Monitoring the Execution of Space Craft Flight Software

Klaus Havelund,
Alex Groce, Margaret Smith
Jet Propulsion Laboratory (JPL), CA USA
California Institute of Technology

Howard Barringer
University of Manchester, UK
thanks to members of the MSL flight software team

• Chris Delp
• Dave Hecox
• Gerard Holzmann
• Rajeev Joshi
• Cin-Young Lee
• Alex Moncada
• Cindy Oda
• Glenn Reeves
• Lisa Tatge
• Hui Ying Wen
• Jesse Wright
• Hyejung Yun
outline

• the Mars Science Laboratory
• suggested testing methodology
• the LogScope monitoring system
• relationship to other work
our contributions

• user-friendly temporal quantified logic
• translation into quantified automata
• both can be used for specification
• runtime verification of log files
• testing of real spacecraft software
Mars Science Laboratory (MSL)
MSL information

• planned launch 2011
• biggest rover so far to be sent to Mars
• programmer team of 30
• testing team of 10+ people
• rover compute element (RCE): controls all stages of the integrated spacecraft
• programming language is C, 2.5 M LOC
• highly multi-threaded (over 130 threads)
testing
problems with testing FSW

• flight software engineers work under tight schedules: hard to access.

• system = hardware + software: it is cumbersome to run.

• difficult to determine what properties to define of what events.
emphasis on offline log analysis
separation of concerns
architecture

LogScope

spec

violations

LogMaker

[e_1, e_2, ..., e_n]
log events
log events

![Diagram showing the relationship between CHANNEL, COMMAND, PRODUCT, and EVR with state samples, state changes, commands, products, and transitions.]
example log

COMMAND 7308 {
  Args := ['CLEAR_RELAY_PYRO_STATUS']
  Time := 51708322925696
  Stem := "POWER_HOUSEKEEPING"
  Number := "4"
  type := "FSW"
}

EVR 7309 {
  message := "Dispatched immediate command POWER_HOUSEKEEPING: number=4, seconds=789086392, subseconds=1073741824."
  Dispatch := "POWER_HOUSEKEEPING"
  Time := 51708322925696
  name := "CMD_DISPATCH"
  level := "COMMAND"
  Number := "4"
}

EVR 7311 {
  name := "POWER_SEND_REQUEST"
  Time := 51708322925696
  message := "power_queue_card_request - sending request to PAM 0."
  level := "DIAGNOSTIC"
}

EVR 7312 {
  message := "Successfully completed command POWER_HOUSEKEEPING: number=4."
  Success := "POWER_HOUSEKEEPING"
  Time := 51708322944128
  name := "CMD_COMPLETED_SUCCESS"
  level := "COMMAND"
  Number := "4"
}

EVR 7313 {
  name := "PWR_REQUEST_CALLBACK"
  Time := 51708322944128
  message := "power_card_request - FPGA request successfully sent to RPAM A."
  level := "DIAGNOSTIC"
}

CHANNEL 7314 {
  channelId := "PWR-3049"
  DNChange := 67
  dnUnsignedValue := 1600
  type := "UNSIGNED_INT"
  Time := 51708322944128
  Name := "PWR-BCB1-AMP"
}

COMMAND 9626 {
  Args := ['set_device(1)', 'TRUE']
  Time := 51708372934400
  Stem := "RUN_COMMAND"
  Number := "18"
  type := "FSW"
}

EVR 9627 {
  message := "Validation failed for command RUN_COMMAND: number=18."
  DispatchFailure := "RUN_COMMAND"
  Time := 51708372934499
  name := "CMD_DISPATCH_VALIDATION_FAILURE"
  level := "COMMAND"
  Number := "18"
}

...
example log

COMMAND 7308 {
    Args := ['CLEAR_RELAY_PYRO_STATUS']
    Time := 51708322925696
    Stem := "POWER_HOUSEKEEPING"
    Number := "4"
    type := "FSW"
}

EVR 7309 {
    message := "Dispatched immediate command
POWER_HOUSEKEEPING: number=4,
seconds=78906392, subseconds=1073741824."
    Dispatch := "POWER_HOUSEKEEPING"
    Time := 51708322925696
    name := "CMD_DISPATCH"
    level := "COMMAND"
    Number := "4"
}

EVR 7311 {
    name := "POWER_SEND_REQUEST"
    Time := 51708322925696
    message := "power_queue_card_request-
sending request to PAM 0."
    level := "DIAGNOSTIC"
}

