Multi-Mission Simulation and Visualization for Real-time Telemetry Display, Playback and EDL Event Reconstruction

M.I. Pomerantz, C. Lim, S. Myint, G. Woodward, J. Balaram, C. Kuo
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, 91109

The Jet Propulsion Laboratory's Entry, Descent and Landing (EDL) Reconstruction Task has developed a software system that provides mission operations personnel and analysts with a real time telemetry-based live display, playback and post-EDL reconstruction capability that leverages the existing high-fidelity, physics-based simulation framework and modern game engine-derived 3D visualization system developed in the JPL Dynamics and Real Time Simulation (DARTS) Lab. Developed as a multi-mission solution, the EDL Telemetry Visualization (ETV) system has been used for a variety of projects including NASA's Mars Science Laboratory (MSL), NASA'S Low Density Supersonic Decelerator (LDSD) and JPL's MoonRise Lunar sample return proposal.

I. Introduction

The JPL EDL Reconstruction Task has developed a re-useable, multi-mission software system that combines physics-based spacecraft and environmental simulation with real-time telemetry processing and interactive 3D visualization to support a variety of NASA spacecraft mission projects and domains. Originally designed as a flexible tool to help mission engineers reconstruct actual spacecraft events that occurred during a mission's EDL phase, ETV has shown to be a valuable asset during the pre-EDL trajectory planning effort as well as during the operational EDL phase when configured to process and display spacecraft states, via acquired telemetry data, in real-time.

On landing day, the MSL[1] mission used the ETV system to help operations engineers, mission personnel and the public, better understand the state and health of the spacecraft during EDL by visualizing flight software states, vehicle position, orientation, altitude, velocity and health information obtained from the mission real-time telemetry stream, along with environmental information such as Sun, Earth, Mars orbiter positions. This spacecraft data combined with accurate high resolution digital elevation maps of Mars and the Gale Crater landing site provided the MSL engineers with a powerful and accurate visual representation of MSL's EDL phase. The LDSD[2] project is also currently using ETV to visualize physics-based simulation results of their spacecraft for a variety of test flight scenarios to help better understand predicted vehicle performance during EDL, which will eventually allow the delivery of heavier payloads than currently possible to the Martian surface.

II. System Requirements, High-Level Software Design and Use Cases

We determined very early on that the key system requirements would include system accuracy, multi-mission re-usability, re-configurability and reduced cost deployment moving from mission to mission. In addition, the system would need to run on standard workstation-class or high-end laptop computer systems with consumer-level or better graphics cards. To maintain compatibility with legacy and newer JPL simulation, engineering and analysis tools, we chose C++, Python and Fedora Linux as our development languages and operating system platform.

To help us achieve these requirement goals, the core ETV system has been built upon the ongoing and mature space vehicle software simulation framework and 3D visualization software developed in the JPL DARTS[3] Lab. For physics-
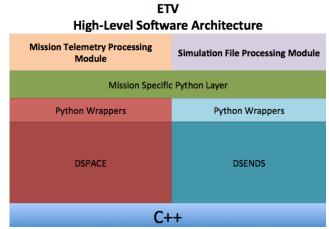

Figure 1. ETV Software Organization. *The thin, mission specific, ETV layer is built in Python and accesses DSENDS and Dspace C++ API's via SWIG-wrapped Python code*

1

based EDL simulation, the DSENDS[4] framework provides an accurate and mission-proven basis for our reconstruction capability and the Dspace[5] 3D visualization software gives us a high-performance, kinematically accurate, interactive 3D visualization capability.

Both DSENDS and Dspace are built on C++ core frameworks with SWIG[6] wrapped Python[7] front ends that allowed us to maintain ETV's thin mission-specific Python layers while maintaining system performance and reducing development time and cost. In simple terms, both the DSENDS and Dspace APIs are fully available at run time via Python scripts that control the operation of the system. In addition, because we run the system in a Python main script with a Python command line, we can inspect system state information and/or make function calls into the Python or SWIG-wrapped C++ layer.

To achieve a true multi-mission capability, ETV has been designed to be completely data driven. Spacecraft mechanical system characteristics, including mass, dimensions, center of gravity, location and type of articulated full 6 DOF joints, components and physical shape, via CAD models, can be provided to the system as Python dictionary items available to both DSENDS simulation and Dspace visualization. If desired, the system can import and execute high-fidelity models for a variety of spacecraft sub-systems such as Attitude Control System (ACS), Reaction Control System (RCS) and accurately modeled CAHVOR[7] imaging cameras models. Planetary environmental data such as gravity and atmospheric models accessed by DSENDS can be used to help mission planners predict spacecraft landing zones, while other environmental data, such as accurate sky/star maps, terrain digital elevation maps and texture imagery, can be used both for determining altitude above the planet's surface and for presenting accurate visualization to users when presenting "plan" or "context" views of a large terrain area or simulating spacecraft imaging camera views.

