

Ground Data System Analysis Tools To Track Flight System State Parameters for the Mars Science Laboratory (MSL) and Beyond

Dan Allard¹ and Lloyd Deforrest²

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, 91109

Flight software parameters enable space mission operators fine-tuned control over flight system configurations, enabling rapid and dynamic changes to ongoing science activities in a much more flexible manner than can be accomplished with (otherwise broadly used) configuration file based approaches. The Mars Science Laboratory (MSL), Curiosity, makes extensive use of parameters to support complex, daily activities via commanded changes to said parameters in memory. However, as the loss of Mars Global Surveyor (MGS) in 2006 demonstrated, flight system management by parameters brings with it risks, including the possibility of losing track of the flight system configuration and the threat of invalid command executions. To mitigate this risk a growing number of missions have funded efforts to implement parameter tracking parameter state software tools and services including MSL and the Soil Moisture Active Passive (SMAP) mission. This paper will discuss the engineering challenges and resulting software architecture of MSL's onboard parameter state tracking software and discuss the road forward to make parameter management tools suitable for use on multiple missions.

I. Introduction

The Mars Global Surveyor (MGS) spacecraft was lost in early November of 2006 following ten years of successful operations. The root cause of the failure was found to be a flight software parameter change that had corrupted two parameters at the same time. Both parameters manage behavior of the High Gain Antenna (HGA).¹ The incorrect parameter change and resulting parameter corruption was not identified until loss of MGS, some five months later. When a later flight command referenced the corrupted parameters, a complex, anomalous chain of events occurred resulting in the HGA pointing away from the Earth, which exposed its batteries to the sun. This caused the batteries to quickly overheat and drain. From that point on, the spacecraft was never contacted again. The fact that such a corruption could linger unnoticed for five months ahead of the eventual spacecraft loss highlighted a major discrepancy in the flight/ground systems in the area of flight state tracking, and in particular in the area of in-memory flight system parameters.

In-memory flight system parameters are relied upon by flight software to configure and manage a wide range of space vehicle behaviors. Understanding the state of these parameters is critical for flight mission operators to safely manage science activities and engineering operations.

Flight system operators need to track parameter state for a variety of purposes. During standard daily operations, when parameters are "set" onboard it is important to validate that the updated state is correct before continuing science activities; otherwise there is a risk that systems and instruments will be commanded to an unsafe configuration. During periods of transition, such as major flight software (FSW) upgrades, all of the mission flight software parameter settings must be evaluated before the full transition can be completed. Another area of parameter use is in the initialization of flight software testbeds, where science-planning scenarios are exercised. This ensures the software testbed is in exactly the same state as the flight system for the science scenario test execution. And of course a complete and correct understanding of state is critical during anomaly investigation.

On the MSL project, late in the preparations for launch, a need was identified to manage MSL's growing numbers of parameters in some sort of tool that would enable automated tracking and provide reports of onboard parameter state. Three months prior to launch the project authorized development of the Parameter Management

¹ MGSS Ground Software Engineer, Mission Control Systems, 4800 Oak Grove Drive, M/S: 301-480.

² Technical Group Supervisor, Mission Control Systems, 4800 Oak Grove Drive, M/S: 301-480.

Toolkit (PMT) to satisfy this need. The resulting tool took the form of a centralized web service with a back-end database, with the initial version deployed to MSL operations just ahead of the November 2011 launch and further versions deployed for cruise and surface operations. Despite all of the challenges inherent in the MSL software implementation, the PMT development team made every effort to provide as “multi-mission” a tool as possible, by making use of proper abstraction layers and componentization. An adaptation has just been delivered for the Soil Moisture Active Passive (SMAP) earth-orbiting mission, and future mission versions look likely.

This paper will discuss the Parameter Tracking problem in more detail, our approach to solving the parameter management and tracking problem through development of the PMT, the use of PMT during the cruise and surface phases of the MSL mission and will finish up with some conclusions about the future of the PMT.

II. Definitions

It is helpful to define a number of terms related to the problem of parameter tracking:

<i>Term</i>	<i>Definition</i>
Parameter Management Toolkit (PMT)	A collection of tools including a server application providing the functionality to track mission parameters.
Telemetry	Data delivered from the spacecraft to the ground systems, in a variety of formats.
Channels, Channelized Data	Refers to the sensor values sent down by the Flight Software (FSW). Each flight component’s sensor is allocated a “channel” with a unique ID, usually in the form of a [stem]-[number]. The [stem] could be a letter representing a flight module or an abbreviated flight module name, such as T-0001 or THRM-0001, respectively. Channel data are collected by the FSW and packaged as packet payload for transport.
Event Record (EVR)	Event records are messages from the FSW indicating the occurrence of a certain event. Events are identified by a unique (to the mission) event ID, which is defined in an EVR dictionary. Event records are associated with a criticality level (e.g. ACTIVITY_HI, WARNING_LO, FATAL, etc.) and may contain additional metadata to further describe the event. Event records are collected by the FSW and are packaged as packet payload for transport.
Data Product	Any collection of data generated onboard the spacecraft. Data products are “files” in the FSW and may contain anything from channelized data to science data from specific instruments. They are packaged as packet payload for transport and may be processed by AMPCS or a tool downstream from AMPCS, depending on the packet contents.
Flight Software (FSW) Parameter	System control and configuration values that reside in flight system memory. This refers both to values in “volatile” and “non-volatile” memory. FSW parameter values are typically modified by uplinked commands.
MPCS	The Mission data Processing and Control System, a new ground system developed in support of MSL.
Spacecraft Event Time (SCET)	The time that an event has occurred on the spacecraft.
Volatile	Volatile memory, specifically the memory for parameter values.
Non Volatile Memory (NVM)	Non-volatile memory for parameters.
Context or Context Revision	A PMT database structure that includes all of the information required to query all of the parameter state at any given time where parameter samples have been collected. The two terms will be used interchangeably.
Module	A component of the flight software organized around a flight subsystem, e.g. Arm, controlling the spacecraft arm behavior, Thermal, controlling spacecraft temperatures, etc. Every parameter is associated with a single FSW module.
Group	An organizational construct that encapsulates a subset of a FSW module’s parameters.
Group Copy	A parameter index structure within a group.
Parameter Name	A name of a parameter. Note that this is not the value of the parameter.
Snapshot	A parameter state report of the most recent values as of a given SCET.

PMT Related Spacecraft Commands	SET	Update a set of values in volatile parameter memory
	SETSAV	Update a set of values in volatile memory and save updated values to NVM
	DMP	Dump a file containing a sample of parameter state for a FSW module or group.
	SETSAVDMP	As in SETSAV but complete with a dump file of the values changed.

III. Context for Development

The need for a new software application that manages MSL’s flight software parameters had been discussed for several years prior to MSL launch, but was only green-lighted 3 months prior to launch. Many factors contributed to the delay in the decision to begin implementation of PMT including pressure to finalize the launch version of the flight software, availability of ground software engineers to work on such a tool and the late realization that such a tool was even needed.

In preparation for the MSL mission launch, a team of developers was formed to implement the PMT service in support of cruise and eventual surface operations. PMT originated with lessons learned from the Mars Exploration Rover (MER) mission’s parameter tracking, which was largely performed by loading ground-identified parameter state values into a spreadsheet tailored for tracking purposes. While MER tracked around 3,000 parameters, MSL would track over 54,000 distinct parameters and would see a great deal more change activity than MER, and so clearly a spreadsheet based solution would not be feasible. The development team built a database-backed web service which would automatically collect and store parameter information following the completion of a data downlink pass. This new service provided “snapshots” of parameter values across different flight software subsystems (also called modules), as well as parameter state histories.