EVR 7312 {
    message := "Successfully completed command
POWER_HOUSEKEEPING: number=4."
    Success := "POWER_HOUSEKEEPING"
    Time := 51708372934499
    name := "CMD_COMPLETED_SUCCESS"
    level := "COMMAND"
    Number := "4"
}

EVR 7313 {
    name := "PWR_REQUEST_CALLBACK"
    Time := 51708322944128
    message := "power_card_request -
FPGA request successfully sent to
RPAM A."
    level := "DIAGNOSTIC"
}

CHANNEL 7314 {
    channelId := "PWR-3049"
    DNChange := 67
    dnUnsignedValue := 1600
    type := "UNSIGNED_INT"
    Time := 51708322925696
    ChannelName := "PWR-BCB1-AMP"
}

COMMAND 9626 {
    Args := ['set_device(1)', 'TRUE']
    Time := 51708372934499
    message := "Validation failed for command
RUN_COMMAND: number=18."
    DispatchFailure := "RUN_COMMAND"
    Time := 51708372934499
    name := "CMD_DISPATCH_VALIDATION_FAILURE"
    level := "COMMAND"
    Number := "18"
}

EVR 9627 {
    message := "Validation failed for command
RUN_COMMAND: number=18."
    DispatchFailure := "RUN_COMMAND"
    Time := 51708372934499
    name := "CMD_DISPATCH_VALIDATION_FAILURE"
    level := "COMMAND"
    Number := "18"
}

...
standard practice

• logs analyzed by writing Python scripts
  – properties coded up in Python
• this results in “specifications” that are:
  – time consuming to write
  – difficult to read, hindering:
    • maintenance
    • communication
    • specification-sharing
    • reuse
  – difficult to auto-generate
representation of logs in Python

```python
log = [
    {
        "OBJ_TYPE": "COMMAND",
        "Type": "FSW",
        "Stem": "PICT",
        "Number": 231,
    },
    {
        "OBJ_TYPE": "EVR",
        "Dispatch": "PICT",
        "Number": 231,
    },
    {
        "OBJ_TYPE": "CHANNEL",
        "DataNumber": 5,
    },
    {
        "OBJ_TYPE": "EVR",
        "Success": "PICT",
        "Number": 231,
    },
    {
        "OBJ_TYPE": "PRODUCT",
        "ImageSize": 1200
    }
]
```
the first scripture

trigger

look: DRILL_DMP
  evr(CMD_DISPATCH, positive)
  evr(CMD_COMPLETED_SUCCESS, positive)
  evr(CMD_COMPLETED_FAILURE, negative)
  chan(id: CMD-0004, positive, contains opcode of last immediate command)
  chan(id: CMD-0007, positive)
  chan(id: CMD-0001, negative)
  chan(id: CMD-0009, negative)
  prod(name: DrillAll, 1, *)

consequences
Property $P_1$

$P_1$: Whenever a flight software command is issued, then eventually an EVR should indicate success of that command.
Property $P_1$ refined

$P_1$: Whenever a **COMMAND** is issued with the **Type** field having the value "FSW", the **Stem** field (command name) having some unknown value $x$, and the **Number** field having some unknown value $y$, then eventually an **EVR** should occur, with the field **Success** mapped to $x$ and the **Number** field mapped to $y$. 
formalization

pattern P1:
COMMAND{Type:"FSW", Stem:x, Number:y} => EVR{Success:x, Number:y}

∀x, y ∙
□(COMMAND{Type:"FSW", Stem:x, Number:y} ⇒ ◇(EVR{Success:x, Number:y}))
pattern syntax

pattern ::= 'pattern' NAME ':' event '=>' consequence

consequence ::= event
            | '!' event
            | '[' consequence_1,...,consequence_n ']
            | '{' consequence_1,...,consequence_n '}'
Property $P_2$

$P_2$: Whenever a **COMMAND** is issued with the **Type** field having the value "FSW", the **Stem** field (command name) having some unknown value $x$, and the **Number** field having some unknown value $y$, then an **EVR** should thereafter not occur, with the field **Failure** mapped to $x$ and the **Number** field mapped to $y$. 
formalization

pattern P2:  
COMMAND{Type:"FSW", Stem:x, Number:y} =>  
! EVR{Failure:x, Number:y}
pattern syntax

pattern ::= 'pattern' NAME ':' event '=>' consequence

consequence ::= event
| '!' event
| '[' consequence_1, ..., consequence_n ']' 
| '{' consequence_1, ..., consequence_n '}'
Property $P_3$

