



Test verification and anomaly detection through configurable telemetry scanning

Alan S. Mazer

Instrument Flight Software Group
Instruments Division
Jet Propulsion Laboratory, California Institute of Technology

E-mail: alan@judy.jpl.nasa.gov

- Despite hundreds of hours of testing (or more), flight software still launches with undiscovered errors
- By launch, software has passed through many hands
 - Developers
 - Peer reviewers
 - Integration and test (I&T)
 - ATLO pre-launch testing
- Sometimes, if not often, anomalous behavior is captured in test data unnoticed
 - GALEX
 - MICAS camera (Deep Space 1)

The realities of software testing

- Time constraints
 - Sometimes we barely have enough time to write the software
- Software developers aren't suited to testing
 - Testing is tedious
 - Engineers are limited by their "creator" perspective
- Independent testing is a thankless job
 - Learning curve costs time and money
 - Find problems and people are upset; don't find problems and people wonder why you're paid

Why aren't problems found during development?

- Time constraints
 - System I&T is usually pressed by schedule
- Errors may present subtly
 - Small telemetry oddity may reflect larger problem
- Cost constraints
 - Expertise to recognize software errors is not always present
- Trust
 - Test teams rely on developer testing, prioritizing software checkout below other pressing issues
 - Software problems can always be fixed “later”

Why aren't problems found during instrument I&T?

- “Human factors”
 - People get tired and make mistakes
 - Testers may not want to question what they’re seeing
 - People following procedures focus on following the steps rather than thinking about what they’re seeing
- Late changes
 - Without regression tests, late changes introduce risk as new requirements are implemented by developers who have already moved to other projects and forgotten the code

And...

- Phase B/C (pre-I&T)
 - Define scriptable tests to exercise code
 - Provide visibility into software operation through (perhaps optional) telemetry
 - Verify telemetry to determine whether or not test passed
- Phase D (I&T, ATLO)
 - With system engineering, create validity rules for all telemetry points, capturing expertise and determining which anomalies are reportable
 - Verify all test telemetry against rules

What can we do about this?

- Detailed telemetry verification is not well supported by common tools
- One approach to verifying a test is to compare test telemetry to previous runs
 - Simple
 - Works only if telemetry outputs don't vary from run to run (e.g., due to harmless timing variations)
- Another is to use Unix *expect* (a selective *diff*) to verify critical outputs
 - Can ignore innocuous variations in telemetry
 - But...
 - All telemetry must be converted to ASCII
 - Repetitive goals are tedious to set up
 - Doesn't support all-telemetry checks

Verifying telemetry is still hard

- Decided to create a rule-based parser, HKCheck, based on ASCII user-authored configuration files
 - Post-processes binary data streams
 - “Protocol” spec describes packet/message format(s)
 - “Test” spec describes constraints on each telemetry point, and user goals to be satisfied by a particular test
- Supports phase B/C test verification by checking for test goals in telemetry
 - A goal might be an intended error or receipt of a particular command
- Supports phase B/C/D by scanning telemetry and calling out unexpected values

Wrote HKCheck to parse telemetry

- “Protocol” spec
 - Supports heterogeneous packet streams, matched to packet definitions at run-time based on packet contents
 - For example, engineering and science packets in a common stream
 - Packets may be variable-length
 - Provides about a dozen built-in data types
 - Integer, floating- and fixed-point values
 - Various time types, with a variety of epochs
 - Several byte orderings
 - Allows user-defined constants and data types, and arrays
 - Display formats are specific to each telemetry point

“Protocol” defines packet formats

```
consttable packetType = {  
    NOMINAL = 0  
    DUMP = 1  
}
```

User-defined constants

```
packet sciencePacket = {  
    uint8:packetType    PacketType  
    uint16:dec          PacketNumber  
    uint8:hex          Status  
    time4s4ss:date     SpacecraftTime  
    uint8:hex          ScienceData[200]  
} if (PacketType == NOMINAL)
```

Packet def

```
packet dumpPacket = {  
    uint8:packetType    PacketType  
    uint8:dec          DumpLength  
    uint8:hex          DumpData[DumpLength]  
} if (PacketType == DUMP)
```

Packet def

- Each packet def lists a sequence of telemetry points contained in that packet type.
- Each telemetry point has a data type (e.g., uint8), a display format (e.g., date, hex), and a name

Simple Protocol Definition

```
datatype error = {  
    uint8:errorID      errorID  
    uint8:hex          details[5]  
    time4s2ss:date     errorTime  
}  
  
datatype downloadCommand = {  
    uint16:hex  memoryAddr  
    uint16:dec  bytecount  
}
```

“Error” data defines structure of single telemetry point for display

- User-defined data types allow multiple telemetry points to be grouped as one
- Reduces complexity of packet definitions
- Simplifies output displays (e.g., error description is one line rather than 3)

Simple User-defined Types

```
subpacket status = {
  uint16:dec    PktCnt
  uint8:hex     FswVer
  uint8:hex     ScienceVer
  uint8:hex     SensorVer
  uint16:hex    Status
  uint8:hex     Mode
  time4s2ss:dec SCTime
  uint16:hex    CRC
  uint8:dec     Resets
  uint8:dec     TimesMiss
  uint16:dec    CmdsRcvd
  uint16:dec    CmdsExec
  uint16:dec    CmdsRejected
  mwrMessage   LastMsg
  mwrError     LastErr
  uint16:dec    ErrorCount
}
```

