Advanced Simulation Environment for Autonomous Spacecraft

Jeffrey J. Biesiadecki **Abhinandan Jain** **Mark L. James**
Jet Propulsion Laboratory/California Institute of Technology
4800 Oak Grove Drive M/S 198-235, Pasadena, CA 91109 USA

ABSTRACT

NASA is developing technology to increase spacecraft on-board autonomy, in an effort to reduce overall mission cost and mission operations resources. Achievement of this objective requires the development of a new class of ground-based autonomy testbeds that can enable rapid development, test, and integration of the new autonomous spacecraft flight software. This paper describes the development of the Autonomy Testbed Environment (ATBE), designed to address these needs.

1 INTRODUCTION

The Autonomy Testbed Environment (ATBE) supports spacecraft simulation over a wide range of engineering platforms, functional and fidelity models, fault injection, test scenarios and duration. Conventionally, such breadth of 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 to meet the development and test needs of many different kinds of users. ATBE's design enables easier maintainability and usability, and perhaps most significantly, continual evolutionary changes in model requirements, functionality, and fidelity. Additionally, ATBE provides a high degree of visibility into model state variables, extendable interfaces to data monitoring and plotting tools, and simulation checkpointing.

The ATBE toolkit includes libSim, which uses a data flow paradigm for connecting higher-level device and subsystem models, and provides special features for modeling faults. (Examples of libSim models include bus interfaces, device electronics, and valves.) It also includes Dshell, a high fidelity real-time dynamics simulation package with models for the various actuators and sensors on a spacecraft. This paper describes these tools in detail in sections 2 and 3. See reference [1] for a broad overview of ATBE.

ATBE models are roughly categorized as real-time, containing functions that are executed every tick of the simulation, and non-real-time, which do work in response to events or commands. Currently all real-time models execute in the same thread. Event-driven models run as separate processes and typically communicate via messages. libSim and Dshell models are real-time models. An example of an event-driven model is a scene generator which is used to simulate an 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. Models are implemented as non-real-time due to the nature of the device they simulate, or to ensure that critical real-time performance requirements of the simulator are met. Figure 1 shows an example of the kinds of models that are included in an ATBE simulation.

The data flow simulator libSim is the highest layer of the ATBE simulator. It contains a model that wraps Darts/Dshell (figure 2), issuing commands to Dshell hardware models based on its inputs and setting its outputs based on those received from Dshell models. Event-driven models each have corresponding simple libSim models that send and receive messages to and from the non-real-time processes, to incorporate the data from these processes into the real-time core.

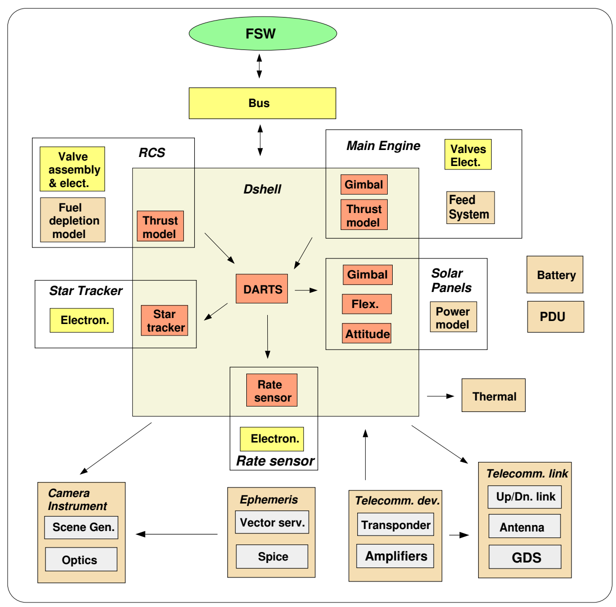

Figure 1: Representative types of models in an ATBE S/C simulation

2 libSim DATA FLOW SIMULATOR

libSim is a library to facilitate the development of spacecraft subsystem and hardware models for a simulator using a data flow paradigm [2]. It formalizes what constitutes a model, and provides frameworks for both independent model development and the connection of multiple models in a simulator. It is implemented in C and C++, has a C functional interface, and runs on both Unix and VxWorks platforms.

The free software package *Tool Command Language (Tcl)* [3] is used for the command line and script interface.

A libSim model consists of state variables, inputs and outputs, an **init** function, and a **tick** function. The init function calls libSim functions to register the model's state/input/output variables and tick function. The tick function is called repeatedly during the simulation to set the values of outputs based on the values of inputs and internal state variables.