The MSL implementation of PMT faced many engineering challenges. A primary issue was that by the time the PMT implementation started, the flight software was years into development and largely “locked down” from further changes. Detailed end-to-end engineering of the parameter tracking problem was not performed until the PMT implementation began, so any issues or inconsistencies identified in the flight software arena had to be dealt with by architectural workarounds that fell squarely on the PMT implementation. For example, parameter dictionaries were developed for human readability as HTML web documents as opposed to machine readable structures such as XML and so a good deal of work was required by the ground system engineers to identify the set of parameters to track.

Approximately 18 months prior to launch, work was underway in earnest on a similar tool, called the Data Management Toolkit (DMT). DMT is a Web-based tool that tracks the state of on-board spacecraft products, determines if the product has been received on the ground and if not, re-requests the product for later transmission to the ground. If the product is determined to be on the ground, DMT makes a request to the flight software for that product’s deletion from the spacecraft’s product cache. The DMT accomplishes this task by ingesting data product summary reports into a local database and then provides visualization of the data product’s state and tools to either delete the product from the spacecraft’s product cache or re-request the product’s transmission to the ground. DMT was developed in conjunction with MSL’s next generation command and telemetry management system, called the Mission data Processing and Control System (MPCS), which for the first time at JPL, makes all of a mission’s telemetry products (Channels, EVRs and Data Products) available through a set of well-defined database APIs. In this way, DMT makes simple database queries to receive the latest, or earlier, data product summary reports. When PMT was green lighted, it seemed that the DMT had a similar design to what the PMT needed and that PMT could inherit significantly from the DMT work, whose launch version was completed 3 months prior to MSL’s launch. One ongoing challenge to the PMT work was that the DMT was always considered much higher criticality, and needed DMT improvements continuously trumped ongoing PMT development

Another challenge to developing the PMT at such a late stage in MSL’s lifecycle was the complete lack of documented parameter management concept of operations, and design and interface specifications. Since the flight system’s parameter management approach evolved at a quickened pace as time drew nearer to the launch date, very little was documented to guide Ground Data System (GDS) engineers in the development of tools to handle MSL’s large and exceedingly complex parameter situation. By the time the PMT development started it was far too late to negotiate flight software changes that would lessen MSL’s parameter management complexity on the ground.

Implementation of the PMT began approximately 3 months prior to MSL launch at a fevered pace. The PMT implementation and test team consisted of three half-time developers, most of which continued to support the DMT after its initial release, concurrent with PMT development.

IV. Parameter Tracking Problem Space

To understand the considerations for the architecture of the service we need to review the problem space as a whole. Following are key concerns that drive the PMT design:

A. About Flight System Configuration by In-Memory Parameters

Configuring flight system behavior involves more than simply commanding specific actions or even activities. There are far too many individual flight system “control knobs” to command at once to the proper setting (using up valuable activity time and uplink bandwidth), so flight systems are “pre-configured” to be in the right state for the ongoing flight system activity.

One typical approach to spacecraft configuration is a table-driven approach, which involves specification of configuration settings in an onboard persistent table structure such as a file. There may be multiple tables onboard the spacecraft with a single table specified as “prime” so to speak, and the others ready to take over as prime via ground commands. For example, Command Behavior Module (CBM) tables contain a collection of configuration information defining and specifying uplink and downlink behavior (e.g. specification of data rates between the lander and orbiters).

In contrast to the table-driven configuration approach, use of flight software parameters represents a more recent trend to set and save configuration values by managing values in flight system memory. For example, the thermal team, responsible for keeping the spacecraft within temperature tolerances, will set parameters to specify the “target temperature” of thermal components to drive heating behavior. Typically, “live” parameters are in RAM (often referred to as “volatile” memory). These RAM values can be modified by command sequences over the course of daily activities. Persistent store is typically referred to as “Non Volatile Memory” or NVM. NVM parameter values typically represent a “baseline” of parameter state. For a flight system with a sleep/wake cycle, as with MSL, volatile parameter memory is instantiated from NVM upon vehicle wake-up. Another way to look at it, when the spacecraft goes to sleep, Volatile memory disappears, and any changes to Volatile that are not made to NVM will not be present on the next wakeup.

B. Parameter Tracking End Use Cases

There are a number of use cases for ground tracking parameter services:

- 1) *Validating Commanded State* - On any given day, the system operators needs to validate that parameters are set to the correct values before proceeding with activities dependent upon those parameter settings.
- 2) *Identifying Anomalous States* – The system needs a method for identifying anomalous states.
- 3) *Trending Parameter Changes Over Time* – The system needs to provide a method to identify expected/unexpected patterns of usage over long periods of time, including life-of-mission.
- 4) *Testbed Instantiation* – The system needs to enable instantiation of testbeds with parameter states at all points of a mission, including over the course of operations, so as to properly simulate flight activities.
- 5) *FSW Transition/“Toe-Dip”* - The system needs to support transitions from one FSW version to another, including a “toe-dip” step, where the spacecraft is commanded to wake up with the new version, downlink a number of configuration states including all parameter values, and then transition back to the original FSW version while engineers analyze the recently downlinked results ahead of a full FSW version transition.

C. Defining and Identifying Parameters

FSW modules define parameters. Legacy missions tended to define parameters completely in exclusion from each other, with no standards in terms of how values would be commanded or monitored. As missions like MER began to take advantage of automated code generation, standards have been introduced, such as command formatting (e.g. common command stem formats) and monitoring (e.g. common dump file formats of parameter state changes). The MER-inherited MSL auto-code FSW generation includes the creation of a parameter dictionary, albeit in HTML and CSV form for human readability rather than, say, XML. In contrast, the SMAP mission defined parameters directly in XML and delivered a parameter dictionary in addition to the other core flight dictionaries such as frame, packet and channel definition dictionaries.

Parameter organization and meta-data also varies across missions. MSL flight software organizes by the following:

- Flight software modules (also called subsystems, e.g. Thermal, Power, Arm, etc.)
- Groups within a software module (e.g. Arm groups Frame, Stow, Fault Protection, etc.)
- Parameters in a group. Many of MSL's modules are indexed, either numerically or by string, by a "group copy", such that a group of X parameters with Y group copies identifies X * Y distinct parameter values.

The following diagram is a screenshot of the HTML output of an MSL auto-coded parameter dictionary for the group Monitor (MON) of the Battery Control Board (BCB) Module:

MON Parameter Group					
Parameters specifying fault monitor configuration					
<p>There are 5 versions of this group, with the following enumerated indices (from FSW type <code>BcbFaultMonitor</code>; <code>ILLEGAL</code>; <code>BCB_NUM_MONITORS</code>):</p> <p><code>batt_offline_at_boot_bcb1</code>, <code>batt_offline_at_boot_bcb2</code>, <code>batt_offline_imminent_bcb1</code>, <code>batt_offline_imminent_bcb2</code>, <code>fail_to_communicate_with_both_bcbs</code></p>					
Parameter	Type	Default	Units	Range	Description
<code>detection_enabled</code>	Bool (in <code>gbl/gbl_pub.h</code>)	TRUE	none	FALSE : TRUE	<i>Whether or not detection is enabled for this monitor</i>
<code>response_enabled</code>	Bool (in <code>gbl/gbl_pub.h</code>)	TRUE	none	FALSE : TRUE	<i>Whether or not response is enabled for this monitor</i>
<code>error_persistence</code>	U32	1	none	1 : 0xFFFFFFFF	<i>Error count at which monitor becomes YELLOW</i>
<code>fault_persistence</code>	U32	1	none	1 : 0xFFFFFFFF	<i>Error count at which monitor becomes RED</i>

Figure 1: A sample of an auto-coded FSW dictionary entry for the Battery Control Board (BCB) parameters. The entry is for the Monitor group, responsible for detecting BCB errors. The 5 "versions" (or "group copies") refer to different components or properties of the BCB that are to be monitored. Listed are the group copy index names, and the definitions of all the parameters associated with each index.