Figure 2. ETV for MoonRise Lunar Sample return proposal. *ETV can visualize simulated spacecraft-acquired imagery using actual camera pointing and field of view information.*

Typical system use cases include mission planning and proposal phase visualization and simulation of proposed spacecraft flight paths, as well as actual flight hardware and/or 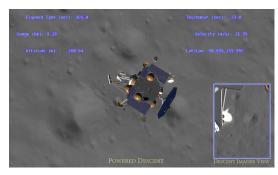software in-the-loop. JPL's MoonRise Lunar sample return proposal included flight radar hardware in-the-loop for Lunar Descent, while both Descent and Ascent mission phases were visualized using prescribed motion spacecraft data. Simulation and real time telemetry-based mission operations support and display of EDL phase mission data was performed for the MSL mission, and simulation and real-time telemetry-based flight experiment support is currently under development for the LDSD project. Extended use cases include running in closed-loop mode with flight hardware and/or flight software, with ETV providing spacecraft flight profile information, simulated imaging or sensor data hardware. Accurate line-of-sight communications modeling is supported by the underlying DSENDS simulation framework and is also supported by Dspace the visualization system.

III. System Deployment

When providing simulation and visualization capabilities to analysts, a typical desktop configuration (Linux-based workstation) at the user's work area is sufficient. For example, the LDSD project provides time-ordered simulated prescribed motion spacecraft and state data can be provided in file format and/or serialized via Python pickling. Spacecraft data usually includes position and velocity vectors and orientation quaternions, but can also include other ancillary information such as simulated spacecraft health data, camera pointing information, fuel remaining, thermal data,
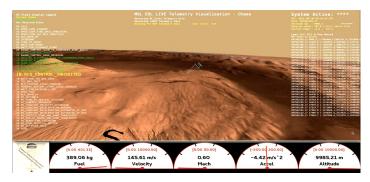

Figure 3. ETV Supporting MSL Landing Night Operations. *ETV used real-time mission telemetry data to display vehicle state and health to mission personnel and the public.*

vehicle states (chute deployment, etc.) or other simulated sensor acquired data. For actual project flight experiments, ETV would be deployed to the flight test range and project engineers will capture live telemetry from the test vehicle. In addition to providing real-time displays to the LDSD engineers, ETV would be configured to run vehicle trajectory simulations, in parallel with the real time telemetry display, to provide predicted vehicle landing or splashdown locations based on the last known vehicle flight parameters, if those predictions are desired by project personnel.

For MSL, ETV provided testbed support during critical operational readiness tests (ORTs) by displaying the spacecraft states in real time from the simulated telemetry generated by the MSL flight hardware/software testbed. On landing night, real time telemetry-based visualization of the final few minutes of the EDL phase was performed by reading the mission telemetry stream and processing the telemetry data that originated in the MSL spacecraft and then relayed back to Earth through the Mars Odyssey orbiter and NASA's Deep Space Network[8] (DSN). Telemetry data was ingested in channelized form that contained spacecraft states were processed by ETV for real time viewing. Post-EDL, MSL engineers are using ETV to playback selected sets of predicted EDL trajectories to better understand actual vehicle performance during those last seven minutes prior to landing. These post-EDL studies and reconstruction will be used to tune and improve current EDL models and tools as well as guide the trajectory design for future Mars or other planetary landings.

IV. High-Performance Visualization

The front-end of ETV is built on top of JPL's Dspace 3D visualization system. With the latest version built on the Ogre3D[9] open source game engine, Dspace supports high frame rate rendering of polygon-based spacecraft and planetary terrain using consumer-level graphics cards and workstations. While core library and API features are implemented in C++, users and high-level control scripts have full access to the API via auto-generated, SWIG wrapped, Python bindings and a Python prompt. When special rendering is required, Dspace makes full use of modern GPU programmability via GLSL[10] shaders and Ogre3D's material scripts. For example, when depicting spacecraft flight paths that transition from orbital to atmospheric altitudes, we developed a simple fragment shader that blends between sky and star map textures based on the altitude of the Ogre3D viewing camera to better depict sky color at all possible altitudes. All of ETV's past and current supported projects requested that real time telemetry updates be displayed on screen in the form of tabular data and a gauge filled dashboard. To support this, we've built a number of python scripts to auto-place, update and color on screen tabular data at run-time and for MSL, we augmented the tabular data with a dashboard that updated whenever telemetry data was received from the MSL spacecraft as shown in Fig. 3. For the LDSD project, we also added 3D CAD ornaments, attached to the spacecraft's reference frame, that depict compass heading, line-of-sight to comm. station, velocity vector and spacecraft coordinate axes and because these ornamental objects are known by the underlying DSENDS simulation, we can accurately update the ornament's pointing and orientation information at every visualization redraw.

