves less code, is self-contained, and is faster and easier to run.

3 Simulation Models in the ATBE Simulator

The following is a brief summary of the various models in the DS1 simulator as shown in figure 1, and their functionality. Models for the fuel usage, power load, heaters, etc., are included in the device simulations as appropriate. Failure modes are also built into the models.

RCS: Models for thrusters, the latch assembly, the propulsion drive electronics (PDE), fuel tank pressure/flow-rate/consumption etc.

Rate sensor: A rate sensor model with bandwidth, drift and noise characteristics, and the A/D and electronics interface.

Star tracker: A star tracker model with sky coverage model, and electronic interface.

Sun sensor: A sun sensor model with the number of heads, their characteristics, modes, and electronics interface.

SEP gimbal: A model for the SEP gimbal actuator and encoder including its electronics interfaces.

SEP engine: An analytical model for thrust, flow rate, pressure, etc. for the SEP engine and its control unit.

Scarlet gimbals: A model for each of the Scarlet panel gimbal actuators and encoders including their electronics interface.

Scarlet solar panel power: An analytical model for the power generated as a function of spacecraft attitude and panel articulation.

Scarlet dynamics: A structural dynamics model for the Scarlet solar panel flexibility with values for the assumed modes, the vibrational frequencies and the damping ratios; the current structural dynamics modeling estimate is 5 modes/panel.

Spacecraft dynamics model: This model will define the kinematics and multibody dynamics model for the spacecraft including inertias, modes, hardware locations, etc.

On-board battery: Model for charge/discharge behavior of the on-board battery including an electronics interface for controlling its charging/discharging mode.

PDU, PASM: Models for the power switching logic and interfaces.

SSR: Solid state recorder model with definition of interfaces and data transfer mechanism.

Micas camera: A scene generator for generating image data as needed for autonomous navigation and science experiments.

Science instrument: Simulation models of the science instruments.

Vector server: A real-time module to supply earth/sun/asteroid vectors.

1553/1773 bus: A model for the bus operation.

Telecomm.: A model for the SDST transponder, power amplifiers, wave guides and switches etc.

Up/down link: Model channel integrity as a function of antenna earth-pointing angle.

4 Dshell Spacecraft Dynamics Simulator

Darts Shell (Dshell) is a multi-mission spacecraft simulator for development, test and verification of flight software and hardware. Dshell is portable from desktop workstations to real-time, hardware-in-the-loop simulation environments. Dshell (Figure 4) integrates the Darts S/C flexible multibody dynamics computational engine and a library of hardware models (for actuators, sensors and motors) into a simulation environment that can be easily configured and interfaced with flight software and hardware for various real-time and non real-time S/C simulation needs. The main goals of the Dshell environment are: to significantly reduce the software development