Expanding the group definition into a table shows that there is a distinct parameter value for each combination of Parameter Name and Group Copy:

Group Copy Param Name	batt_offline_at_boot_bcb1	batt_offline_at_boot_bcb_2	batt_offline_imminent_bcb_1	batt_offline_imminent_bcb_2	fail_to_communicate_with_both_bcb
Detection_enabled	False	True	True	True	False
Response_enabled	False	True	True	False	False
Error_persistence	10	5	1	1	1
Fault_persistence	20	5	5	1	1

The given group would have a total of 20 parameter values, one for each pair of version (group copy) and parameter, defined by the combination of each. For example, “batt_offline_at_boot_bcb1_detection_enabled” is one parameter value, and “batt_offline_at_boot_bcb2_detection_enabled” is another.

One key difference between MSL’s parameter definitions and SMAP’s is in how parameters are identified in the dictionary and in the downlink stream. For MSL, parameters are defined by their meta-data, and the identifier has to essentially be derived from a union of “module name + group copy + parameter name”. The identifier is not called out explicitly in the dictionary. For SMAP, each parameter in the dictionary has a unique identifier, along with some additional meta-data such as the module (which still features in the parameter identifier) and “operational category”. Note that the MSL construct-an-identifier scheme would be the source of integration problems as will be discussed under section V, Implementation, below.

D. Command and Control

Flight parameters are normally set in one of two ways: either as the default value for a given flight software version/module, or else “commanded” to various states via a ground-commanded sequence execution. It is possible that a given parameter is not modified at all from the default over the lifetime of a mission. This suggests, however, that perhaps that parameter should not be a true “configurable” parameter in the first place.

MSL parameters can be commanded in a few specific ways:

- Values in Volatile memory can be “set” to a specific value with a SET command.
- Values in Volatile and NVM memory can be set with a SET_SAVE command.
- All of the parameters in a group or a module can be saved from Volatile to NVM by way of a GROUP_SAVE and MODULE_SAVE command.

One characteristic of MSL parameter changes is that changes can only be made to parameters in groups. It is not possible to actually modify a single parameter in a module or group.

There are other commands associated with parameters that are associated only with monitoring, which will be discussed in the next sub-section.

E. Parameter State Monitoring

To understand the challenges of parameter state monitoring, one must keep in mind that there is no way to have 100% assurance (or *confidence*) of any given onboard state value as of current Earth time “now”. It is important to remember that, as with any other flight system state, parameter values are only “known” on the ground as of the time of data receipt. At best any state knowledge is already at least as old as the delivery time from the remote asset to the local GDS, and unless there is a continuous stream of reported onboard state samples being delivered by the spacecraft, it is likely that state knowledge is quite a bit older than that. And since it isn’t feasible to downlink a constant stream of parameter states, the knowledge of state “ages” over time. As parameter state knowledge grows old, so does the risk from unexpected/unmonitored value changes increase.

Actual monitoring of parameter values and value changes can be accomplished in a number of different ways. The most simple is to request a sampled “memory dump” of Volatile and/or NVM state, which itself can be delivered via any of the available telemetry types (Channel, Event Record and Data Product). However the lion’s share of parameter value samples are delivered via products.

The downside of the monitor-by-sample data approach to monitoring is that it utilizes a significant percentage of both available onboard CPU activity time and downlink bandwidth. It uses activity time in the production and downlink of dump files, and of course the dump files eat up valuable downlink. As it happens, MSL does not have the tight constraints on downlink volume due to the high lander-to-orbiter data transfer rate that largely results from the Adaptive Data Radio (ADR) utilized on the Mars Reconnaissance Orbiter (MRO). MSL has a bit more flexibility on the choice of downlink as compared with legacy lander missions such as MER. Still, downlink capacity remains a carefully monitored resource. Also, it is possible that a parameter dump may not be available to the GDS due to data loss in transit.

There are other ways of tracking parameter changes besides monitor-via-dump-files. Value changes can be inferred through validated sequence execution. When parameter change commands execute successfully (or not) onboard the vehicle, a record of command execution is reported, typically in an EVR.

Besides file dump and EVR based parameter value state change tracking, the flight system provides a few more ways that parameter evidence can be validated, if not identified outright. Checksums of parameter memory are downlinked regularly, and can be used to validate whether or not parameter values across modules and groups have changed. Command “set” counters can be used to identify missed command executions. While these “secondary” sources of parameter change evidence may not be useful to actually identify parameter values, they can be used to identify where known parameter values have been changed.

One significant MSL flight system discrepancy is the inability of flight software to downlink the NVM state directly, despite the NVM being the information that is most important to track. Volatile-only updates only last until the next sleep/wake cycle and are thus transient. MSL NVM state can be inferred in a couple of different ways. For example, if there have been no changes to parameter state for a given module since the time the spacecraft woke up, then a Volatile module dump will also represent the state of NVM. The same is the case if there are no Set commands between a MODULE_SAVE and a dump file, then the dump file state represents both Volatile and NVM.

V. Early Design and Implementation

A. Requirements and Concepts

The PMT implementation team began work by sitting down with mission customers and reviewing use cases, identifying requirements, and laying out the parameter identification and tracking interfaces. Early on it was conceived that a “software service” with a back-end persistent store would best encapsulate the overall problem.

Key requirements on this software service identified early on included:

- Track all parameter values over the lifetime of the mission
- Track parameter evidence at the Module and Group/Group Copy level
- Track secondary sources of evidence such as checksums and counters
- Provide access to parameter data via command line tools as well as broadly accessible graphical user interfaces (e.g. web interfaces)
- Can be used in a wide range of “venues” in support of testing, analysis and operations
- Be capable of “triggered” automated updates of the database on a pass-by-pass basis
- Provide a query of a “snapshot” of latest available known state, up to and including a snapshot of “all known parameter state as of time X”
- Provide a query of a “history” of parameter updates over a time range
- Provide data in a variety of formats, ideally modifiable by end-users
- Can be used to initialize a testbed to a known parameter state

In support of these requirements, the PMT team identified a set of “information concepts” to represent the different types of information the system would need to manage. Most of these concepts are eventually represented as “information objects” by the software service. The following table lists the primary concepts:

Concept	Description
Venue	A means of organizing parameter data within the server. All parameter data tracked by the service is associated with one Venue or another, and once associated with a Venue, is

	distinct from any data in any other Venue. A server may support a single Venue, such as the “Surface Operations Venue”, or may support N venues such as “testbed1, testbed2...”
Dictionary	A representation of the definition of parameters. The dictionary includes all the information derived from the MSL auto-coded dictionary, including the parameter name, module, group, group copies, units, description, default values, and ranges.
Evidence	A specific source of parameter knowledge. For example, if the parameter values are reported in a data product file, then the Evidence will be a “product”.
Trigger	A representation of the act of automated data collection. A trigger may be a directive such as “collect all the new parameter data found between time A and time B”.
Snapshot	A latest-available-data view of parameter values. Snapshots can be filtered down to the module, group, group copy and parameter levels, or may be queried for “latest available state for all tracked parameters”.
History	A view of parameter value changes over time, filtered in the same way as a Snapshot.
Template	A means to format data.