Figure 4. ETV Supporting LDSD Flight Simulation. *ETV receives mission telemetrey data and displays vehicle state and health using polygon-based 3D spacecraft and environment representations, as well as run-time updated tabular data and ornamental CAD items such as compass graphics, velocity and line-of-sight vectors.*

Terrain Rendering and Very Large Texture Rendering For MSL

In support of MSL, we have combined the highest resolution Mars Orbiter Laser Altimiter [11] (MOLA) data available with a very high resolution area of Gale crater made up of multiple MRO HiRise [12] digital elevation maps. Our Mars context surface, basically everything that was not Gale crater was constructed from MOLA terrain data and contained about 150,000 triangles. The Gale crater terrain had a much higher resolution and contained about 2.85 million triangles. To color our simulated Mars appropriately, we first applied a lower resolution 16K by 8K pixel Mars context texture to the entire Martian surface. To ensure that we did not run into any OpenGL texture size limits on the variety of graphics cards that we planned to support during development and deployment, we deliberately divided our full Mars terrain into two halves and applied half of the total 16K by 8K to each terrain portion. This resulted in the application of two 8K x 8K pixels textures across our full Martian surface. For the Gale crater landing area, we applied a high-resolution 8K x 8K pixel texture over the crater area. The Gale crater texture, and the portion of the lower resolution Mars texture that overlapped Gale crater were then blended together at run-time in a GLSL fragment shader program to make the transition between the lower context texture and the higher Gale crater texture less apparent. This blending allowed us to display a Mars surface with a more uniform texture color, with no degradation of rendering performance.
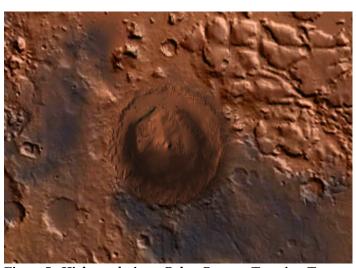

Figure 5. High-resolution Gale Crater Terrain Texture blended With Lower Resolution Full Surface Mars Context Texture. *GLSL fragment shader seamlessly blends high and low resolution Mars textures at run time.*

User Interface Elements and Display Configuration for MSL

As shown in Fig. 3, the MSL mission requested an easy to read dashboard user interface element to help mission operations engineers easily understand spacecraft states during the last 7 m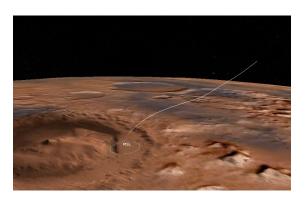inutes of EDL. Working with EDL engineers, it was determined that the five important spacecraft states to watch during EDL were Spacecraft Fuel Remaining, Velocity, Mach number, Acceleration and Altitude. The dashboard elements were updated with data derived from telemetry. Altitude, for example, was computed based on a mission provided reference Mars radius for the early part of EDL and for the terminal phase, by using spacecraft's radar return information relative to a computed Mars surface frame. The actual dial graphical elements are based on a custom Python class that modifies state variables for the underlying GTK [13] widget.

Figure 6. MSL Trajectory View With Final Landing Ellipse

To help tell a complete story for MSL, ETV was configured to display both a spacecraft chase view as shown in Fig. 3, and a spacecraft trajectory view in a second, simultaneous visualization viewport. Both views displayed the current state of the MSL flight system based on flight software states received, in combination with current spacecraft position and orientation data. The trajectory added to the chase view by displaying the spacecraft trajectory line with flight software mode changes

4

attached to the trajectory line at the location that the mode changed. This helped MSL engineers better understand when and where important mode changes, such as parachute or heat shield deployment occurred. To maintain a "hands-off" operation during landing night activities, ETV was pre-programmed to present varying views and camera locations automatically at various times during the EDL phase. These auto view changes were tied to flight software mode changes as those changes were received in the telemetry stream.