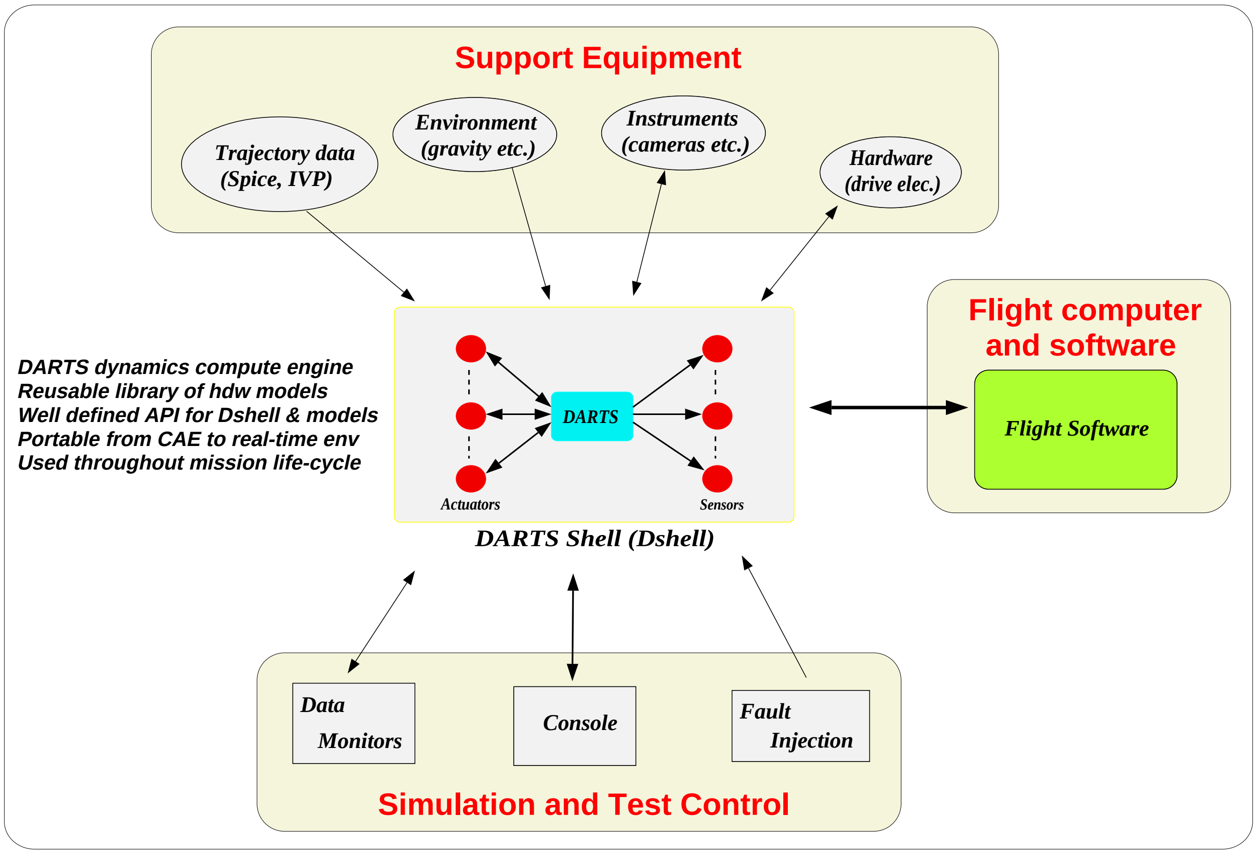

Support Equipment
Trajectory data (Spice, IVP)
Environment (gravity etc.)
Instruments (cameras etc.)
Hardware (drive elec.)
DARTS dynamics compute engine
Reusable library of hdw models
Well defined API for Dshell & models
Portable from CAE to real-time env
Used throughout mission life-cycle
DARTS
Actuators
Sensors
DARTS Shell (Dshell)
Flight computer and software
Flight Software
Data Monitors
Console
Fault Injection
Simulation and Test Control

Figure 4: Dshell architecture with Darts and device models

required to interface dynamics simulators, actuator and sensor hardware models and hardware-in-the-loop devices; to eliminate the need for separate interface development efforts across the various (analysis, software and real-time) testbeds within a project, and allow easy migration of models between testbeds; to allow the easy support of a variety of S/C configurations and models and simulation environments for all the phases of the mission; to permit the easy reuse and customization of hardware models across various missions.

The DARTS dynamics compute engine [5] implements a fast and efficient spatial algebra recursive algorithm [6, 7] for solving the dynamics of multi-body tree-topology, flexible spacecraft systems. Actuators are models that can impart a force on a body, such as a thruster. Sensors provide data, such as a star tracker or gyroscope. Motors are used to articulate bodies which are joined by various kinds of hinges (such as a pin, U-joint, gimbal, and others).

DSHELL's library of reusable hardware models includes sensor and actuator devices such as gyroscopes, thrusters, star-scanners, etc., with standardized D-function interfaces to DARTS and the external simulation environment. The plug and play simulation can be easily configured and interfaced to flight software for algorithm development, as well as for test and integration. The object-oriented model library includes extensive instrumentation for giving a user the high visibility into the simulation necessary for effective use as a design, development and test tool. DSHELL is in use by several of NASA's inter-planetary deep space missions including Galileo, Cassini, Mars Pathfinder and New Millennium Deep Space I.