When running in a multiple model simulator, the outputs of one model may be hooked up to the inputs other models (figure 2). libSim provides functionality for specifying these connections and facilities for advancing simulation time and calling model tick functions. Each input and output can be connected to only one **signal**; but there is no limit to the number of inputs and outputs that can be attached to a signal. Signals are shared buffers, and inputs and outputs are pointers to these buffers. So when a model writes to its output, it is directly writing to the inputs of any connected models without overhead due to copying or message passing. Inputs and outputs are also time stamped, so it is possible to determine and specify when this data is "fresh". The order in which model tick functions are called is determined from the dependencies implied by the data flow. If desired, libSim models can be wrapped and connected using other packages such as Real-Time Innovations, Inc.'s software *ControlShell* or Matlab's *Simulink*.

When running a stand-alone unit test for a particular model, the developer can set the value of the inputs, take a step, and look at the values of the outputs. This can be done at the command line, or in a script for automated/batch testing. The commands giving visibility into the models are the same as for the full-up simulator. In this mode, the user writes a trivial *main()* function and links to the libSim library to get an executable.

libSim is reconfigurable in the sense that model instantiations and connections are specified in an input file that is read at run time, and may be modified without recompiling any code. Additionally, models may be deactivated (meaning their tick functions will not be called) and reactivated during run time. This allows alternate implementations for a device, perhaps one being an interface to actual hardware-in-the-loop and another being a pure software simulation. It also facilitates debugging.

Models can register state variables with libSim. By doing so, these variables will automatically have a command line interface at run-time, allowing the user to look at and modify their values. This provides a standard interface to the model and simplifies debugging. State variables can be checkpointed, to set the initial conditions for a future run. Types allowed for state variables are any basic C data type, arrays, and C enumerations.

libSim provides special support for modeling faults, intended to help reduce coding for implementing simple fault states. The built-in faults are a specialized form of integer enumeration states. All fault variables may be in a "nominal" mode which is mapped to an integer value of zero. The model developer adds other keywords to a fault variable that map to other, non-zero, values. Each one of these values should correspond to a mutually exclusive fault condition. For example, a valve may have a fault state that could be set to "nominal", "stuckClosed", or "stuckOpen".

Faults may be triggered within the model's tick function if the model determines that some criteria is violated. More commonly, however, faults are injected by using a *poke* command or GUI at run-time by the user. Faults may also be fixed (i.e., set to "nominal") by the model in its tick function, as well as through a run-time *poke* command. Special support is given for the automatic correction of faults to help reduce repetitive coding in models. Automatic fault correction can occur if a time out expires for the fault, if the model receives a soft reset, and if the model is power cycled. All of these capabilities are optional, and may be controlled at the command line.

Other features of libSim include a scheduler to perform tasks at either a specific time or every simulation step, logging routines with verbosity selectable on a per model basis, a global database to associate names with pointers and organize global variables, and an extensive set of commands available to make inquiries about the simulation and models. These commands are useful for writing scripts and graphical user interfaces, debugging, and monitoring.

3 Dshell 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 integrates the Darts flexible multi-body dynamics computational engine and libraries 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 spacecraft simulation needs.

The main goals of the Dshell environment are: to significantly reduce the software development 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 testbeds (analysis, software and real-time) 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; and to permit the easy reuse and customization of hardware models across various missions.