MSL EDL engineers also requested a variety of other views of the Gale Crater landing site that displayed MSL along with the Odyssey and MRO orbiters at touchdown time and view of Odyssey from MSL. Other views that ETV supported for MSL included MRO looking at MSL and a view of Mars from along the Earth vector. ETV can support any number of varying and multiple viewports, by attaching viewport cameras to any of the spacecraft or planetary body objects reference frames in the simulation, resulting in a wide range of accurate views into the simulation environment.
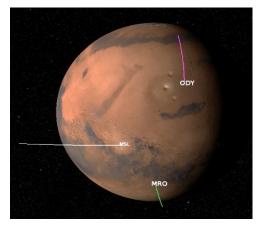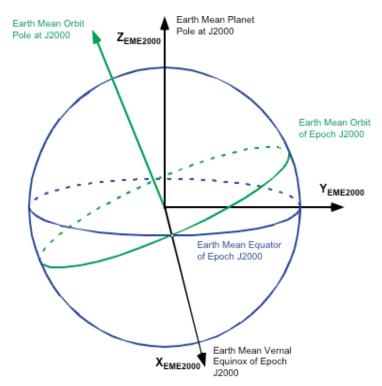

Figure 7. View of Gale Crater, MSL, Odyssey and MRO from orbit range. *Sun illumination is correct for EDL day on Mars.*

V. Multi-Mission Simulation and Support For Multiple Frames of Reference

The ETV software was written entirely in Python using the DSENDS and the Dspace 3D Visualization System. The DSENDS framework is based on The JPL DARTS Lab's Dshell++ Multi-Mission Simulation Framework[14]. Our ETV simulations are typically driven from time-tagged position, attitude and velocity data (no forces were applied to the models) in file format or, in the case of MSL, actual acquired spacecraft telemetry. Dshell++ "body" objects are used to model the relationship between spacecraft and the primary planetary body. Spacecraft bodies are connected to this primary planet body by a full 6 DOF (translation + rotation) hinge. Spacecraft motion during a simulation is controlled by prescribing spacecraft hinges and by using the appropriate spacecraft position and attitude data, relative to the primary planetary body. For space mission simulations that model cruise phase the primary planetary body might be the Sun, Earth or destination other planet and for multi-spacecraft rendezvous scenarios, spacecraft "A" may be connected to a destination planetary body, while spacecraft "B" is connected to Spacecraft "A". This parent-child relationship is required for our simulation bodies, as every hinge must have a parent hinge. Note that this time-tagged spacecraft motion data can be applied to any
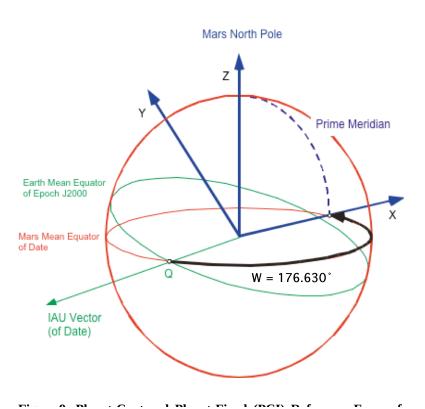
American] **Figure 8. Planet-Centered EME2000 (J2000) Reference Frame for**

spacecraft supported in a given simulation, though spacecraft and planetary body motion data can also be provided via NAIF Spice[15] kernels as well. Typically all of our ETV simulations combine time-tagged motion data with Spice kernel motion data and in the case of our MSL simulation, motion for three Mars orbiters: MRO, Mars Odyssey[16] and Mars Express[17] were Spice kernel-based as were the planetary bodies representing Mars, Earth, Sun and Star Field.

A.   Coordinate Frames

Spacecraft position, velocity and attitude data are always supplied relative to a reference *frame*. Here are some of the frames used for telemetry reconstruction:

1. Planet-Centered EME2000 (J2000) Frame. This is an inertial frame with origin at the center of the planet and orientation aligned with the Earth Mean Equator and Equinox of Epoch J2000 inertial reference system[18]. This is a right-handed Cartesian set of three orthogonal axes chosen as follows: The +Z-axis is normal to the Earth mean equator at epoch J2000, +X-axis is parallel to the vernal equinox of the Earth mean orbit at J2000, and the +Y-axis completes the right-handed system. The epoch J2000 is the Julian Ephemeris Date (JED) 2451545.0. Spacecraft position, velocity and attitude telemetry data are given in the J2000 Frame.