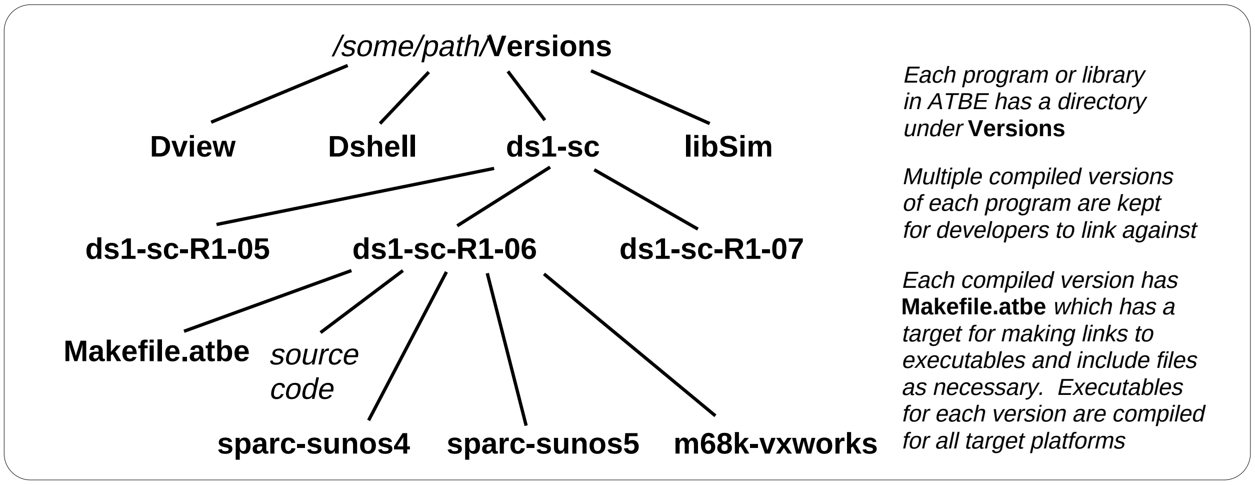
Data for DSHELL models consists of parameters, discrete states, continuous states, commands, and outputs. Parameters are values that are set while reading a configuration script upon startup, but are not changeable by the model itself. Discrete states are initialized at startup, and may be modified by the model and the user during run time (like the states described in the *ControlShell* models described from the previous section). The text interface allows general nested data structures as data types. Continuous states are updated by the numerical integrator in DARTS, which requires the model builder to provide a method for computing the derivatives of these states. Commands are time tagged data structures sent by flight software, and outputs are time tagged data structures sent to flight software. There are various methods available for a DSHELL model to define its functional interface. The most important of these methods are those updating discrete states and for calculating derivatives of continuous states.

A DSHELL *model file* is read at run-time, and specifies the bodies that make up the spacecraft as well as their masses and the types of hinges that join them together. Then, actuators and sensors are specified, with the bodies they are on and their locations on those bodies. Configuration changes can be made by editing start up files like this one, without recompiling any code. DSHELL also has a *Tcl* interface, which can be used to get information about the simulation and the models therein.

5 The YAM Configuration Management tool

ATBE source code control is carried out using *CVS* which is a third party tool based on *RCS*. It allows entire directories to be recursively checked in and out from a central "repository", and permits multiple developers to work on the same code simultaneously. It also has facilities for making new branches of the source tree, and merging these branches back onto the main trunk.

The ATBE subsystem is composed of several tools, both libraries and executables, each of which are referred to as a **module**. The size and build time for the source code makes it inconvenient for each developer to check out his/her own copy of the entire ATBE subsystem software. Considering that a typical developer generally works on only one or two modules at a time, a more flexible way to develop and use the ATBE software has been developed. This configuration management system collectively referred to as *Yet Another Make (*YAM*)* system consists of a layer of *Perl* scripts on top of *CVS*. The objective is to allow developers to choose which ATBE modules they wish to checkout for development purposes and which ones they simply need available. T A versions area maintains released versions of each module in the directory structure shown in Figure 5 so that developers have all module versions available for checkout or linking.