The resultant design involves a set of tools built around a layered service and database architecture. A ReST service layer fully encapsulates a back-end database. The following diagram depicts the components of the service and the key external interfaces:

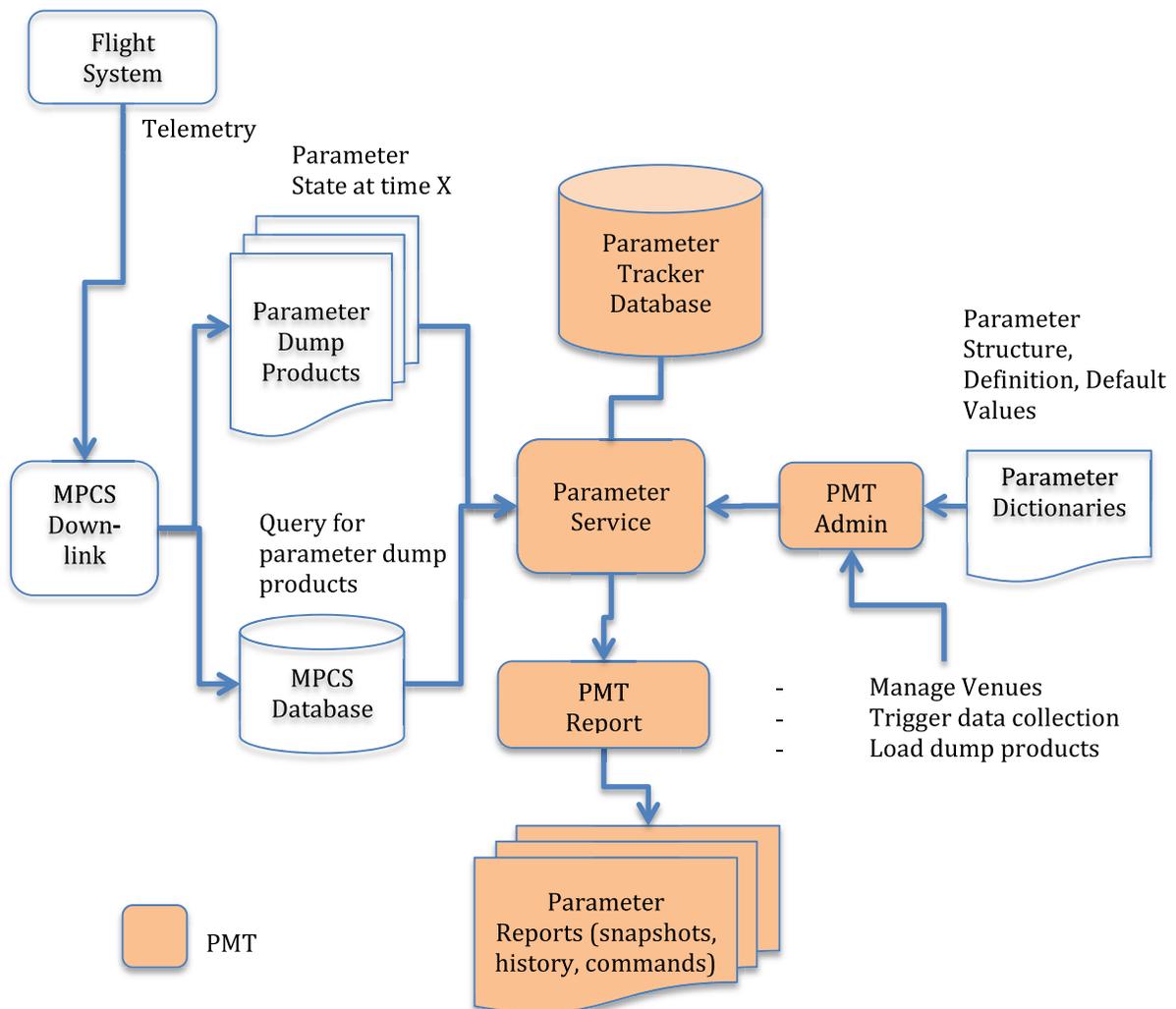


Figure 2: Component view of the architecture of the PMT. A centralized service encapsulates all interactions with the persistent store. Administration tools manage the service and drive the data collection, and reporting tools provide snapshot and history queries. The MPCS database and MPCS processed dump files provide the prime source of parameter evidence.

In this architecture, the Parameter Service is the “hub” for all inbound and outbound information (thus, the PMT should be considered to be compliant with a “Service Oriented Architecture” (SOA)). It completely encapsulates a Parameter Tracker Database, which stores all service-related information. External tools include the administration reporting scripts. The administration tool is used for everything related to service setup and data collection; basically everything except for the queries of parameter data. Additional snapshot and history scripts are provided as separate tools.

B. Application Design

As mentioned, the chosen architecture was a service model completely encapsulating a persistent store database. In particular, the interface to the server was chosen to be Representational State Transfer (ReST), such that all interactions with the server were HTTP GET and POST calls meeting the ReSTful specification. Essentially, to all outside sources, the server would look just like any other ReST-ful web server.

One characteristic of the ReST is that information is provided as “resources”. Ideally, resources are *nouns* identifying information. For example, a resource providing a snapshot of information might be called “Snapshot”, as opposed to a resource that provides “GetSnapshot” and another “PublishSnapshot”. You would GET and POST to the Snapshot resource instead.

Besides the intrinsic values of utilizing the ReST pattern, the ReST pattern itself has been seeing broadening use across JPL mission systems and underlies communication interfaces across a growing number of mission tools.

As for the implementation language, Java was chosen for a few reasons. For one, all team members were primarily Java developers, so there would not be an up-front cost in learning a new language. Another reason is the availability of a broad suite of Java libraries, both core and 3rd party including a great deal of high quality open source. Additionally, PMT developers were pulled in from other implementation tasks to meet the short development timeframe, and had in hand a number of already implemented components including time support and XML manipulation in particular.

To implement the web service layer, the team chose ReSTlet. This open source Java provides a framework for running an application as a ReST server. With ReSTlet, certain classes are identified as providing ReST resources, with the framework handling the translation of the input POST and GET into one or more accept() methods in the class.

The server was implemented to provide the following ReST resources:

Resources	POST Behavior	GET Behavior
Venue	Create a new Venue	List available venues
Dictionary	Load a new dictionary	List available dictionaries or retrieve a dictionary
Evidence	Load data evidence from a source file to the server	List evidence sources files loaded to the server.
Trigger	Kick off automated data collection according to the parameter specified in the trigger.	List history of triggers executed
Snapshot	None	Retrieve a snapshot of latest available parameter values as of a specified input time
History	None	Retrieve a history of parameter updates from specified time A to time B
Template	Load a new template to the server	Retrieve a list of templates from a server or retrieve a single template from a server according to input parameters

To manage persistent data, the team chose to utilize the “Hibernate” persistent data framework, rather than interact with a database directly. Hibernate solves a problem of *object-relational mapping*, in that it allows an application developer to implement persistence as “objects”, whereas the database representation is relational.

In Hibernate, a class represents each persistent data type, and each persistent class is specified in a persistence definition file. Each class in the definition file will have at least one database table representing that object information. Specifically, each “get” method of an object maps to a column in the table. For example, the object type “Venue” has a “getDescription()” method, therefore it will have a Description field in the database.