2. Planet-Centered Planet-Fixed (PCI) Frame. This is an inertial (non-rotating) frame with origin at the center of the planet. The +X-axis is defined to pass through the point on the equator defined at zero latitude and zero longitude. The +Z-axis is through the North Pole. The Y-axis completes the right-handed axes system. The J2000 base time (epoch) is always specified by the mission we're supporting and for that specific simulation data set. In Dshell++, the spacecraft position, velocity and altitude are specified in the PCI frame.

3. Planet-Centered Rotating (PCR) Frame. This frame is the PCI Frame with an angular velocity equal to the planetary rotational velocity. A PCR frame is used when simulation CAD ornaments such as surface landing frames, spacecraft trajectory or ground track lines, representations of ground-based compass heading lines and physical structures such as communication towers or buildings need to be placed on a planets rotating surface. For MSL we attached CAD ornaments for spacecraft trajectories and surface frame coordinate axes to the PCR frame and for LDSD, which is an Earth-based simulation, compass and communication structure graphics.

4. Surface Fixed (SF) Frame. This is an East-North-Up (ENU) frame at the planet's surface at or near the landing site. The SF Frame is attached to the PCR Frame so that the SF Frame rotates with the planet. For MSL, this frame is called the "Mars Surface Frame" (*MSF*). The +UP axis is the radial vector from the center of the planet to the planet surface. The +EAST axis is constructed from the cross product of the +Z-axis in the PCR frame and the +UP-axis. The +NORTH axis is the cross product between the +UP and +EAST axes.

For the MSL mission, the acquired telemetry spacecraft position data switched from the J2000 frame to the MSF frame near the end of the EDL phase. Unfortunately, because the

Figure 8. Planet-Centered Planet-Fixed (PCI) Reference Frame.

MSF frame was determined on-board the MSL spacecraft during the EDL sequence and after the landing radar has been activated, and then only stored on-board and not transmitted via telemetry back to Earth, the location and orientation of the MSF frame were not available to our live telemetry playback visualization, so only an estimate of the location of the MSF frame could be derived. Because this type of reconstruction capability fits nicely into the ETV charter, we were able to construct an estimated MSF frame by comparing the last known J2000 spacecraft position with the first received MSF spacecraft position from telemetry. Since the MSF frame location is constructed from the radial position of the spacecraft in the PCR frame, the error of the MSF frame location depends on the time difference between the last known J2000 spacecraft position and actual position of the spacecraft when MSF Frame was created. A post-landing analysis using best available estimates, of the actual landing night MSF frame location, showed that our computed MSF frame was off by about 2.5 km from the actual MSF frame location. On-going, reconstruction of the spacecraft flight path during EDL is facilitated by combining spacecraft telemetry with ground-truth actual descent or orbiter imagery which allow for accurate registration of the spacecraft flight path with the surface of Mars.
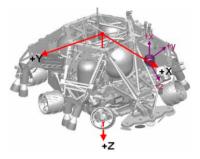

Figure 10. Example MSL Descent Stage Body Frame. *MSL spacecraft body frame as supplied by the MSL mission.*

5. Spacecraft Body (SC) Frame. This frame is attached to the spacecraft body and is typically defined by a mission, based on the mechanical specifications of the spacecraft. In Dshell++, the SC Frame is attached to the PCI Frame.

6. SPICE Frame. This frame is attached to astronomical bodies and spacecraft defined by SPICE kernels. Orbits of spacecraft and planetary bodies obtained from Spice kernels are defined in a SPICE frame.

B. Frame-to-Frame Transformation Operations

At each time step in our simulation, which from simulation to simulation could range from a time step resolution of two seconds or lower for test data to 1000hz or higher for high-rate simulation data, Dshell++ provides C++ libraries, with Python interfaces, to create Frame objects and to transform vectors and quaternions from one frame to another. Below is the procedure used to set a typical spacecraft's position, velocity and attitude from simulation or telemetry data:

1. Obtain spacecraft position, velocity and quaternion vectors in J2000 frame from mission data.

2. Use Dshell++ Frame library to convert the position, velocity and quaternion from J2000 frame to the PCI frame.

3. Set the spacecraft body hinge translation vector to the position in the PCI frame.

4. Set the spacecraft body hinge rotation to the quaternion in the PCI frame.

5. Compute relative ground velocity by converting velocity from PCI frame to PCR frame. The relative ground velocity is used to estimate the spacecraft's Mach number with a zero wind velocity.

VI. System Testing for Correctness

Because ETV was scheduled for use by MSL as an operations tool we needed to test the ETV software for reliability and correctness in multiple ways. First, we fuzz tested to test for the stability of the software. We used the comma separated value (CSV) formatted data input mode of the software for this. This mode of operation is a testing-only mode that simulates telemetry data input. To do the fuzz testing, we mutated existing CSV data files that we captured from a variety MSL mission tests. Mutation involved re-ordering data rows and adding noise to numeric data and flipping of Boolean value data elements. In a scripted loop, we mutated data and fed it into the software to make sure that we exercised the software fully.