/some/path/Versions
Dview
Dshell
ds1-sc
libSim
ds1-sc-R1-05
ds1-sc-R1-06
ds1-sc-R1-07
Makefile.atbe
source code
sparc-sunos4
sparc-sunos5
m68k-vxworks
Each program or library in ATBE has a directory under Versions
Multiple compiled versions of each program are kept for developers to link against
Each compiled version has Makefile.atbe which has a target for making links to executables and include files as necessary. Executables for each version are compiled for all target platforms

Figure 5: Multiple compiled versions of each program are maintained

When a developer wants to make changes to the code of one or more modules, he/she must first run a “setup” script. This script will create an ATBE root directory for the developer, as shown in figure 6. The developer then specifies which modules are to be actually checked out from *CVS* (known as **work module**s) and which are to be symbolically linked to a pre-compiled version (known as **link module**s). This information is kept in a configuration file that allows the developer to tell at a glance which modules they have checked out, and which versions of link modules they are using. The developer’s *$PATH* environment variable and Makefiles are kept clean because links to all ATBE executables exist in a single **bin** directory, and links to all ATBE header files and libraries are kept in single **include** and **lib** directories. Multiple copies of each module are kept around so developers get new versions of link modules only when they are ready.

Each module has a **Makefile.atbe** file which has a **mklinks** target that will make symbolic links to the binaries and header files for the module. The **Makefile.atbe** file defines a clean interface between the YAM build procedure and the module, allowing the easy addition of externally developed programs and libraries group as ATBE modules. The creation of a **Makefile.atbe** for the module is all that is required to plug into the YAM build process and no modifications of the code or the modules’ Makefiles are required.

A **Release** directory contains an `ATBE_ROOT` directory for each release of the entire subsystem. All modules in these releases are specified as link modules, and make it clear which versions