One benefit of the Hibernate is that it manages its own database schema. When properly configured and executed, Hibernate automatically instantiates the entire schema from the persistence definition in persistence.xml. Relations between objects are automatically handled in index-mapping tables, relating the unique index of each object to the other. If the persistent object definitions are modified, the database tables are automatically updated to reflect the changes.

Note that the persistent types line up fairly closely with the ReST resources, although what is persisted is not necessarily what is being delivered by POSTs or returned from GETs. For example, when the user POSTs a Trigger, this kicks off an automated data collection. But what is persisted is a record of the Trigger event.

Rather than pre-define how data would be displayed to the user, the team chose to take advantage of a template engine called Velocity to transform raw queried data into useful user forms. Velocity is a lightweight templating language that allows for specification of data transforms via a template in a scripted, programmable manner. Basic Velocity templates provided have included HTML and Comma Separated Value (CSV) with a range of selected data columns.

C. Query Design

A key feature of PMT is its ability to query parameter states from any point in time. To understand the context for query design we need to take a look at the patterns of data retrieval as well as the desired query results. The following diagram shows how groupings of parameter value samples arrive over time forming a “sparse” matrix of results:

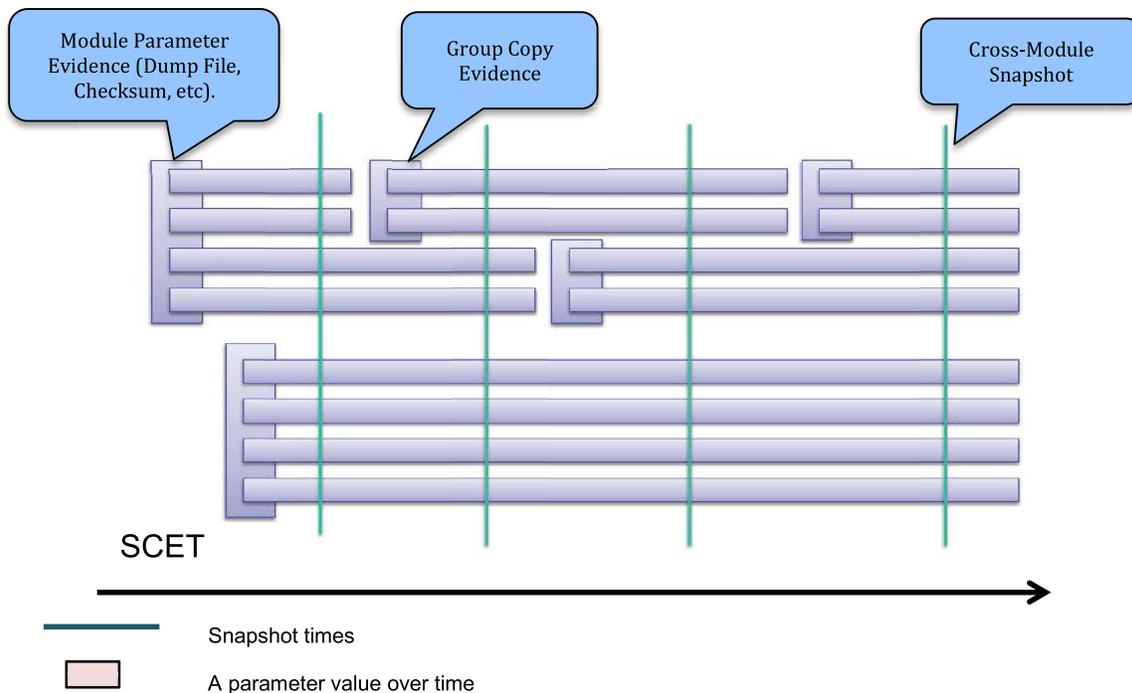


Figure 3: A conceptual view showing how samples of collected evidence arrive in structures of modules and groups, and how snapshots may be taken at various times over the time range of collected data. The reported data in each snapshot needs to be the most recent samples of evidence on or before the snapshot time. As the

above suggest, actual data evidence in a snapshot may be derived from a large number of evidence samples across the whole parameter data set.

A History query of the data above is relatively straightforward to implement; essentially, query all results for parameter changes from time A to time B. However, a query for a Snapshot of data, especially a high performance snapshot over a large range of data, is more complex. To enable a high performance snapshot query, the team designed a “revision” based approach, structured around the format of the parameter dictionary.

With the revision based approach, all the evidence data received at a certain time is associated with a “context revision”, which is itself a type of object saved in the database. The context revision contains all information related to the new evidence, but also information from the previous context revisions as well. The logic works like this:

- When new evidence is received, generate a new context revision
- Generate module, group, group copy and parameter revisions for all new data received and associated with the new context revision
- Query for and relate any module revisions from the previous context revision and associate with the new context revision

The following diagram depicts how the context, module, group, group copy, and parameter revision structure looks:

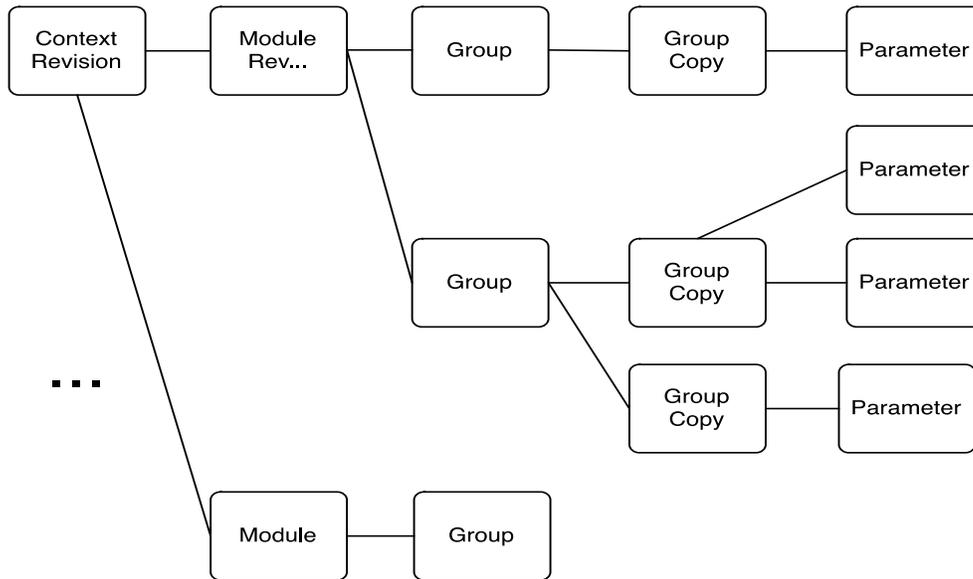


Figure 4: Shows the relationships between a Context Revision and the module, group, group copy and parameter data associated with it. Note that each box represents a revision of the shown type.

As the above diagram shows, a single “context” is related to multiple modules (in fact, *all* of the modules) including the modules related to groups, groups to group copies, and so on, all at one specific time.

