Fishing in the Telemetry Ocean

Rob Foweraker[1] and Neal Nelson[2]
MakaluMedia GmbH, Robert-Bosch Strasse 7, 64293 Darmstadt, Germany

Colin Rogers[3]
75 rue du Javelot, 75013 Paris

In this paper we describe the development of a telemetry data analysis system for use by spacecraft operations engineers on the fleet of telecommunication satellites operated by Eutelsat. This system is called TeleChecks. We discuss various aspects of the design and implementation that we consider important to the success of the project. In the first section we have a bit of fun elaborating on our preferred metaphor of 'Fishing in the Telemetry Ocean', rather than data mining. In the second section we describe in some detail the Python language framework that we developed for our data analysis scripts. In the third section we briefly look at the system architecture. We finish with some lessons learned and some future plans.

I. Introduction

A. A Flood of Data

AS we all know, satellite Mission Control Systems (MCS) produce a lot of engineering data. The amount of data increases every day like a *flood* that never subsides. Telemetry, Telecommands, Out of Limits, Events, Schedules, Logbook data, etc. Satellites are getting bigger and payloads more complex. A modern telecommunication satellite can have in excess of 50000 individual telemetry parameters. A fleet of telecommunication satellites can produce several hundred million data points a day. In contrast, the size of the engineering teams that control a typical satellite are definitely not growing at the same rate as the data that is produced.

Of course, most of this data is of no interest and will never be looked. But, in hidden beneath the *surface* is some very important information. *We think that finding that important information is like finding fish in the ocean. We need to know what kinds of fish we are looking for, where they can be found and then have a big net to catch them.* To make things harder, data from different sources is often not accessible easily in a common location or common format that can be processed by a single application.

This *ocean* of data is not a problem that is unique to the satellite operations domain. There are many other domains that have even greater amounts of data. We can learn much from these domains, but ultimately we need to tailor solutions to the specific requirements and working practices of the satellite operations industry. Let's face it, data archiving and data analysis are not the reasons we launch satellites. We store and analyse data because it might prolong the life of the satellite, improve the service offered to customers or help design the next generation of satellites. Compare this with the financial industry where data analysis algorithms can make lots of money, or with social networking websites where (like it or not) analysis of user data is seen as the key to inflated valuations. In the space industry, we want the great tools but we don't have the luxury of large amounts of money to invest in them. We are also competing against the financial and web industries to attract the best talent.

B. Looking for Fish

If we think of our offline archive as a big *ocean* of data then how can we picture our Mission Control System? If we stop and think about it, the individual satellites are like *streams* or *rivers* of data flowing through the MCS and ending up in our *ocean*. Real-time monitoring within the MCS is like a *net across the river* and we can think of the interesting events we are searching for as the *fish*. We can easily catch lots of big *fish* (e.g. Out of Limits), but the small ones slip through the *net*. Why is this? The MCS looks at data at a single instant or over a short time span so

[1] Head of Software Development.
[2] Senior Software Engineer
[3] Satellite Operations Engineer, now retired from Eutelsat SA.

patterns that occur over longer periods are not easily detected. Sometimes we require a large number of parameters and a complex algorithm to detect a particularly interesting *fish*, therefore, in the case of a minor problem, it is not worth the effort of implementing this in the MCS. We normally don't have the ability to add specific monitoring for every small problem directly in the MCS as it must be a tightly controlled environment. Lastly, we do not actually *want* to detect all problems on the MCS as this would potentially raise too many alarms and distract the controllers. Therefore, in addition to real-time monitoring we need *offline* monitoring within the data analysis function.

Ultimately the data flows into our offline data archives and is stored there. It is in the 'Offline Data Ocean' that we have more flexibility for ad-hoc offline monitoring and data analysis.

What kind of analyses do the satellite operations engineers want to perform?

1. Trend Analysis

We might be interested in the value of a certain TM parameter over a long period of time, but more often, the value we are interested in must be calculated using two or more TM parameters (potentially many more). Examples include the difference between the daily minimum and maximum values of a parameter or the number of anode spikes detected on a TWTA. Alternatively the values we are interested in might only be available at a specific time, for example at a specific point during a manoeuvre or an eclipse cycle.