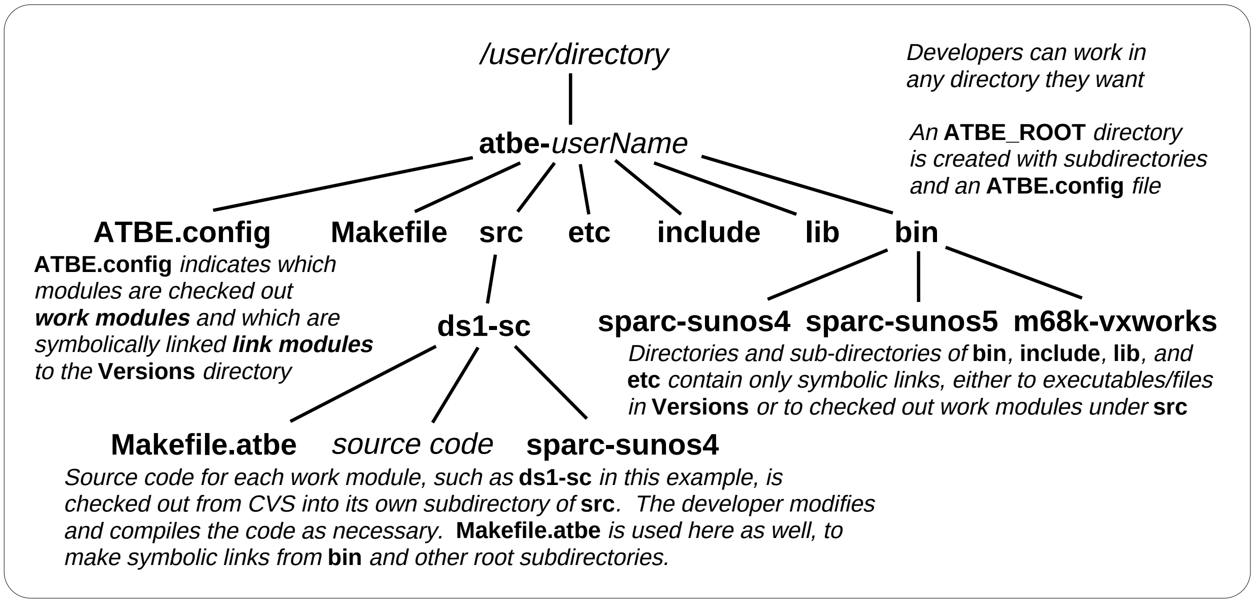

/user/directory
Developers can work in any directory they want
atbe-userName
An ATBE_ROOT directory is created with subdirectories and an ATBE.config file
ATBE.config
Makefile
src
etc
include
lib
bin
ATBE.config indicates which modules are checked out work modules and which are symbolically linked link modules to the Versions directory
ds1-sc
sparc-sunos4
sparc-sunos5
m68k-vxworks
Directories and sub-directories of bin, include, lib, and etc contain only symbolic links, either to executables/files in Versions or to checked out work modules under src
Makefile.atbe
source code
sparc-sunos4
Source code for each work module, such as ds1-sc in this example, is checked out from CVS into its own subdirectory of src. The developer modifies and compiles the code as necessary. Makefile.atbe is used here as well, to make symbolic links from bin and other root subdirectories.

Figure 6: Developers can select which modules they want their own copies of

of the modules work together. This also provides a place to go to simply run a version of the subsystem. *CVS* tags are used to tag source code when new versions of a module come out and when releases are made, so releases are tracked by *CVS* as well.

6 Other Tools

In addition to the third party tools already mentioned, *ControlShell*, *Tcl/Tk*, and *CVS/RCS*, there are several other tools used in the simulator and testbed environment. A brief description of these tools is given below.

Console: The *Console* is an ATBE program for launching, running, monitoring, and gracefully shutting down multiple processes. For example, it can start up a version of ACS flight software, 1553 bus manager flight software, and the ATBE simulator. It provides a *Tcl* command line, and can send *Tcl* commands to any one of these processes, or to all processes. It provides a support library for the receiving processes (like ACS) to handle *Tcl* commands and send back results. The console also has provisions for a clock, which drives all processes. So all processes can be started or stopped at the same time, and take time steps of the same size (simulation time) and frequency (wall clock time). The console is highly configurable, so the user can specify which subsystems and tools they wish to use in a startup script, so no code needs to be re-compiled. Additionally, it can distribute these processes to multiple hosts/platforms by using *rsh*, and display windows and GUIs on multiple monitors. Communication between the console and the processes may take place using a number of different protocols.

Dview: *Dview* is a 3D spacecraft renderer developed for use with DSHELL. It can run on several different platforms including Silicon Graphics and Sun (using the public domain graphics library *MESA*). *Dview* reads an input file similar to the DSHELL model file, which specifies the same bodies that make up the spacecraft. Additionally, it knows about the geometry and color to render these bodies in. So the same program can be, and is, used for different spacecraft without changing any code. During run time, DSHELL sends messages to *Dview* indicating the position and attitude of the spacecraft. Messages that articulate bodies on the spacecraft may also be sent, as well as "thruster-fired" messages which display a plume from the specified thruster. As with the console, a number of different communications protocols are available. On Silicon Graphics machines, there are options for doing fancier rendering with lighting and texture maps.

libState: The text interface to state data used by Tcl commands in DSHELL and the DS1 simulator is implemented using a library known as *libState*. This library defines C++ template classes which keep track of a reference to a user variable and a text interface to that variable. This allows a C/C++ program to access its variables as usual, while still having simple text "peek" and (optional) "poke" access to the data through standard text string get/set methods of the *libState* base class. A single data type may have more than one kind of interface available for it. For example, an integer may have both a numeric interface as well as an enumerated keyword/value interface. Or, a double may have a standard interface, and one in which some kind of units are expected as well, automatically converting the number into internal units for computation. Since C++ templates are used, everything is done in a completely type-safe manner. New interfaces may be added to existing or new data types by defining parse and print methods. However, special support for possibly nested data structures and both fixed and variable length arrays is provided for convenience. Data structures can implemented as a compound group of sub-states, allowing access to either individual fields of the structure or the entire structure at once, without the need for specialized parse/print methods. Arrays also allow access to either individual elements or the entire array. Future work may involve adding an automatic XDR interface, so rpcgen and special code are not needed to save these variables in a binary file or to send them over a network in binary. *libState* works on both Unix and VxWorks operating systems.