We also performed regression testing on our software. Regression testing involved taking software in a known good state and having it render images to disk. These images would be kept under revision control to make sure that

any changes we made in the future would not introduce regressions in the form of visual changes. In the JPL DARTS Lab, we use a continuous integration server. We used this continuous integration server to automatically test our ETV software and all underlying software libraries on a regular basis.

To make sure that bad code was never merged into the main development branch, we made use of our configuration management software's pre-save hook capability. Whenever a developer tries to release the code into the main development branch, the hook would check that the developer ran the developed regression tests and that all tests passed without error. Since we were developing mostly Python code, we used the Pylint [19] static analysis tool for Python. It would find things such as undefined variables, unused variables, and bad indentation (which is important in Python).

In addition to automated testing, we also did extensive manual testing and while a bit tedious, this was the only way to exercise the variety of pathways in the code when users performed a variety mouse operations or keyboard hotkey presses which, for example, triggered a variety of different visual capabilities in the code such as enabling/disabling auto view changes or toggling on/off the MSL landing ellipse. Because we had the help of three interns spending the summer at JPL, we were able to perform many hundreds of manual runs in a relatively short period of time prior to the final MSL readiness tests and of course landing night. For this manual testing, we constructed a spreadsheet-based testing form with questions that the interns and ETV staff members answered regarding system operations at a minimum of ten different and random points during each simulation run. Interns and staff ran our ETV software against each of close to thirty MSL testbed simulation files, checking for correctness. Among other things, they checked the numerical output, displayed as tabular text data on the ETV viewports, against the dashboard gauges to ensure the correctness of the ETV visuals. Because we had about thirty stored MSL test cases, each manual test resulted in approximately seventy data points per run. 3D viewport visuals were also checked during these test runs to ensure that the ETV viewports displayed spacecraft position and orientation as expected, as well as flight system changes such as chute deploy and powered descent mode. As expected, multiple bugs were found by staff and interns, which we quickly fixed.

Finally, and prior to final system deployment, we stress tested the software by running it in a continuous loop. This basically confirmed that no memory leaks or similar bugs existed in the code. We performed this loop test while running on a variety of our development workstations, as well as on the actual deployment machine that we planned to use on landing night.

VII.   MSL Telemetry Processing Module

The MSL Telemetry Processing Module (MSLTelem) is the interface between the MSL EDL spacecraft telemetry and ETV. MSLTelem is written entirely in Python and directly interfaces with the Python mission specific code layer as shown in Fig. 1.

Fig. 11 shows the data flow and interfaces within MSLTelem to the simulation and MSL EDL telemetry. Regardless of where the telemetry originates from; e.g. a test data file, near real-time playback or a live telemetry feed, the processing is the same. MSLTelem will execute a script that queries the mission's channelized spacecraft telemetry data stream for specific EDL data products. MSLTelem processes and maps the data to the required ETV interface inputs that are used to advance the simulation and update the spacecraft state and health information.

MSLTelem is modeled as a finite state machine (FSM) using the FSM Class Module that is part of the DSENDS Simulation Framework. This ensures predictable execution paths with the FSM's defined list of known states, actions and triggering conditions (or events) for state transitions.
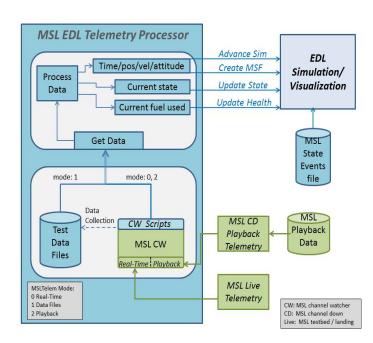

Figure 11.   MSL Telemetry Processing Module Data Flow and

Essentially after initialization, MSLTelem will enter into a polling state waiting for telemetry data that will be processed and sent to ETV.

A. Interfaces

1. MSL EDL Telemetry

The functional requirements for the MSL Telemetry Processing module were developed in collaboration with MSL EDL engineers and documented in an Interface Control Document (ICD). The ICD specifies the EDL real-time data products (RTDP) and variables for spacecraft timestamps, position, velocity, orientation, ground altitude along with their channel IDs, reference frames and valid state ranges. Additional information is documented for spacecraft clock offset, key event timeline, spacecraft states, expected data gaps, data rates, old interleaved data, fuel health data, data priorities and flight rules.