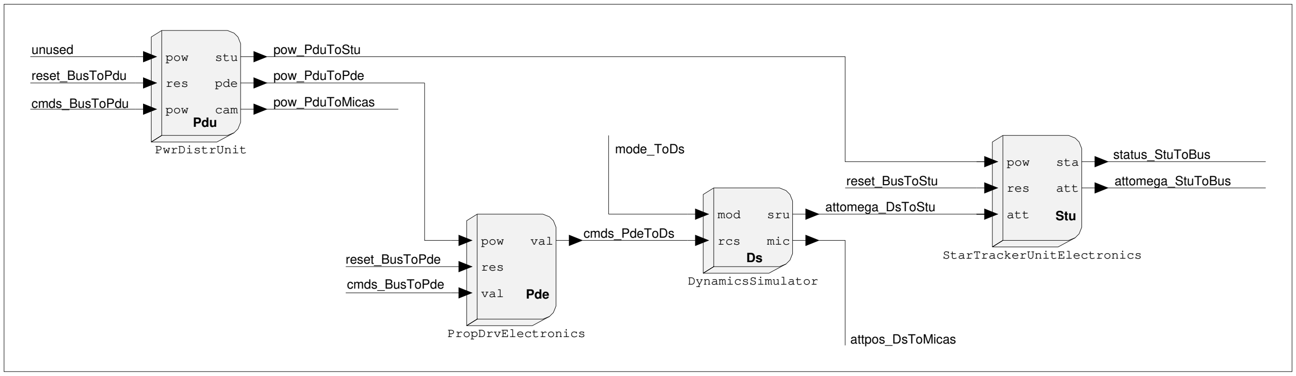

Figure 2: Example of libSim models viewed with *ControlShell*'s Data Flow Editor

Dshell is a library implemented in C++ may be embedded in another simulator as described in section 1. Or, a small "main()" routine can be written to send data between flight software and Dshell models, and advance simulation time. For model development, a generic "open-loop" version of main() is available in which the user controls time and data to and from models. This is invaluable for writing batch scripts to do regression testing.

Simulation time is tracked from the start of simulation. An **I/O step** consists of an integer number of **integration steps**. Darts dynamics are computed each integration step. Input and output to and from Dshell models is expected to occur each I/O step.

3.1 Darts – Dynamics Algorithms for Real-Time Simulation

The Darts dynamics compute engine [4] implements a fast and efficient spatial algebra recursive algorithm [5,6] for solving the dynamics of flexible, multi-body, tree-topology systems. It is very general, and is also in use for non-spacecraft applications such as molecular dynamics [7]. Darts is a library implemented in ANSI C available for Unix and VxWorks platforms.

An analyst provides a text input file that is read at run time and specifies the **bodies** that make up the spacecraft: their masses, inertial and flexibility properties, as well as the types of **hinges** that bind them together. A hinge connects two bodies, and there are many types available (such as pin, U-joint, gimbal, translational, and others). Bodies may be connected in a tree topology, with each body having a single parent body, and the root of the tree being referred to as the **base body**. The locations of named **nodes** where forces may be applied or dynamics properties should be computed are also specified in a Darts input file. Because the above data is not hard-coded, dynamics models can be easily constructed for different missions, and models can be changed without necessitating the recompilation of source code.

3.2 Dshell Model Classes

Dshell provides C++ base classes for hardware device models. **Actuators** can impart a force on a node of a body, such as a thruster. **Sensors** are attached to a node of a body and make use of dynamics calculations produced by Darts for that node. Examples of sensor models include star trackers and gyroscopes. **Motors** are attached to hinges and are used to articulate the bodies that the hinge connects. **Encoders** are also attached to hinges, and are to motors what sensors are to actuators. Dshell device models are massless, and other than applying a force or articulating a body, do not affect the dynamics of the spacecraft. All four of these classes are derived from a common base class (**Model**), which defines data and methods associated with each model.

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 both the model and the user during run time. **Continuous states** are updated by the numerical integrator in Darts, and require 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. Parameters, discrete states, commands and outputs may be of any basic C data type (such as *int* or *double*), C enumeration, structure, or fixed-size array. Structures may be nested, may contain arrays, arrays of structures are permitted, and so on. Continuous states are either *double* or arrays of *double*.

There are various methods available for a Dshell model to define its behavior. **Pre-** and **post- I/O step** methods are called at the beginning and end of an I/O step, and are typically used for models to retrieve commands from and send data to flight software, respectively. **Pre-** and **post- integration step** methods are called at the beginning and end of an integration step, and are typically used to compute discrete states. Each integration step, an integrator calls a function to compute the time derivative of the Darts state vector. This function also calls **pre-** and **post- derivative** methods for each Dshell model immediately before and after computation of Darts derivatives. The pre-derivative method is typically used for actuators to apply forces to the nodes they are attached to. The post-derivative method is typically used to compute the time derivative of any continuous states the model may have. The number of times these derivative methods are actually called per integration step depends on the numerical integration algorithm selected. Note that unlike libSim, Dshell models do not interact with each other directly, so the relative order in which their methods are executed does not matter (figure 3).

The base classes provide several methods useful to a model, including methods to get the simulation time, step sizes, and Darts information. These would be called from the model's pre/post I/O step and other methods described in the previous paragraph.