Stethoscope: *Stethoscope* is a real-time plotting tool from Real-Time Innovations, Inc. It can plot variables from VxWorks tasks as well as Unix processes. Variables and *ControlShell* signals can be "installed" to *Stethoscope* at run time, and multiple Stethoscopes can be run on the same target at the same time without interference. On VxWorks, stethoscope can look at the global memory directly so while running, a task does not have to tell *Stethoscope* about updates to variables. *Stethoscope* runs as a low-priority task to minimize impact to other tasks.

NDDS: *Network Data Delivery Service (NDDS)* is a fast, reliable, message passing tool also from Real-Time Innovations, Inc., with a very general API. It runs on both VxWorks and Unix platforms and can pass messages between processes running on different hosts. Each host runs an *NDDS* daemon giving it a domain number and list of "peer" hosts. By having domain numbers, multiple *NDDS* daemons may run on the same hosts simultaneously without interfering with each other. Programs register themselves as "producers" and/or "consumers" of messages. The same program can both produce and consume multiple kinds of messages, and there may be multiple consumers of the same message. Consumers may be either "polled" or "immediate". Polled consumers execute a callback for an incoming message only when a poll function is called (so the program has control over when the message is handled). Immediate consumers execute their callback as soon as a message arrives and no poll function is needed. There are many other options for dealing with real-time issues available as well. Data is passed using an XDR-like mechanism, and new message data types can be added using an rpcgen-like program.

IPC: *IPC* is the messaging system [8] used by DS1 flight software to pass messages between its own tasks, and has capabilities similar to those of *NDDS*. ATBE accepts *IPC* messages for data from a 1553 bus model.

7 Conclusion

An adaptable spacecraft simulation testbed is essential for the design, development, testing and integration of autonomy flight software and hardware. The testbed needs to support the development and test simulations which span a wide range of engineering platforms; functional and fidelity models; test scenarios; and durations. This paper describes the reconfigurable ATBE simulation environment which supports the end-to-end development, integration and test needs for the autonomy flight software for the New Millennium Deep Space I project. A significant fraction of the effort to date has been spent on the design of ATBE's architecture so that it is flexible and adaptable to meet the needs of the autonomy flight software development. During the coming weeks the ATBE effort will transition to support the large influx of new models into the spacecraft simulation environment, and support the development and implementation of real-time hardware-in-the-loop simulations.

8 Acknowledgements

The research described in this paper was performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

References

[1] L. Fesq, A. Aljabri, C. Anderson, R. Connerton, R. Doyle, M. Hoffman, and G. Man, "Spacecraft Autonomy in the New Millennium," in *19th Annual AAS Guidance and Control Conference*, (Breckenridge, CO), Feb. 1996. Paper AAS-96-001.

[2] M. Rayman and D. Lehman, "NASA's First New Millennium Deep-Space Technology Validation Flight," in *Second IAA International Conference on Low-Cost Planetary Missions*, (Laurel, MD), Apr. 1996. Paper IAA-1-0302.

[3] S. Lisman, D. Chang, and F. Hadaegh, "Autonomous Guidance and Control for the New Millennium DS-1 Spacecraft," in *AIAA Guidance, Navigation and Control Conference*, (San Diego, CA), June 1996. Paper 96-3817.

[4] E. Riedel, S. Bhaskaran, S. S., W. Mollman, and G. Null, "An Autonomous Optical Navigation and Control System for Interplanetary Missions," in *Second IAA International Conference on Low-Cost Planetary Missions*, (Laurel, MD), Apr. 1996. Paper IAA-L-0506.

[5] A. Jain and G. Man, “Real–Time Simulation of the Cassini Spacecraft Using DARTS: Functional Capabilities and the Spatial Algebra Algorithm,” in *5th Annual Conference on Aerospace Computational Control*, Aug. 1992.

[6] G. Rodriguez, K. Kreutz-Delgado, and A. Jain, “A Spatial Operator Algebra for Manipulator Modeling and Control,” *The International Journal of Robotics Research*, vol. 10, pp. 371–381, Aug. 1991.

[7] A. Jain, “Unified Formulation of Dynamics for Serial Rigid Multibody Systems,” *Journal of Guidance, Control and Dynamics*, vol. 14, pp. 531–542, May–June 1991.

[8] R. Simmons, “Structured Control for Autonomous Robots,” *IEEE Transactions on Robotics and Automation*, Feb. 1994.