2. Health/Performance Monitoring

There may be a need to perform certain health monitoring procedures regularly. Instead of getting a controller or engineer to perform these procedures manually at the MCS console, they can be automated as an offline data analysis task.

3. Event Detection

An expected event might be a manoeuvre or an eclipse. In the case that one of these events are detected some extra analysis might be required, for example to record the temperature profile of a thruster before and after the manoeuvre or to look at the depth of discharge or battery temperature variations during an eclipse.

4. Anomaly Detection

When an anomaly has been observed and analysed, it becomes possible to implement the same methods to detect future events and generate a detailed report automatically. *Once we know what a fish looks like and where it lives, we can catch them with ease*. It is also possible to apply retrospectively new analysis methods to previous telemetry.

5. Report Generation

We can also use this system to collect data that is required for reports. This analysis might be required on a monthly or yearly basis.

C. Some thoughts about the Fishermen

Our *fishermen* are satellite operations engineers and analysts. They are rarely trained software engineers, statisticians or data scientists. They are, of course, experts on the satellite. They know the problems they are looking for and usually where to find them in the data.

Analysing the data is only a part-time job for a typical satellite operations engineer. The development of data analysis scripts must be as quick and easy as possible and it must be possible to schedule them for regularly non-attended execution. The language must be easy to learn and use yet powerful.

This is a key requirement of the system. We want the system to allow more data analysis to be done with the same sized team of engineers and not to create more work.

D. Fishing Boats

Excel© and VisualBasic© are widely used by satellite operations engineers. It is interesting to consider why this is the case. Excel is usually already installed and easy to use. Engineers can get hold of some data in a format such as comma separated values (CSV), load it into Excel, visualise it, massage it a little and either plot it or write a few macros to analyse it. Excel is a reasonably quick way of doing some ad-hoc *playing* around with data. We think that 'playing' is the key word.

The problems come later because Excel + VB doesn't work well as an operational system. Nothing is optimized for reusability, performance or efficiency. Engineers can share files but there is no central repository or version control. At first using Excel may be efficient, but in the long run it won't scale.

What are the features of a system that encourages engineers to play with the data? After some thought, we came up with the following shortlist.

- An intuitive high-level language with a clear syntax
- Be able to visualise the input data and output results, when required
- Make fetching and using large data sets as easy as possible
- Provide good feedback on script errors and good debugging tools

- It should not be possible to break the system or overwrite any important existing data by accident
- A developer should be able to see what other engineers have developed and be able to copy it
- There should be good user documentation with plenty of useful examples

E. Building a better Ocean

In order to support efficient data analysis a good data archive needs to have certain features.

- We need a high performance archive - retrieving historical samples for a single parameter over a long time period must be very fast. It should be possible to retrieve a day of data for a parameter changing once per second in less than one second.
- There **will** be gaps in the archive – these gaps must be clearly marked in any retrieved data so that data analysis algorithms can handle them.
- There **will** be invalid data and this must be available but clearly marked. The validity of the data is an opinion and the MCS and the engineers can have different opinions. The satellite operations engineers need to know whether the MCS thinks the data is invalid or not, but they must have access to all data.
- The archive should either store or be able to calculate statistical data. Quite often a script may only be interested in the minimum, maximum or average daily value and in this case it would be inefficient to fetch all values from the archive and calculate the required statistics in the data analysis system.

We had already developed a suitable telemetry data archive for Eutelsat, as part of a previous development. This archive is called TeleViews.

II. A Framework for Analysing Telemetry Data

The TeleChecks system we describe in this paper grew from an in-house Eutelsat system developed over the last 10 years by Colin Rogers. This system ran FORTRAN code and fetched telemetry data derived from a raw frame archive. Eutelsat tasked MakaluMedia with re-developing the basic ideas into an operational system that could be used by all the operations engineers rather than one or two experts. As part of the development work, most of the existing FORTRAN scripts had to be converted to work in the new system.

It was very important to select the right language for the script development. We had the following requirements:

- A high-level, open source, scripting language with no compilation required.
- A clear, explicit syntax that would be easy for engineers to learn and use
- A readable language, to make maintenance easier. Of course, this does not ensure that another engineer can take over the maintenance of a script, but it helps.
- A language with good scientific computing libraries
- An Open Source language with a suitable license

We selected Python. A major reason for choosing Python was the existence of the open source NumPy/SciPy libraries that contain many mathematical and scientific methods. The combination of Python and SciPy is somewhat equivalent to MATLAB©, a tool that many engineers now use during their education.