The highest data priority is the state data, followed by the Mars Surface Frame (MSF) data and then the J2000 entry navigation data. This means that there could be cases where the only data MSLTelem receives would be the state data or state data with possibly some portion of the J2000 entry navigation data or MSF data. Note that real time telemetry product rates range from 0.5hz to 4hz and any telemetry data product whose time tag is older than the current received spacecraft clock time and is also outside of the bounds set by that products data rate, is assumed to be older, interleaved, spacecraft recorded data and is ignored.

2. EDL Simulation/Visualization (ETV)

The ETV interface used by MSLTelem consists of three generic and one mission specific interface functions (Fig. 11). The *Advance Sim* function is used to advance the simulation in time and update the spacecraft's position, orientation and velocity in the simulation and visualization. The *Update State* function reflects the transitions in the spacecraft's event states (e.g. heading alignment, parachute deployment, heat shield separation, Mars Surface Frame, backshell separation, rover deployment, etc.) and the onscreen tabular event state list is visualized to show the current state and past received states. The *Update Health* function, specific to our MSL simulation, sends values representing the amount of fuel used by the MSL spacecraft to the onscreen fuel remaining gauge.

The fourth interface was an additional functionality added to ETV for MSL to support a transition of reference frames from J2000 to MSF navigation. The MSL spacecraft calculates the MSF on-board, however that MSF information is not contained in the telemetry stream, therefore ETV generates its own predicted MSF. The *Create MSF* interface function was added to support the creation of this simulated MSF frame. MSLTelem provides ETV with the spacecraft position and orientation information from the latest J2000 reference frame along with the first valid spacecraft position and orientation in MSF. ETV can then reconstruct an MSF at this reference frame transition. Additionally there are two off-nominal cases that can affect the creation of the MSF. In the off-nominal case where no J2000 positional data is available MSLTelem still requests ETV to generate an MSF. However, without the J2000 data, ETV has to make certain assumptions that result in the creation of a simulated MSF that resides in the center of the landing ellipse. In the second off-nominal case where in addition to no J2000 positional data items there are also no J2000 timestamps. In this case MSLTelem isn't able to determine valid time stamped groupings of the MSF positional data, therefore no MSF can be created. So for this second off-nominal case, the ETV onscreen display would not get any spacecraft positional updates but would still likely be receiving the updated state data which is the highest priority data. Note that additional details on MSF are described in the Section V.

A. Testing

As shown in Fig. 11, MSLTelem runs in three modes that designate if the telemetry is from a test data file, live telemetry stream or near real-time playback. MSLTelem feeds the data through the same Python sub-process pipe mechanism regardless of its origin so essentially telemetry data source is independent from the actual processing.

The MSL test data files were collected during testbed tests and were used to support ETV system development unit tests, system integration and stress testing. The data files also provided a means to generate other test conditions such as data gaps and interleaved data, as well as being massaged for stress testing. More information is provided on the system regression and stress testing in Section VI.

Live testing was performed with the MSL testbed data whenever the MSL EDL engineers ran an EDL test or whenever there was a flight software regression test or a second chance test might result in nominal or off-nominal

spacecraft performance. Live testing also provided the capability for MSL EDL to use a channel simulator that generated a telemetry stream with data gaps or entire dropouts of J2000 or MSF data.

The capability of a near real-time playback of previous testbed telemetry data using the MSL Channel Down tool allowed simulated live testing when live testbed testing wasn't available, as well as being able to repeat live testing with near real-time feed.

Additionally, during some tests the runtime output was logged in case there was a need for post-test analysis of ETV's behavior. For example, this output data was used to help understand the data gaps in telemetry and how the simulation was handling those data gaps. Initially, when data gaps were encountered the visualization trajectory trails showed a step-wise jump when there was a gap as opposed to an expected straight-line segment. Data plots from the logs clearly illustrated the data gaps and revealed that the simulation needed to make an adjustment in how the spacecraft simulation body was rotated along with the Mars surface in order to handle these data gaps properly.

Verification of system correctness as it related to the simulated spacecraft's final landing location was done by visual comparison with MSL predicted simulation runs and reviewed by the MSL ETV and MSL EDL engineers. Additionally, for a subset of MSL testbed runs, we inspected ETV's final reported spacecraft landing locations and compared against MSL EDL engineer-provided spacecraft positions with respect to the MSL Landing Target Frame (LTF).