$P_3$: Whenever a flight software command is issued, there should follow a dispatch of that command, and no dispatch failure before that, followed by a success of that command, and no failure before that, and no more successes of that command (exactly one success).
not:
EVR{
    DispatchFailure : x
}

not:
EVR{
    Failure : x,
    Number : y
}

not:
EVR{
    Success : x,
    Number : y
}

if

then

and then

COMMAND{
    Type : "FSW",
    Stem : x,
    Number : y
}

EVR{
    Dispatch : x,
    Number : y
}

EVR{
    Success : x,
    Number : y,
}
formalization

pattern P3:
COMMAND{Type:"FSW", Stem:x, Number:y} =>
[ ! EVR{DispatchFailure:x},
  EVR{Dispatch:x, Number:y},
  ! EVR{Failure:x, Number:y},
  EVR{Success:x, Number:y},
  ! EVR{Success:x, Number:y} ]
expressed in “quantified” LTL

\[ \forall x, y \bullet \]
\[ \Box (\text{COMMAND} \{ \text{Type: "FSW"}, \text{Stem: } x, \text{Number: } y \} \Rightarrow \neg \text{EVR} \{ \text{DispatchFailure: } x \} \cup \]
\[ \text{EVR} \{ \text{Dispatch: } x, \text{Number: } y \} \]
\[ \land (\neg \text{EVR} \{ \text{Failure: } x, \text{Number: } y \} \cup \]
\[ \text{EVR} \{ \text{Success: } x, \text{Number: } y \} \]
\[ \land \Box \Diamond \neg \text{EVR} \{ \text{Success: } x, \text{Number: } y \}) \]
pattern syntax

```
pattern ::= 'pattern' NAME ':' event '=>' consequence

consequence ::= event
            | '!' event
            | '[' consequence_1,...,consequence_n ']
            | '{' consequence_1,...,consequence_n '}'
```
property P4

$P_4$: Whenever a flight software command is issued, there should follow a dispatch of that command, and also a success, but the two events can occur in any order. In addition, there should never at any time (to the end of the log) after the command occur a dispatch failure or a failure of that command. Finally, after a success there should not follow another success for that same command and number.
formalized

pattern P4:
COMMAND{"FSW", Stem:x, Number:y} =>
{  EVR{Dispatch:x, Number:y},
  [  
    EVR{Success:x, Number:y},
    ! EVR{Success:x, Number:y}
  ],
  ! EVR{DispatchFailure:x},
  ! EVR{Failure:x, Number:y}
}
predicates

$P_5$: The success of a command with a number $y$ should never be followed by the success of a command with an equal or lower number $z \leq y$. 

pattern $P5$:

$\text{EVR\{Success: _, Number:y\} =>}$

$! \text{EVR\{Success: _, Number:z\} where \{: z \leq y :\}}$
Python predicate definitions

```python
{: 
def within(t1,t2,max):
    return (t2-t1) <= max
:}

pattern P6:
    COMMAND{Type:"FSW",Stem:x,Number:y,Time:t1}
    where {: x.startswith("PWR_") :} 
=>
    EVR{Success:x, Number:y, Time:t2}
    where within(t1,t2,10000)
```
specialized range predicates

pattern P7:
COMMAND {Type: "FSW", Stem: "PICT"} =>
[
    CHANNEL {DataNumber : {0 : 1, 4 :0}},
    PRODUCT ImageSize : [1000,2000]
]
event actions

```python
{: 
  counter = 0

  def count():
    global counter; counter = counter + 1

  def within():
    return counter < 3
:
}

pattern P8 :
  COMMAND {Name : x} where {: within() :} do {: count() :} =>
    EVR {Success : x} do {: print x + " succeeded" :}
```
scopes