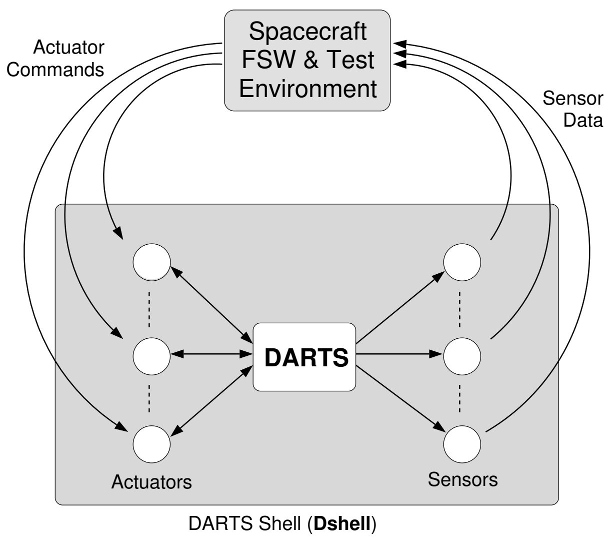

Figure 3: Typical data flow for a Dshell simulation

3.3 Dshell Model Libraries

Classes for actual device models are derived from any of the four base classes described in section 3.2. The code for model classes may be grouped into reusable libraries, organized perhaps by mission, by vendor, or by type of device. There are several models available for thrusters, gyroscopes, star scanners, accelerometers, and other devices used on JPL spacecraft. They can be used as-is for quick prototype simulations, or as a starting point for similar models on a new spacecraft.

An automatic code generator is available to simplify model development. The model developer writes a text file that describes the model, listing the types, names, and descriptions of the parameters, states, commands, and outputs associated with the model. A prototype graphical user interface is available for generating this file. The code generator takes this file as input, and generates a C++ header file and stub source file for the model class. The developer then fills in methods (pre/post I/O step and the rest) as needed to define the model's behavior. Very little knowledge of C++ is needed, but it is useful to be familiar with C.

The automatic code generator also makes an **interface class**, specific to the model class the developer is defining (figure 4). The developer never changes this code and does not need to even look at it. This class provides model-specific functions to issue commands and retrieve outputs from a model, code commonly needed to define a text interface to the model's data, and other methods needed by Dshell. The command and output functions would typically be called from the simulator or main() routine that calls other Dshell routines. They are model-specific to keep them type-safe (avoiding the use of **void *** pointers reduces the occurrence of some programming errors). This also allows a simpler interface for commands and outputs of basic types, and is faster than performing any kind of marshalling or conversion of structures. The code generated for the interface class is meant to eliminate tedious coding by a developer that is typically needed for a model. It is generated in a class separate from the actual model class to clearly delineate code the developer should modify. This helps keep the code for the stub model class small.

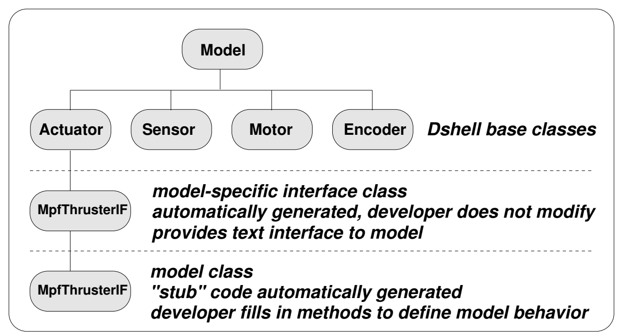

Figure 4: Dshell class hierarchy

3.4 Dshell Run Time Environment

The input file containing Darts information may also contain statements to instantiate models, specifying the model class and instance name. States and parameters for a model may be initialized here as well. Again, not hard-coding this information makes it easier to change configurations without recompiling code.

Like libSim, Dshell also has an extensive set of *Tcl* commands which can be used to get information about the simulation and models therein. In particular, the values of model states and parameters can be peeked and poked from the command line, commands to models can be issued as if they came from flight software, and outputs from models can be examined. There are enough commands available to query which models are instantiated and the data types and descriptions of model states that a graphical user interface to display state data can dynamically create itself, so a programmer does not need to change GUI code if the simulation configuration changes or new models are added. A prototype of such a GUI has been implemented using *Tk*.

DARTS and DSHELL model state variables can **checkpoint**ed to a text file containing "poke" commands. This file can be edited by the user if necessary without needing to know any syntax other than the already familiar *Tcl* commands. On a subsequent run, this file can be used to initialize states and **resume** a previous run.

DSHELL can also keep track of multiple S/C dynamics models. Alternate dynamics models of the same spacecraft can be selected from (such as in-cruise versus in-orbit models with different fuel slosh behavior, or pre- versus post- probe release). Only one such alternate dynamics model may be active at any given time, and DSHELL device models implicitly interface only to the active model. Or, multiple spacecraft can be bookkept, as in the New Millennium Program's Deep Space Flight 3 formation flying mission. Any combination of alternate models of multiple spacecraft is allowed.