“Status” subpacket groups status items which appear in both science and engineering packet formats

- Subpackets group related telemetry items for inclusion across multiple packet definitions

Subpackets

```
set byteorder=msb4thin8
```

```
subpacket header = {  
  uint16:hex  
  uint16:dec  
  uint16:dec  
  uint16:hex  
  uint16:hex  
}
```

```
syncWord  
msgID  
wordCount  
flags  
checksum
```

Middle-endian byte ordering specified

All packet formats include common header

```
packet timemarkPacket = {  
  subpacket  
  fixed<1+20+43>:hms  
  fixed<1+17+46>:hms  
  uint16:none  
  uint16:dec  
  uint16:dec  
  uint16:dec  
  uint16:hex  
} if (msgID == 3623)
```

```
header  
gpsSecs  
utcSecs  
pad[5]  
day  
month  
year  
data[wordCount-15]
```

Format of GPS TimeMark packet

```
packet allOthersPacket = {  
  subpacket  
  fixed<1+20+43>:hms  
  uint16:hex  
}
```

```
header  
gpsSecs  
data[wordCount-3]
```

Format of other packets

Protocol for GPS/IMU Data Stream

- Typical protocols for flight instruments run to hundreds of lines
 - User-defined data types and constants
 - Subpacket definitions
 - Multiple packet definitions

Real-life Protocols Are Large

- “Test” spec contains actions for each telemetry point to be performed on each applicable packet
 - Allows each telemetry point to be verified against user-defined conditions and/or conditionally displayed
 - Error and display conditions...
 - Use C-like syntax
 - Can reference the current, previous, and last-different values
 - Can reference the age (in packets) of the current value

“Test” defines conditions for each telemetry point

- For this example, want to...
 - Verify packet numbers are sequential
 - Verify that S/C time in each science packet is later than previous S/C time, but not by more than 5 seconds
 - Display the contents of each non-empty dump packet
- Nomenclature:
 - \$ refers to current value; _\$ is last value
 - "template", "check", and "show if" are keywords

```
template mytest = {  
    PacketNumber          check $ == _$+1  
    SpacecraftTime       check $ > _$ && $ <= _$+5  
    DumpLength           show if $ != 0  
    DumpData[0..254]     show if DumpLength != 0  
}
```

Simple Test Actions

- “Test” files may specify sequential goals to be met
 - Can be used to verify that a test completed successfully as reflected in telemetry
 - Goals are simply conditions using same syntax as used for checks

“Test” file defines optional goals to satisfy

- For this example, want to...
 - Verify that first packet in stream is science packet
 - Verify that we have at least one non-empty dump packet
- Nomenclature:
 - “goal” is a keyword

goal “First packet is science packet”
(PacketNumber == 1 && PacketType == NOMINAL)

goal “Found dump”
(PacketType == DUMP && DumpLength != 0)

Simple Test Goals

- HKCheck takes the protocol and test file(s), along with the binary telemetry input, and generates a report
- Reports show
 - Rules violated ("check")
 - Conditionally-displayed values ("show if")
 - Goals met and unmet ("goal")
 - Summary notes ("startnote" and "endnote")

Output

- In this portion of a run on flight telemetry from Mars Climate Sounder, HKCheck found an odd time increment (nominal is 2-3 seconds)
- Nomenclature:
 - “start” is a keyword which evaluates true the first time a packet type appears in the stream

SCTim has an error value: 887581376 (was 887581375)

Requirement:

```
start || Resets == _Resets+1 ||  
($ >= _$+2 && $ <= _$+3)
```

Output Example

LastCmd UPLOAD XRAM 0xcee7 138 0x80 0x75 0x2d

LastCmd UPLOAD XRAM 0xdd46 8 0x02 0xc6 0x77

LastCmd UPLOAD XRAM 0xde84 8 0x02 0xc6 0x30

LastCmd EQX 0 250

Met goal: "CRC check"

Met goal: "Pos-error resync #1"

Met goal: "Pos-error resync #2"

Met goal: "Pos-error resync #3"

...

Status has an error value: 0x42 (was 0x02)

Requirement:

\$ == 0x00 || \$ == 0x02 || \$ == 0x40

Met goal: "Pos-error resync #4"

EOF

All goals met

Failed -- found one or more errors

Another Output Example

- Useful for ASCII-fying telemetry through “show” statements as a test record
- Optionally generates spreadsheets as .csv files, or native Excel (with commercial add-on package)

Miscellaneous Capabilities

- Enables rapid, repeatable testing during development
- Post-launch telemetry can be scanned...
 - to confirm instrument health
 - postmortem, to look for odd conditions prior to a failure
- Allows expertise to be encoded in rules, reviewed, and carried through the life of the project
- Used for flight software regression testing or telemetry scanning on
 - Mars Climate Sounder (MRO), Diviner (LRO), Microwave Radiometer (Juno), Phoenix MECA, GALEX, and various airborne missions
- Open-source release pending

Summary