Once we had selected a language, we still had the task of working out how to use the features of Python to develop scripts for analysing telemetry. Key features are:

- Automatic initialisation of satellite specific and task specific data
- A model of the satellite environment (sun direction, eclipse status, etc)
- A simple interface to fetch, store and manipulate data from the archive
- Sensible handling of data gaps and invalid data samples
- Generation and storage of computed results and events

We wanted to be in line with the principles of the Python language (be 'pythonic') and have a clear simple usage. This took time and several iterations.

The Python scripts are executed in a specially constructed 'Execution Environment'. This is a constrained version of the standard Python global environment with some important functionality removed. This creates a sandboxed environment, so that scripts may execute without the ability to affect other scripts or the operation of the system as a whole. For example, we removed the all file access, print, standard input and system call functionality. We automatically include the following Python modules:

- **dates**, which we enhanced with some specific date conversion
- **datetime,** a standard Python module
- **SciPy**, which provides extensive mathematical functionality in addition to that built into the system.

In the system, a script is always executed for a specific satellite and a specific time period, we refer to a specific execution as a *task*. When a script is run, the execution environment will automatically include a satellite specific initialisation module. This module can be used to define certain satellite specific constants, for example the telemetry frame rate, the satellite ID code, the telemetry parameter that records the current longitude etc. This module can be modified within the system because some of this information could change. The initialisation is automatic, the script developer does not have to add any specific code. At the end of the initialisation information related to the current task and the current satellite will be available to the developer through the **TaskInfo** and **SatelliteInfo** objects respectively. These Table 1 summarises these global values.

Global Value	Description	Return Type
day	Task Execution Day (days since 2000-01-01)	*int*
task.begin_time	Start of task execution period	*date object*
task.end_time	End of task execution period	*date object*
task.use_invalid	Use invalid data	*boolean*
task.version	Version of script	*int*
sat.name	Satellite name code	*string*
sat.family	Satellite Family code	*string*
sat.frame_counter	Satellite Frame Counter name	*string*

Table 1 : *Global Values available to developers*

The initialisation module also sets up an Environment Model that is accessed through the SatelliteInfo object. The environment model implements methods that a developer can call to determine the sun azimuth or elevation at a particular time, the solar radiation at a particular time or determine whether the satellite is in eclipse by either the earth or the moon at a particular time. There is obviously great value in making these calculations extremely easy. The developer does not have to consider the current satellite longitude or time during the year as these are automatically taken into consideration. It should be mentioned that for the purposes of data analysis we did not need to implement a very accurate environment model, knowing the eclipse entry time to within 30s or the direction of the sun to within half a degree was sufficient. The available methods are summarised in Table 2.

Method	Description	Return Type
sat.longitude	Satellite Longitude (on task execution day)	*degrees East*
sat.sunAzim(t)	Sun Azimuth at time t	*radians*
sat.sunEl(t)	Sun Elevation at time t	*radians*
sat.sunRad(t)	Solar Radiation at time t	*float* (W/m^2)
sat.inEclipse(t)	Overall Eclipse factor at time t	*integer*
sat.eclipses(t1,t2)	List of eclipses between times t1 and t2	*string*

Table 2 : *Environment Model methods available to developers*

In order for the scripts to perform any useful work, facilities are provided for the retrieval and manipulation of telemetry data as well as the production of various kinds of results. We implemented two different objects to fetch, store and manipulate of telemetry data.

- The **TMData** object
- The **TMStats** object

The TMData object fetches all the telemetry samples between two dates. This is required when a developer needs to work with all the sample values. The TMStats object only fetches statistical data (minimum, maximum and average value). Using the TMStats object can be faster when there are many samples during a time period and only statistical data is required. We will discuss the TMData object in more detail as it is much more interesting.