As with LIBSIM and DARTS models, DSHELL models can be deactivated from the *Tcl* command line or startup file. This is useful for debugging, or if there are alternate models for the same spacecraft device (perhaps one would interface to actual hardware-in-the-loop).

It is also possible to schedule C functions and *Tcl* scripts at run time for either one-time or repeated execution. This is very handy for debugging and monitoring variables. It is also useful for interfacing DSHELL to other tools. Such interfaces have been created to Real-Time Innovation, Inc.'s data monitoring tool *StethoScope* and to JPL's 3D viewer *Dview*. Interfaces to other tools can be created in a similar manner, without having to change DSHELL code. Aside from keeping DSHELL code smaller and cleaner, it makes it easy to mix and match interfaces among testbeds which use different monitoring tools.

4 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 simulations with a wide range of capabilities. This paper describes the reconfigurable ATBE simulation environment and tools that comprise it.

Both LIBSIM and DSHELL use the same core code for providing a text interface to their models, which is why the capabilities for giving visibility into the model *data* are similar. The main differences between the tools are in the *methods* associated with models, and when they are called. LIBSIM provides a mechanism for models to share data with each other, while DSHELL provides methods to interface with the dynamics integrator at the appropriate times.

Many models needed for simulation neither provide nor require dynamics information, but do depend on and should affect the states of other models. Adding inter-model communication and sorting to DSHELL was considered, but the data flow becomes messy because DSHELL models have more than one method executed per time step. A model could conceivably try to use an input value in its pre I/O step method, for instance, but the upstream model producing the value not do so until its post I/O step method is called. There did not appear to be an easy way to specify or detect such a case. Additionally it is not clear when to do the dynamics integration in a time step. So LIBSIM was designed to allow a model one "tick" function per time step.

The tools can be used together when a developer writes a LIBSIM model that wraps DSHELL. The "tick" function of this wrapper model calls DSHELL functions to issue commands to DSHELL models, advance DSHELL's notion of time, and retrieve outputs from DSHELL models. For example, the "DynamicsSimulator" components shown in figure 2 is such a wrapper. The "PropDrvElectronics" model receives thruster commands from flight software via a bus, but if it is powered off by the "PwrDistrUnit" or is in a fault state, it will not pass these commands on to "DynamicsSimulator" which contains thruster models that do the actual work of applying forces. Experience with New Millennium Deep Space 1 has shown that moving fault injection and other functionality that had traditionally been in DSHELL models to LIBSIM has kept the DSHELL models simpler and focussed on the tasks for which they are intended.

Future work will include further blending of these tools, in part by implementing an automatic DSHELL wrapper model, so a developer does not need to write the code described in the previous paragraph. Additionally, we are looking into how ATBE's architecture and toolkit can be enhanced to provide thermal, power, and fuel consumption modeling. The ability to use these tools either independently or in various combinations with one another has been valuable and will be retained.

ATBE tools have been used on many JPL flight projects, and are continually evolving based on experience with these projects. DARTS is used by the Cassini project development, test, and integration teams. DSHELL is used by Galileo, Mars Pathfinder, and in the Flight System Testbed on many new projects including Stardust. LIBSIM is in use by the New Millennium Program's Deep Space Flight 1. Development of Cassini High Speed Simulator is nearing completion and will be used during Cassini mission operations to test command sequences prior to uplink; this simulator uses ATBE tools as well.

5 ACKNOWLEDGEMENTS

The authors would like to express their thanks for the work performed by the other ATBE team members: David Breda, Kirk Fleming, Martin Gilbert, David Henriquez, Linh Phan, Ling Su, and Matt Wette.

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] J. Biesiadecki, A. Jain, "A Reconfigurable Testbed Environment for Spacecraft Autonomy," in *Simulators for European Space Programmes, 4th Workshop*, (Noordwijk, The Netherlands), Oct. 1996.

[2] J. Biesiadecki, "libSim Simulator Model Development Library," Jet Propulsion Laboratory, internal document, (Pasadena, CA), Nov. 1996.

[3] J. Ousterhout, "Tcl and the Tk Toolkit," *Addison-Wesley Publishing Company*, 1994.

[4] 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.

[5] 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.

[6] 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.

[7] A. Jain, N. Vaidehi, G. Rodriguez, "A Fast Recursive Algorithm for Molecular Dynamics Simulation," *Journal of Computational Physics*, vol. 106, no. 2, pp. 258–268, June 1993.