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 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 1 comet/asteroid rendezvous mission. This simulation testbed has been designed to enable rapid design of flight modules, easy 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 to this objective will require the development of a new class of ground-based autonomy testbeds that can enable low cost and rapid design, test, and integration of the radically new autonomous system flight software. This paper describes the development of such a new class of ground-based autonomy testbed — the Autonomy Testbed Environment (ATBE) — that enable the low cost and rapid design, test, and integration of
2 Architecture of the VRTI Environment

The VRTI is a software environment designed to support the development of virtual reality applications. It provides a platform for developers to create immersive interactive environments. The VRTI architecture is composed of several key components:

- **The VRTI Core:** This is the central component of the environment, responsible for managing interactions between applications and the virtual world. It provides a high-level interface for developers to create and control the virtual environment.

- **Virtual Reality Interface:** The VRTI supports various virtual reality hardware, such as head-mounted displays, motion tracking systems, and joysticks, enabling developers to create applications that utilize these technologies.

- **Application Framework:** Developers can use this framework to create applications that run within the VRTI environment. It provides tools and libraries for creating interactive, multi-user applications.

- **Scripting Language:** The VRTI supports a scripting language that allows developers to write scripts for controlling the virtual environment. This can be used for creating interactive scenarios or automating certain tasks.

- **Documentation and Support:** The VRTI comes with comprehensive documentation and support resources to help developers get started and troubleshoot issues.

The VRTI environment is designed to be flexible and scalable, allowing developers to create applications that meet a wide range of needs in different domains.
Software development: The feedback loop of the software development process.

On the other hand, the dynamic software development model is inherently tied to the ability to make changes in the software development process. On the one hand, the dynamic software development model supports significant changes on the software development process to improve the software development process. On the other hand, the dynamic software development model supports significant changes on the software development process to improve the software development process. On the other hand, the dynamic software development model supports significant changes on the software development process to improve the software development process.
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 bulk of the ATBE architecture and focus the ATBE software development in knitting together these tools and implementing new features into a usable flight software development and test environment. The architecture is also being designed for reuse in missions that follow 1 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 lip-) level categorization and decomposition of Sils-systllls into real-time, DSHELL and event-driven models is illustrated in Figure 1. The real-time models or tools are implemented and interconnected 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 3: D51 simulator and assorted tools
outputs. Additionally, simulation time can be advanced, power to the device turned 011 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 'Tel' scripts to enable limit testing. While these capabilities are available in the full-up simulator, the mini environment has the advantages that it involves less code, is self-controllable, and is faster and easier to run.

3 Simulation Models in the ATB8 Simulator

The following is a brief summary of the various models in the ATB8 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 (1×1 DE), fuel tank (∫(s±l[efc]$x-late(collslllll}$ tic)ll 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**: A 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**: A 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.

**Telecom:** 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.

### 4.1 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

![Diagram of Dshell architecture with Darts and device models](image)

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
The VERTEX configuration management tool

A VERTEX model is read in at run-time and specifies the nodes that make up the structural, logical, and physical nodes of the VERTEX configuration management tool. It allows for the creation of configuration models and manages changes to these models. The tool provides a comprehensive set of features for managing configuration models, including:

- Model creation and management
- Configuration change tracking
- Configuration change approval
- Configuration change implementation
- Configuration change rollbacks
- Configuration change reporting

The VERTEX configuration management tool is designed to help organizations manage their configuration models efficiently and effectively. It supports the entire configuration management process, from model creation to change implementation, ensuring that configuration changes are managed in a controlled and systematic manner.
of which are referred to as a module. The size and build time for the source code makes it in inconvenient for each developer to check out his/her own copy of the entire ATBIE subsystem software. Considering that a typical developer generally works on one or two modules at a time, a more flexible way is to develop and use the ATBIE software has been developed. This configuration management system collectively referred to as YAM system consists of a layer of Perl scripts on top of CVS. The objective is to allow developers to choose which ATBIE modules they wish to checkout for development purposes and which ones they simply need available. A versions area maintains released versions of each module in the directory structure shown in Figure 5 so that developers have module versions available for checkout or linking.

![Diagram](//some/path/Versions)

Each program or library in ATBIE has a directory under Versions

Multiple compiled versions of each program are kept for developers to link against

Each compiled version 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 ATBIE 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 modules) and which are to be symbolically linked to a pre-compiled version (known as link modules). 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 ATBIE executables exist in a single bin directory, and links to all ATBIE 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 make target that will makesymbolic 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 as ATBIE 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 ATBIE.ROOT directory for each release of the entire ATBIE system. All modules in these releases are specified as link modules, and make it clear which versions
6 Other Tools

In addition to the third party tools already mentioned, ControlShell, Tel/Tran (CVS/RCS), there are several other tools used in the simulator and testing 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 Tel command line, and can send Tel commands to anyone of these processes, or to all processes. It provides a support library for the receiving processes (like ACS) to handle Tel 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 start up script, so that code needs to be re-compiled. Additionally, it can distribute these processes to multiple hosts/platforms by using rsh, and display windows and dialogue multiple monitors. Communication between the console and the processes may take place using a number of different protocols.

Dview: Dview is a 3D spacecraft rendering tool developed for use with 1D-SHELL. 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 1 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 1 Dshell, and is used for different spacecraft without changing any code. During run time, 1 Dshell sends messages to Dview indicating the position and attitude of the spacecraft. Messages that articulate bodies on the spacecraft may also 1 Dshell 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 Telcommands in 1 Dshell and the Dshell simulator is implemented using a library known as libState. This library defines C/C++ template classes which keep track of a reference to a user variable and 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 gd/set methods of the libState base class. Single data type may be used in 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 be 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 element or the entire array. Future work may involve adding an automatic XDR interface, so regular and special code are not needed to saw! 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 call Dshell “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 host simultaneously without interfering with each other. Programs register themselves as “10[11:0]:1s” 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 “poll” 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.
REFERENCES