To manage the relationships in the database, the Hibernate automatically instantiates “reference tables”, whose sole purpose is to manage the relationships between separate objects. For example, a Context-Modules table provides object relationships between a Context Revision and associated Module Revisions where the columns include the unique ID (typically auto-generated) of the Context Revision and the ID of the associated Module Revisions. The following diagram shows the full set of tables thus generated in support of parameter tracking:

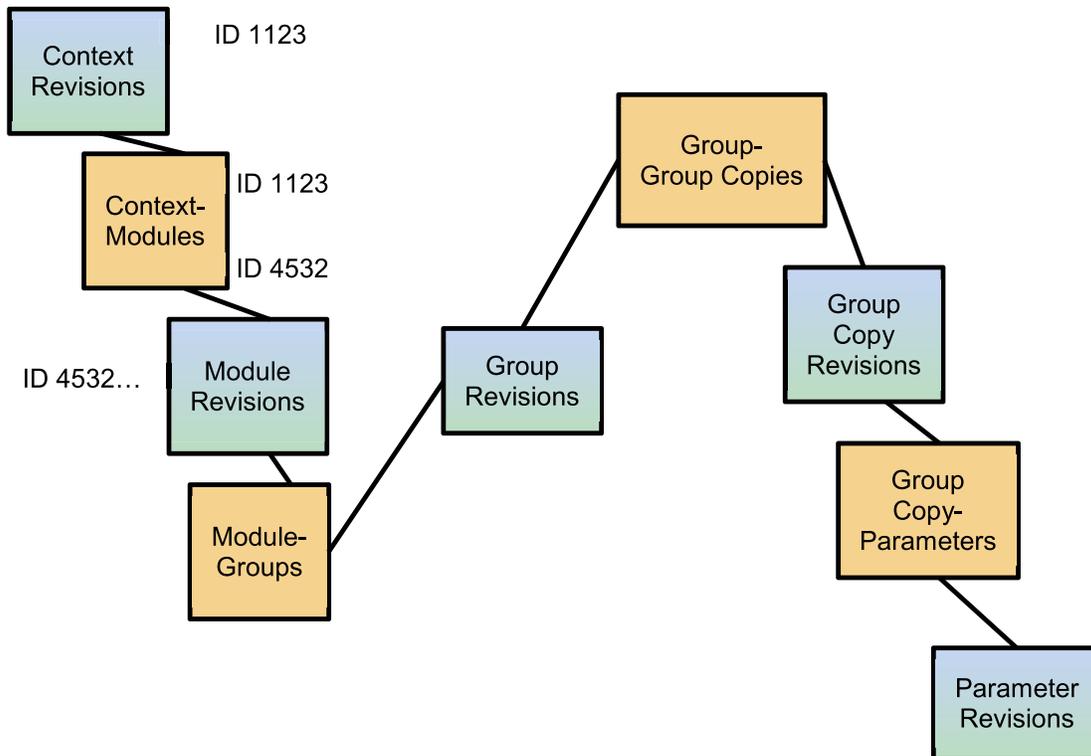


Figure 5: Relationship between the Revision tables in the database. Relationships between revision types (e.g. context and module revisions) are managed in mapping tables, enabling n...n mappings between those objects. Mapping tables are managed automatically by Hibernate according to tags in-line with the object definition code for those data types. A side effect is that tracking a single parameter update requires an update to 10 separate tables in the database.

The following diagram shows an example of typical revision management:

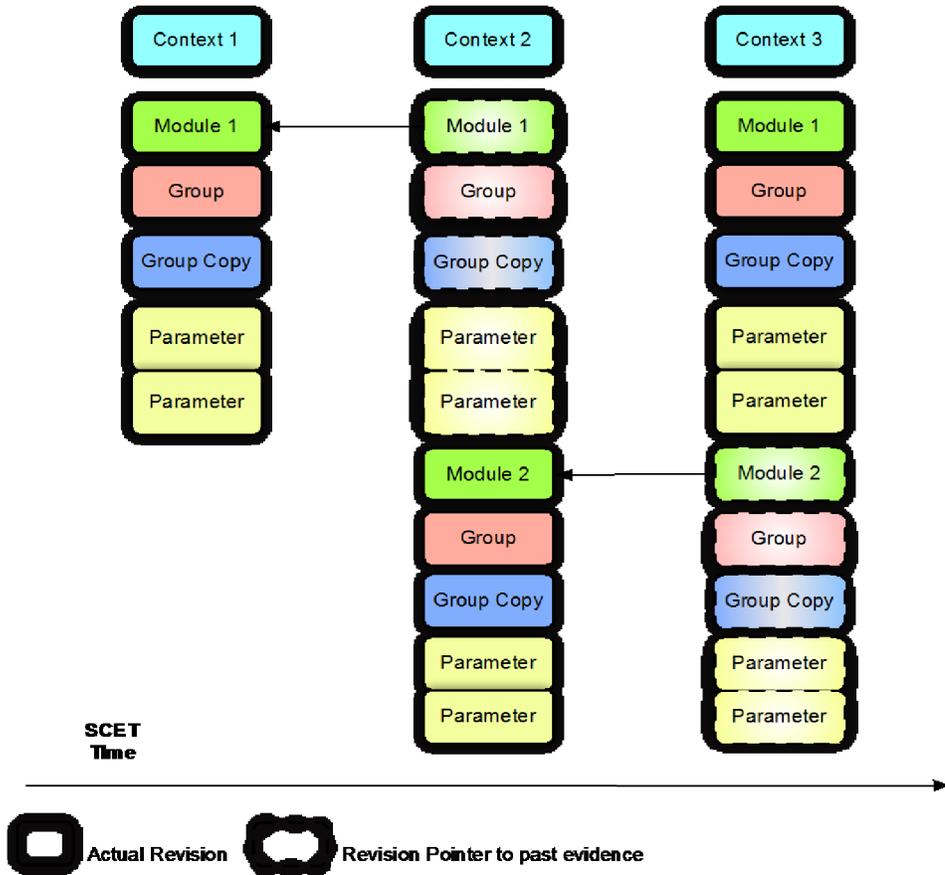


Figure 6: Shows how revisions consist of “actual” revision data as well as pointers back to earlier actual evidence.

The above example shows that evidence is tracked as part of Context 1, including a Module, with its associated Group and Parameters. A Context 2 was created for a Module 2 worth of data, however since there is already a Module 1 in Context 1, that Module 1 revision is associated with Context 2. Furthermore when a Context 3 is provided for Module 1, a back-pointer is created for Module 2 back to the information from Context 2.

The reason for all of this pointer tracking is query performance. With this data architecture, a snapshot query can be executed for any point in time, including between revisions, and all the server needs to do is find the most recent context revision before the snapshot time. That context revision will contain both data from that revision, and a set of pointers back to the most recent context revision for every other available parameter value.

Implementation of the History query was much more straightforward; that was a simple matter of querying for all the context revisions from time A to time B.

The revision architecture enabled the software to meet performance requirements, with snapshot queries taking only a few seconds in the worst case, even for queries of all parameters in the dictionary. However, the architecture proved very problematic in other areas, and issues associated with the revision approach would lead to later unanticipated challenges, as will be discussed in Section VII.

D. Graphical User Interfaces

Besides some straightforward command line tools to POST and GET data to and from the server, some simple web user interfaces were implemented for ease of browsing the data set. The following is a screenshot of the web user “portal” interface for the Snapshot query in particular:

Parameter Management Toolkit

[Parameter History Report - CBM Report&History](#)

Snapshot Report	
SCET	<input type="text" value="now"/>
Venue	<input type="text" value="Surface"/>
RCE	<input type="radio"/> A <input checked="" type="radio"/> B
Persistence	<input checked="" type="radio"/> Memory <input type="radio"/> Saved
Template	<input type="text" value="snapshot.html"/>
Filter	Category <input type="text"/> - OR - FSW Module <input type="text" value="bcb"/> Group <input type="text"/> Group Copy <input type="text"/> Parameter <input type="text"/>
<input type="button" value="Get Snapshot Report"/> <input type="button" value="Reset"/>	

Figure 7: A screen shot of the snapshot portion of the PMT web interface.