It was important that the users do not have to concern themselves with how or where the data is archived. In other words we wanted an abstract interface to the archive. We create a telemetry object supplying the telemetry mnemonic and a start and end time to fetch data for. These objects can be created as required in the script. In order to fetch some TM from the archive into a TM object, the developer just needs to know the parameter mnemonic. By default TM data samples will be fetched for the complete execution day. The <start_time> and <end_time> are optional arguments that allow us to fetch data for any time relative to the execution day.

```
tm_data = TM("a1234", <start_time>, <end_time>)
```

This creates a TMData object. One important feature is that the data is not actually fetched from the archive until the telemetry object is actually used. Fetching data from the archive is time consuming. For example fetching a day of samples for a parameter that changes twice a second will result in around 170,000 samples being returned from the archive, which can take a few seconds. By delaying fetching the data, we can declare all the objects we think we need at the top of the script but we don't have to be concerned that not using an object will adversely affect the overall execution time of the script. Furthermore a script may be written to work with all satellites of a certain platform, and there can be differences in the parameter names.

Our TMData object will contain all the samples found in the archive for the requested time period. For each sample we have the raw and engineering values, the validity status and the out of limit status available. The samples values are stored as an IndexedArray type. Now that we have the data, we can play with it. Times are always in seconds relative to the start of the execution day, so that scripts can be run on any day.

`tm_data.raw[t]`	the raw value at time t1
`tm_data.cal[t1:t2]`	the calibrated (engineering) values between t1 and t2
`tm_data.ool[t3]`	the ool status at t3
`tm_data.gaps`	List of TM gaps (start and end times)
`tm_data.cal[t2:t3].average()`	Calculate the time-weighted average value between t2 and t3

Table 3 : Examples of TMData object manipulation

Consider the operation to get the value of a parameter at a certain time. The telemetry consists of discrete samples sent at intervals. The value at a certain time will be the value that the parameter had at the previous sample time. The developer doesn't have to worry about this when developing the script.

There are currently two methods for outputting results from running scripts.

- Event Messages
- Daily Statistics

Event messages are typically used for alerting the operations engineers when a certain event or anomaly is detected. An example of an event might be a manoeuvre or a payload configuration change, while an example of an anomaly might be a bus voltage spike. An event message can contain one or more plots and any amount of text.

Daily statistics are used to store calculated values for long-term trend analysis. Examples might be the minimum and maximum temperatures of an equipment or the total available power from the solar arrays. The user interface for the web application can display messages and daily statistics output by the scripts.

A common pattern for a data analysis script is to loop through time and use one or more telemetry values to try and detect an event and/or calculate a result. It is possible to iterate through the samples stored in the TMData object directly. This is very efficient as samples are change-only and therefore there is not a constant interval between each sample. We also added a method to merge the sample times for two or more different TMData objects, so that it is possible to iterate through all samples for multiple parameters.

The language also supports vector operations on TMData objects. We thought that this would be a really powerful feature, and, in theory, it is. For example, it is possible to add or subtract TMData objects and the results are as you would expect, a new TMData object with the expected result at each sample time .This even works when the sample times of the TMData objects are different. The framework also supports comparison operators, so it is possible to find a list of times where one parameter is larger than another value. The problem is that it in practice it is not a very intuitive approach and therefore makes the code less readable and therefore maintainable. We do still use vector calculations, but not as much as we thought we would.

We developed a number of utility methods to implement some common algorithms efficiently. These methods are available in a module that can be shared among scripts. For example, we created methods to detect spikes in data, for example an anode voltage spike. Another example was a method to detect changes in switch positions or equipment statuses while also taking into consideration gaps in the TM data or periods of invalid data.

The spacecraft operations engineers were concerned with having to learn Python, which for most was a 'new' programming language. This would have been the case for any language we selected. In practice, only a small subset of the Python language is required to develop a script within the system. A successful project has to have the acceptance of its users; therefore we needed to address these concerns. We tried to provide extensive examples of how to perform common data analysis tasks through a cookbook type user manual. We have found from experience that good documentation together with a few hours of hands on training is sufficient to get started on script development. All developers can see the all scripts written by us before deployment and the other developers. This is a good resource when learning how to write scripts.

A note on source comments in the scripts. The choice of Python was meant to make the scripts more readable and therefore more maintainable, but this can obviously not be guaranteed. *Use of specific engineering knowledge should always be explained in a comment*. You can generally work out what the script is trying to do, but probably not why an engineer chose to do it that way – which may be very important.