- sometimes need for limiting the scope of a property
- without such scope a pattern is checked from it is triggered until the end of the log
- for example: check that some command leads to some behavior, but only up to the next command
scoped version of P4

```plaintext
pattern P4_
    COMMAND{Type:"FSW", Stem:x, Number:y} =>
    { EVR{Dispatch:x, Number:y},
      [ EVR{Success:x, Number:y},
        ! EVR{Success:x, Number:y} ],
      ! EVR{DispatchFailure:x},
      ! EVR{Failure:x, Number:y}
    }

upto COMMAND{Type: "FSW"}
syntax for patterns

pattern ::= 'pattern' NAME ':' event '=>' consequence
[
  'upto' event
]

consequence ::= event
| '!' event
| '[' consequence₁,...,consequenceₙ ']'
| '{' consequence₁,...,consequenceₙ '}'
from patterns to automata

- temporal patterns are translated into parameterized universal automata
- automata language more expressive
- user can use both, in practice only temporal patterns have been used for testing MSL
- automaton language forms a subset of the RuleR language (to be discussed)
automata characteristics

• states, transitions and events
• events are as in patterns:
  – can carry/bind data
  – can have executable Python predicates
  – can have executable Python actions
• states can be parameterized with data
• a transition can target multiple states
  – all must lead to success (\(\land\)-semantics)
recall P3

pattern P3:
COMMAND{Type:"FSW", Stem:x, Number:y} =>
[
  ! EVR{DispatchFailure:x},
  EVR{Dispatch:x, Number:y},
  ! EVR{Failure:x, Number:y},
  EVR{Success:x, Number:y},
  ! EVR{Success:x, Number:y}
]

[Image of a diagram showing the pattern]
automaton A3 {
    always S1 {
        COMMAND{Type:"FSW", Stem:x, Number:y} =>
            S2(x,y)
    }

    hot state S2(x,y) {
        EVR{DispatchFailure:x} => error
        EVR{Dispatch:x, Number:y} => S3(x,y)
    }

    hot state S3(x,y) {
        EVR{Failure:x, Number:y} => error
        EVR{Success:x, Number:y} => S4(x,y)
    }

    state S4(x,y) {
        EVR{Success:x, Number:y} => error
    }
}
recall P4

pattern P4:
COMMAND{Type:"FSW", Stem:x, Number:y} =>
{  EVR{Dispatch:x, Number:y},
    [    EVR{Success:x, Number:y},
        ! EVR{Success:x, Number:y} ],
    ! EVR{DispatchFailure:x},
    ! EVR{Failure:x, Number:y}
}
automaton A4 {
    always S1 {
        COMMAND{Type:"FSW", Stem:x, Number:y} =>
            S2(x,y), S3(x,y), S4(x,y), S5(x,y)
    }

    hot state S2(x,y) {
        EVR{Dispatch:x, Number:y} => done
    }

    hot state S3(x,y) {
        EVR{Success:x, Number:y} => S6(x,y)
    }

    state S4(x,y) {
        EVR{DispatchFailure:x} => error
    }

    state S5(x,y) {
        EVR{Failure:x, Number:y} => error
    }

    state S6(x,y) {
        EVR{Success:x, Number:y} => error
    }
}
@ S1

COMMAND{Type : "FSW", Number : y, Stem : x}

S2(x,y)  S3(x,y)  S4(x,y)  S5(x,y)

EVR{Number : y, Dispatch : x}  EVR{Number : y, Success : x}  EVR{DispatchFailure : x}  EVR{Failure : x, Number : y}

# done  S6(x,y)  # error  # error

EVR{Number : y, Success : x}

# ERROR
expressiveness

recall syntax for patterns:

\[
\text{pattern} ::= \text{'pattern'} \text{ NAME } ':\text{event} \Rightarrow' \text{ consequence} \\
\text{consequence} ::= \text{event} \\
| '!\text{event}' \\
| '[' \text{consequence}_1,\ldots,\text{consequence}_n ']' \\
| '{' \text{consequence}_1,\ldots,\text{consequence}_n '}'
\]

\[\text{trigger} = \text{one event}\]
so we cannot write

pattern P_with_composite_trigger:

\[
[ \\
\text{COMMAND\{Type:"FSW", Stem:x, Number:y\}}, \\
! \text{EVR\{DispatchFailure:x\}}, \\
\text{EVR\{Dispatch:x, Number:y\}} \\
] \\
=> \\
[ \\
! \text{EVR\{Failure:x, Number:y\}}, \\
\text{EVR\{Success:x, Number:y\}}, \\
! \text{EVR\{Success:x, Number:y\}} \\
]}

we can of course write an automaton that expresses the intended semantics.
other automata features

• success states (dual to hot states)
• step states: have to be left in next transition
running LogScope