The above shows the user ability to specify a snapshot time (e.g. “2014-001T00:00:00” or “now”), select a venue, select an MSL spacecraft side (MSL has 2 separate flight computers, referred to as “sides”), retrieve parameters from Volatile (Memory) or NVM (Saved), specify templates, and also filter by the module, group, group copy and parameter name.

An example (partial) HTML snapshot of the BCB module is included as the following:

Parameter Snapshot Report

Venue: Surface
 RCE: B
 Module: bcb
 SCET: 2014-105T19:12:16 UTC
 Report Time: 2014-105T19:12:16 UTC

Module	Group	Group Copy	Parameter	Value	Units	Last Update	From Evidence	Evidence
bcb	mon	batt_offline_imminent_bcb1	detection_enabled	True		2014-091T03:48:04.435	2 weeks earlier	BcbParms_0449595466-30997-1.dat
bcb	mon	batt_offline_imminent_bcb1	response_enabled	True		2014-091T03:48:04.435	2 weeks earlier	BcbParms_0449595466-30997-1.dat
bcb	mon	batt_offline_imminent_bcb1	error_persistence	1		2014-091T03:48:04.435	2 weeks earlier	BcbParms_0449595466-30997-1.dat
bcb	mon	batt_offline_imminent_bcb1	fault_persistence	1		2014-091T03:48:04.435	2 weeks earlier	BcbParms_0449595466-30997-1.dat
bcb	mon	batt_offline_imminent_bcb2	detection_enabled	True		2014-091T03:48:04.435	2 weeks earlier	BcbParms_0449595466-30997-1.dat
bcb	mon	batt_offline_imminent_bcb2	response_enabled	True		2014-091T03:48:04.435	2 weeks earlier	BcbParms_0449595466-30997-1.dat
bcb	mon	batt_offline_imminent_bcb2	error_persistence	1		2014-091T03:48:04.435	2 weeks earlier	BcbParms_0449595466-30997-1.dat
bcb	mon	batt_offline_imminent_bcb2	fault_persistence	1		2014-091T03:48:04.435	2 weeks earlier	BcbParms_0449595466-30997-1.dat

Figure 8: A screenshot of the BCB Monitor parameters for two group copies.

E. Design Lessons Learned

Overall, the ReSTlet and Hibernate frameworks were strongly complimentary. It was easy to add and modify resources, and by default separate resources are well encapsulated from each other (although it did require the implementation of Abstract class layers to avoid some duplicate functionality). Hibernate demonstrated considerable ability to facilitate on-the-fly database modifications, since the only thing required to modify is to add get methods to a class. Hibernate may even be useful for schema prototyping, even if Java and Hibernate are not the eventual technologies to be used. The framework resulting from the integration of the ReSTlet and Hibernate libraries proved useful as the baseline for other recent mission system prototyping efforts and was used in support of the SMAP PMT version.

In contrast, the revision infrastructure proved problematic in a number of areas. For one, it was particularly complex and time consuming to debug, as it often required “walking the tree” from the parameter table to the context table and back, with a given issue hiding possibly anywhere across nine related tables. Another issue is that the approach requires a large amount of meta-data for each actual parameter data point, such that the database over time grows disproportionately considering the amount of “actual” parameter data being stored. A third issue relates to handling of out-of-order data; this will be discussed in detail in section VII, Supporting MSL Surface.

VI. Supporting MSL Cruise

The first version of the tool that was available just ahead of MSL launch, and following Launch/Cruise Operational Readiness Test (ORTs), was first utilized to operationally track actual cruise data about a month after launch following validation on MSL testbeds. It included a basic set of capabilities to create venues, load dictionaries, collect data from parameter dumps, and provide snapshots and histories via command line and the web tool.

Relative to eventual surface operations, the number of parameter changes was very low during cruise. The approach was also very consistent in that every change to a cruise parameter utilized a “set/save/dump” approach, such that every parameter change was updated in volatile and NVM memory at the same time, and a dump file

produced for the ground PMT to ingest. This approach proved to be robust over the course of cruise, where few issues of note arose.

Through much of the cruise timeframe, the PMT team was diverted back to work on a series of upgrades to the Data Management tool in preparation for the much more complex Surface data management problem, and PMT implementation saw minimal progress over this time through early Surface operations.

VII. Supporting MSL Surface

A. Overview

A new version of the PMT tool was ready ahead of Surface operations and executed over consecutive Surface ORTs. However, it was clear fairly early on that the tool's usability would be fairly limited in the face of actual surface parameter usage, as surface operations brought very different usage characteristics from cruise that the tool could not support.

Parameter changes were much more frequent during the surface mission. Updates to parameters in memory were more common, and in particular, a pattern of "set parameter, set parameter, save parameters" saw use. The exclusive use of Set/Save/Dump during Cruise was much easier to track, but used up more downlink capacity and greater impact on machine "save" resources. However, the PMT version at this time was not capable of tracking any evidence not in a dump. With the "set-set-save" pattern, changes are made to volatile and NVM memory without any dump evidence being downlinked at all.

It was here that the FSW issue with NVM reporting in the module dump became a major issue. Since the module level dump did not contain NVM state, that state would need to be inferred by other means, such as the history of successful parameter commanding. However that version of the tool was not coded to include tracking by anything but module dumps. The unfortunate result was that there were a substantial number of updates to Volatile and NVM memory that this version of the software was unable to track. As a result, the Spacecraft Engineering Operations (EO) teams implemented operational workarounds taking advantage of other MPCS tools. In response, the PMT team began designing updates to include other sources of parameter evidence including sequencing results in particular.

For a subset of modules that used set/save/dump commanding, PMT was able to perform basic tracking functions. However, a few months into surface operations, an operator found an issue with the snapshot tool exposing what would turn out to be a major architectural flaw in the revision approach.

B. Ordering Trouble Found

A few months after landing on Mars, an issue was reported that a snapshot did not include a commanded and dumped parameter change. Upon investigation, the PMT team found that the parameter update had been recorded in the tool, as it was present in the database and the History report, however for some reason it was not seen in the snapshot.

The issue had to do with PMT's revisioning approach. What had happened was that operationally, a "group dump" product had been downlinked out of order from other parameter dump files due to lost data during downlink and subsequent product retransmission. A "context revision" was generated in between two existing sets of data, for which the later context contained pointers back to modules in the earlier context, as in the following diagram.