A short example script is included in Appendix A.

III. System Architecture

A. System Architecture

As mentioned earlier, we think it is very important that the engineers feel that they can play with the data without fear of deleting anything important or affecting the execution of the scheduled scripts. For this reason, the final system consists of two separate environments that share exactly the same codebase. The environments run as two separate web applications on the same physical server.

The first instance we called the *Development Environment*. This is the 'playground'. The engineers are free to create scripts, run them and look at the data that is produced. There is no implicit guarantee that any results will be maintained forever or that another engineer will not modify your scripts. We implemented a version control system for the scripts, so that any old versions can be recovered should this be required. In practice, engineers tend to only modify their own scripts and as the teams for each satellite family are quite small chaos is avoided.

The second instance is called the *Production Environment*. This is much more strictly controlled. There is a mechanism to move scripts from the Development to the Production environment. Only scripts that have been run successfully for a certain number of days can be transferred to the Production Environment. Once a script has been transferred it is possible to schedule it for daily execution. All data generated by running a script will be kept until the script is re-execute for the same day, for example in the case that a bug in the script is corrected. All the old historical result generated by the previous system have been imported into the Production Environment so that they are available to the satellite engineers when they want to look at the long term trends.

B. User Interface and Usability

All interaction with the system is through a web browser interface. This simplifies installation, deployment & updates and avoids any need to install a Python programming environment on user workstations. This last point is very important from the security point of view and reduces system administration tasks too. The system can also be used from a computer under any Operating System with a suitable web browser. It is also easy to monitor access to system and the system can be made available remotely (offsite) should this be desired.

The editor is an open source javascript plugin called ACE which is actually very easy to add to the system. Achieving a good user experience in editing, executing and debugging is much more challenging. Developing scripts can easily become a time consuming task so usability is a key driver of overall satisfaction with the application. We introduced an execution log to allow easy tracing of the script execution, with a nice linkage from the error messages to the scripts. This makes script development within the browser a much better user experience.

C. Security

1. Data Access Security

We have a dilemma. On one side we must have a secure system and protect access to the sensitive data. On the other side we actually need people to be able to access the data. *Any* point of access to the data is a potential security risk, but you *do* need access. It is better to have a single, secure, well-monitored point of access for all important data rather than many separate points of access. This allows us to concentrate our security resources in one place.

2. Script Security

We are running scripts on a central server. Python is a general-purpose language and scripts written in Python could potentially do 'naughty' things like access the local network, delete files or run out of control (by accident or design). We handle these concerns with three features. Firstly, scripts are run in a virtual machine sandbox that only has access to the main database web application database. A script cannot access other machines on the network, nor the physical server disk. Secondly, we removed all potentially dangerous python features like I/O, network access, print statement from the environment. Thirdly, scripts that run for longer than a predetermined timeout are stopped by the application.

The virtual machines are restored from an image at each shift change (see below) and this is an additional security feature. If a script managed to modify the execution environment then these changes would be removed at the next shift change. The virtual machine is simply a container for executing the scripts, all the important data, such as the scripts and the outputs that they produce are stored in the main application database.

D. Scheduling and Resources

As mentioned before we have two environments, Development and Production. We also have limited resources for script execution (principally CPU time) and therefore we need to manage our resources carefully. We implemented a shift pattern. During the day more of VMs are dedicated to the development environment to support development and testing. During the night all the VMs are allocated to production environment, so that all the scheduled tasks can complete before the engineers arrive at work. This approach ensures that playing with scripts in the development environment cannot affect the important scheduled tasks in the operational environment.

E. Scalabilty

Running the scripts in Virtual Machines makes the system scalable. We can run multiple VMs in parallel on a single physical server as these servers have multiple cores. The scheduler is also able to manage VMs on other physical servers. We can add enough servers as required in order to complete all the required scheduled tasks in the allotted time. This ability to scale is very important in a fleet environment. Several new satellites are added each year and new scripts are created, so the overall load on the data analysis system will grow over time.