Testing helped define ETV operational processes that were eventually included into the MSL EDL operational timelines for ORTs and Landing night. These checks included such items as whether the simulation epoch required updating due to spacecraft clock drift; updates to the landing ellipse size and location; telemetry data flow pipeline check-out queries; and confirmation of the runtime telemetry channel query parameters by MSL EDL engineers Additionally, a separate channel query process was set up to collect telemetry data to be added to ETV's collection of test data sets.

VIII. Conclusion

The JPL developed EDL Telemetry Visualization system (ETV) has been developed as a multi-mission tool to support a wide range of NASA space mission simulations, vehicle and model performance reconstructions and interactive visualizations via playback of mission predicted, simulated spacecraft state information as well as from actual spacecraft telemetry and mission acquired imagery. Accurate representation of the modeled flight system or systems, including any number of spacecraft with full 6 DOF joints, multiple reference frames and environmental data such as planetary gravity, atmosphere, lighting and terrain are supported. A powerful high fidelity, physics-based simulation framework forms the basis of the ETV system and can perform a wide range of predictive computation for spacecraft vehicle performance and trajectory through the use of built in or third-party provided spacecraft sub-system computer models.

The design and development of the ETV software leveraged existing, mature, and widely used JPL simulation tools and frameworks. This means that we only need to build thin, mission specific code layers that access the thick layer of existing multi-mission features and capabilities of the simulation and visualization frameworks. Because these existing tools have been fully verified and validated for accuracy and come with an extensive regression test suite, we can easily maintain the accuracy and correctness of our deployed ETV systems as we move from project to project.

Future work considerations for simulation, visualization and telemetry processing will include the development of a generic multi-mission telemetry module that will be completely driven by standardized data and command dictionaries which fully define the data products, data rates, and flight rules such as knowing which data products are valid during various portions of a mission simulation, data or reference frame conversions, possible spacecraft clock offsets, and the handling of interleaved time-ordered data. In addition, full setup and runtime operation of the 3D visualization, including layout of onscreen text data, dashboard gauges and camera viewpoint transitions could be driven using a standardized specification. We will also continue to work with MSL, LDSD and future missions to further enhance our suite of simulation and reconstruction tools.

Acknowledgments

References

[1]NASA's Mars Science Laboratory. http://mars.jpl.nasa.gov/msl/

[2]NASA's Low Density Supersonic Decelerator. http://www.nasa.gov/mission_pages/tdm/ldsd/index.html

[2]Dynamics and Real Time Simulation Lab. http://dartslab

[3]J. Balaram, R. Austin, P. Banerjee, T. Bentley, D. Henriquez, B. Martin, E. McMahon, G. Sohl, "*DSENDS - A High-Fidelity Dynamics and Spacecraft Simulator for Entry, Descent and Surface Landing*", IEEE 2002 Aerospace Conf., Big Sky, Montana, March 9-16, 2002.

[4]M. Pomerantz, A. Jain, S. Myint, *"Dspace: Real-time Visualization System for Spacecraft Dynamics Simulation*", Third IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT 2009), Pasadena, CA, July 19-23, 2009.

[5]Simplified Wrapper and Interface Generator. http://www.swig.org/

[6]Python Programming Language. http://www.python.org/

[7]R Madison, M Pomerantz, and A Jain, "*Camera Response Modeling and Verification in ROAMS*," i-SAIRAS 2005, Munich, Germany, September 2005.

[8]Nasa's Deep Space Network. http://deepspace.jpl.nasa.gov/dsn/

[9]Ogre Open Source 3D Game Engine. http://www.ogre3d.org/

[10]OpenGL Shading Language. http://www.opengl.org/documentation/glsl/

[11] Mars Orbiter Laser Altimiter. http://mola.gsfc.nasa.gov/

[12]High Resolution Imaging Science Experiment. http://hirise.lpl.arizona.edu/

[13]GTK. http://www.gtk.org/

[14]C. Lim, A. Jain, *"Dshell++: A Component Based, Reusable Space System Simulation Framework"*, Third IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT 2009), Pasadena, CA, July 19-23, 2009.

[15]NASA's Navigation and Ancillary Information Facility. http://naif.jpl.nasa.gov/naif/

[16]NASA's Mars Odyssey. http://mars.jpl.nasa.gov/odyssey/

[17]NASA's Mars Express. http://www.esa.int/esaMI/Mars_Express/index.html

[18]P.D. Burkhart, "*MSL Update to Mars Coordinate Frame Definitions*", JPL IOM 343B_2006_004, August 15, 2006.

[19]Pylint, Python Code Static Checker. http://pypi.python.org/pypi/pylint/