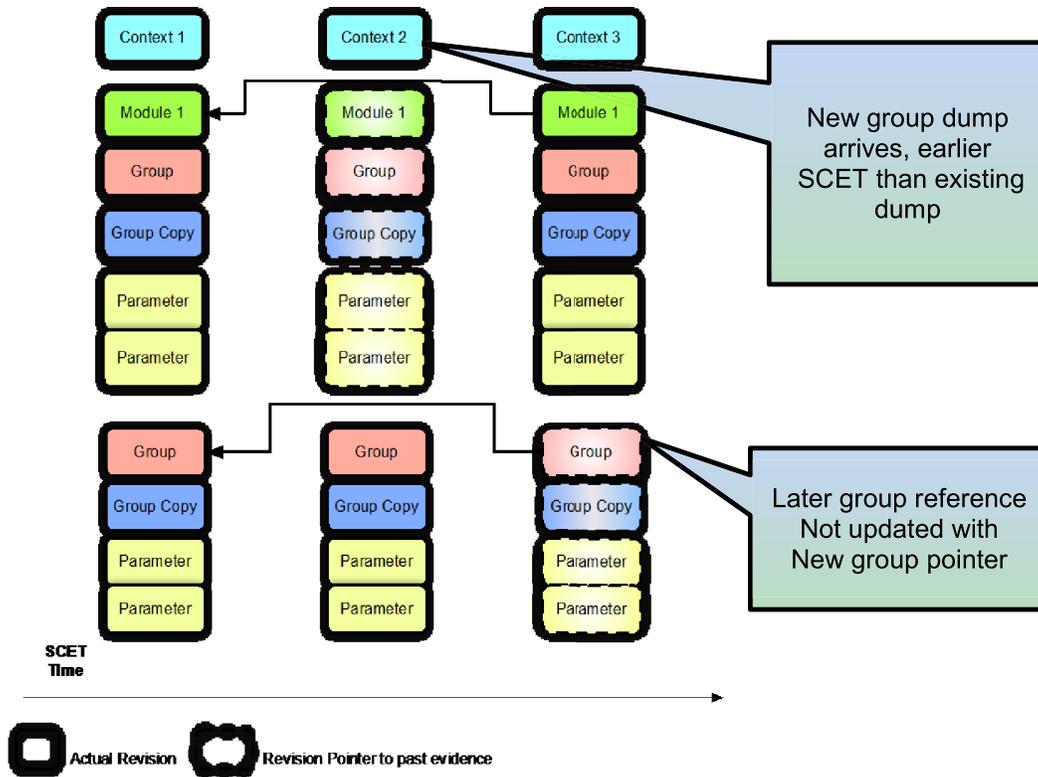


Figure 9: Ordering Issue. This diagram shows how out of order data is missed in the snapshots without fixing the back pointer. In this diagram, Context 2 arrives on the Earth after Context 3.

When the snapshot function is executed for a time later than that of Context 2, the function looks back to find Context 2, and the references Context 2 has to Context 1, and so the Context 3 results are not included in the snapshot.

To address the problem, the PMT team worked out an approach involving “fixing” of module and group references. When a revision is inserted between two other revisions, the pointers in the later reference need to be re-pointed to the new revision. As in the above example, when Context 3 arrives, the group pointer from Context 2 has to be modified to point to the group data in Context 3.

The “fixing” logic works out like the following:

- Query for Context revisions later than the new group revision
- Loop through the context revisions, find any module revisions that match the module for the new group
- For each matching module found, loop through the group revisions and find any matching group. If the SCET of the found group is earlier than the new group, update the module revision with the new group copy

Furthermore, in the original design, the only pointers in place were module pointers. However, since data can arrive at the group/group copy level, the pointer structure needed to be in place at the group and group copy level. This meant that each context revision would need to track a pointer for every module and group/group copy in the dictionary.

The approach would eventually be made to work, but not after several false starts and validation against some more robust testing, costing months of schedule time. As a side effect, the database ballooned with thousands of additional pointer records required for every context in the database.

C. Secondary Evidence: Command Evidence

As mentioned previously, it was identified fairly early during Surface operations that the dump file parameter tracking was not capable of providing the required fidelity of tracking required for operations. The next best source of parameter state evidence was the record of successful execution of a commanded parameter change. Part of that evidence was available in the form of Event Records (EVRs) downlinked as telemetry. As activities are performed onboard the spacecraft, EVRs report events such as wakeup data, changes in spacecraft states, and success or fail of execution of sequences of commands. With this approach, the actual parameter values are derived from the command arguments. A flight software tool was provided to retrieve the full sequence information concerning the parameter command, including all of the necessary command arguments, which are saved as parameter value “samples” in the database. Unfortunately the flight software tool produced output in human readable format as opposed to something more machine-readable, so retrieving the key information like boot times and boot info as well as all of the parameter commands executed required some tricky parsing that took a good deal of time to get right.

The command evidence tracking approach was eventually completed in mid-2013 and demonstrated to provide a level of fidelity for parameter tracking to finally make the tool broadly useful for all subsystems.

D. Comparison with Mechanisms Incon, and Further Identification Issues

Along with the development of secondary evidence tracking, the project recommended validation against some existing EO data sets. This included comparison of the parameter tool against a Mechanisms team tested configuration file, called an “initial conditions” or “incon” file. This file had been kept up to date by the EO team with the “best known” NVM state based off of other (non-PMT) inputs, and represented the best “gold standard” source for parameter knowledge, containing about a third of the overall parameters.

The testing proved that the parameter tool was properly tracking the various sources of evidence, however it exposed several issues with parameter identification. Identifying parameters from the downlink data had always been a challenge, as the parameter name “tagging” in the telemetry products was not consistent with the defining dictionaries, and not consistent across flight software modules, with a number of “special cases” in the transform from the auto-code dictionary to the PMT dictionary. Furthermore, the parameter identifier in the evidence does not map directly to the parameter in the dictionary, making mapping a challenge. For example, as derived from the dictionary a parameter is named “BCB_MON_batt_offline_at_boot_bcb1_detection_enabled”, but might be identified in the downlinked BCB module Group DMP product as “batt_offline_at_boot_bcb1_detection_enabled_dat”. The result is that the mapping required some level of substring matching to map the two together.

The comparison against the Mechanisms initial conditions file showed that not all of the parameter samples that were being reported in the incon were present in the Snapshot report. This was a result of a failure in the substring matching in some cases, where more than one parameter sample was being mapped to a dictionary parameter – meaning that some parameters ended up with multiple values with all but 1 being incorrect, and other parameters with no value at all. The PMT team eventually identified a correct set of patterns to map parameter values to the proper dictionary definitions, however at substantial additional schedule cost.

VIII. Next Steps

As of the time of the writing of this paper, a version of the PMT is being deployed to operations supporting the full range of secondary evidence tracking, and includes fixes to the ordering and identification issues found in earlier phases. Looking ahead, the next phase of work will include updates in support of parameter “truth”. Basically, knowing whether or not any given set of parameter values are valid or not. This work will take into account the parameter checksums and counters as described in section IV, to validate sampled values and provide feedback to the end-user that measures need to be taken to fix known parameter state.

IX. Conclusion

Managing flight software parameters is a seemingly simple task. But as flight systems increase in complexity, so too does the job of ingesting, tracking, storing and retrieving parameter state knowledge with less than optimal parameter state reporting mechanisms. On MSL, we now have a workable parameter management system that

provides reliable parameter state information for now or any time in the past, despite the challenges we have had to overcome. These challenges have included not being able to know the actual NVM parameter state due to lack of a dump capability, significantly less frequent parameter dumps during the surface mission (requiring secondary evidence gathering techniques), an explosion of the number of parameters to track relative to past missions, MSL's overly complex parameter naming convention due to the increased use of flight software auto coding, context revisions received out of sequence, and many more.

The most important lesson that has been learned from development of the PMT has been the increased importance of a firm understanding between flight and ground engineers on the approach and definition of flight system parameters. This understanding needs to be gained early in the mission, and not several months before launch, as was the case with MSL. A strictly enforced parameter dictionary, with unique identifiers, and strict adherence to the dictionary across all modules will greatly reduce the complexity and increase the utility of future mission's parameter management solutions.

Acknowledgments

The author would like to acknowledge present and former JPL software developers David Noble, Paul Wang, Nicole Ameche, Hayk Arutyunyan, Pearl Haw and Frank Hy for support and contributions to the effort.

The work described in this paper was carried out at the Jet Propulsion Laboratory (JPL), California Institute of Technology (Caltech), under a contract with the National Aeronautics and Space Administration (NASA).

*If you want to go fast, go apart
If you want to go far, go together
- African proverb*

References

¹ "Mars Global Surveyor (MGS) Loss of Contact" NASA lesson learned, URL: <http://llis.nasa.gov/lesson/1805>, 13 April 2007.