IV. Lessons Learned

This was a challenging development. We knew what we wanted to create, but it was very hard to imagine exactly how the finished system would look. Usability is the key to success, but we had trouble imagining how we would use it. In these kinds of developments, time is more important than the number of people working on it. You head down the wrong path and have to backtrack. You need to take an agile approach with a small development team.

We spent a lot of time on the details of our language. We knew quite early we wanted to have a TM object, with convenience methods to work with the data. The usage of the language had to seem natural, obvious, (pythonic). It was important that the language became stable a long time before the infrastructure of the system, so we concentrated on this at the start. Changes to the user interface are relatively cheap and easy, changes to the language have a potentially big effect due to the need to re-write scripts and re-validate them.

The concept of a 'playground' for developing scripts is important but there is also a need for a production environment with rules. Development 'Playground' = no Rules, Production = Rules. The output data in the production environment is important and can be trusted. The developer can play around in the development environment without fear of overwriting important production data. How do you make this easy but safe? We ended up with two separate (near identical) environments with a mechanism to transfer scripts from one to the other. Scripts can only be transferred once they have run successfully.

V. The Future

Python is very nice but it is relatively slow compared to Java, C or Fortran. For example, as mentioned before we have a very high performance telemetry data archive. When we request data from the archive we can get 100,000 samples per second in JSON format in the response. After parsing this into a TMData object we end up at about 30000 samples/sec. This is somewhat mitigated by running many tasks in parallel, but more speed would improve the system. We are excited by the current developments in the PyPy area that promise to make Python code as fast as C code.

We would like to make script execution scheduling smarter. Currently, there is no restriction on how many tasks a developer can try to launch at any time. Therefore, it is possible for a developer to monopolise all the system resources for a few hours or more. This can block other users from running scripts. This is a hard problem to solve but we have some ideas.

We would like to add other data sources to the framework and the architecture has been designed from the start with this in mind. We will start by adding access to Telecommands and MCS system events. Other potential data sources that might be exploitable are schedules, electronic logs and payload status data, all of which are available in other systems with in the MCs. We want TeleChecks to be a data analysis *hub*, where scripts can access and work with all important engineering data produced by the satellite fleet.

Appendix A

An Example TeleChecks Data Analysis Script

```
# Fetch some TM.
p0001 = TM("P0001")

# list to store parameter change times
change_times = []

# Loop through the sample times and raw values for p0001
for time, value in p0001.cal:
    if p0001.available(time):
        change_times.append(time)

# Output the change times and values to the execution log
for time in change_times:
    logger.debug("{0}: P0001={1}".format(time, p0001.cal[time]))
    # output sun elevation at time of change
    logger.debug("sun elevation = {0}".format(sat.sunEl(time)))

# output result for number of changes found today
result("changes", "P0001", len(change_times))


# fetch acquisition mode TM (satellite dependent)
if sat.name == "GEO":
    acqmode = TM("a2516", 0.0, 86400.0)
else:
    acqmode = TM("a2616", 0.0, 86400.0)

# Loop through sample times
for time in acqmode.times:
    # Only consider valid samples & exclude TM gaps.
    if time >= 0.0:
        # Check for Acq Mode, when the raw a2516 value of 1.
        if acqmode.raw[time] == 1:
            # Output an event message.
            message(
                "acqmode",
                """
                {acqmode}->TRUE
                Acquisition mode entered at {t0}.
                """,
                t0=dates.secs2hms(time),
                acqmode=acqmode.name.upper(),
                interval=86400.0,
                timestamp=time
            )

# Compute daily statistics and store as results
t100k = TM("T100K")
result("SomeTemp", "max", t100k.cal.max())
result("SomeTemp", "min", t100k.cal.min())
result("SomeTemp", "avg", t100k.cal.average())
```

Appendix B
Acronym List

AND	AlphaNumeric Display
API	Application Programming Interface
CSV	Comma Separated Values
HTML	HyperText Markup Language
HTTP	HyperText Transport Protocol
JSON	JavaScript Object Notation – a lightweight data-interchange format (compared to XML)
MCS	Mission Control System
OOL	Out Of Limits
TC	TeleCommand
TM	TeleMetry
TWTA	Travelling Wave Tube Amplifier
UI	User Interface
URL	Universal Resource Locator
VM	Virtual Machine