```python
import logscope

log = createLog(fromsomewhere)

observer = logscope.Observer("$ISSTA/spec")
observer.monitor(log)
```
summary of errors

========================
Summary of Errors:
========================

P1  :  1 error
P2  :  0
P3  :  1 error
P4  :  3 errors
P5  :  0
P6  :  0
P7  :  3 errors
A3  :  1 error
A4  :  3 errors
automaton A4 {
    always S1 {
        COMMAND{Type:"FSW",Stem:x,Number:y} =>
            S2(x,y), S3(x,y), S4(x,y), S5(x,y)
    }
}

hot state S2(x,y) {
    EVR{Dispatch:x,Number:y} => done
}

hot state S3(x,y) {
    EVR{Success:x,Number:y} => S6(x,y)
}

state S4(x,y) {
    EVR{DispatchFailure:x} => error
}

state S5(x,y) {
    EVR{Failure:x,Number:y} => error
}

state S6(x,y) {
    EVR{Success:x,Number:y} => error
}
}
all errors caused by reaction
to command 9626

... COMMAND 7308 {
  Args := ['CLEAR_RELAY_PYRO_STATUS']
  Time := 51708322925696
  Stem := "POWER_HOUSEKEEPING"
  Number := "4"
  type := "FSW"
}

EVR 7309 {
  message := "Dispatched immediate command "
              "POWER_HOUSEKEEPING: number=4,"
              "seconds=789006392, subseconds=1873741824."
  Dispatch := "POWER_HOUSEKEEPING"
  Time := 51708322925696
  name := "CMD_DISPATCH"
  level := "COMMAND"
  Number := "4"
}

EVR 7311 {
  name := "POWER_SEND_REQUEST"
  Time := 51708322925696
  message := "power_queue_card_request-
              sending request to PAM 0."
  level := "DIAGNOSTIC"
}

EVR 7312 {
  message := "Successfully completed command "
              "POWER_HOUSEKEEPING: number=4."
  Success := "POWER_HOUSEKEEPING"
  Time := 51708322944128
  name := "CMD_COMPLETED_SUCCESS"
  level := "COMMAND"
  Number := "4"
}

**COMMAND 9626** {
  Args := ['set_device(1)', 'TRUE']
  Time := 51708372934400
  Stem := "RUN_COMMAND"
  Number := "18"
  type := "FSW"
}

**EVR 9627** {
  message := "Validation failed for command "
              "RUN_COMMAND: number=18."
  DispatchFailure := "RUN_COMMAND"
  Time := 51708372934499
  name := "CMD_DISPATCH_VALIDATION_FAILURE"
  level := "COMMAND"
  Number := "18"
}

...
*** violated: by event 9627 in state:

    state S4(x,y) {
        EVR{DispatchFailure:x} => error
    }
    with bindings:
        {'y':'18', 'x':'RUN_COMMAND'}

by transition 1:
EVR{'DispatchFailure':'RUN_COMMAND'} => error

--- error trace: ---

COMMAND 9626 {
    Args := ['set_dev(1)', 'TRUE']
    Number := "18"
    Stem := "RUN_COMMAND"
    Time := 51708372934400
    Type := "FSW"
}

EVR 9627 {
    name := "CMD_DISPATCH_VALIDATION_FAILURE"
    level := "COMMAND"
    Number := "18"
    DispatchFailure := "RUN_COMMAND"
    Time := 51708372934499
    message := "Validation failed for command
                RUN_COMMAND: number=18."
automaton A4 {
    always S1 {
        COMMAND{Type:"FSW", Stem:x, Number:y} => 
            S2(x,y), S3(x,y), S4(x,y), S5(x,y)
    }
    hot state S2(x,y) {
        EVR{Dispatch:x, Number:y} => done
    }
    hot state S3(x,y) {
        EVR{Success:x, Number:y} => S6(x,y)
    }
    state S4(x,y) {
        EVR{DispatchFailure:x} => error
    }
    state S5(x,y) {
        EVR{Failure:x, Number:y} => error
    }
    state S6(x,y) {
        EVR{Success:x, Number:y} => error
    }
}
*** violated: in hot end state:

state S2(x,y) {
  EVR{Number:y,Dispatch:x} =>
  done
}
with bindings:
  {'y':'18', 'x':'RUN_COMMAND'}

--- error trace: ---

COMMAND 9626 {
  Args := ['set_device(1)','TRUE']
  Number := "18"
  Stem := "RUN_COMMAND"
  Time := 51708372934400
  Type := "FSW"
}

*** violated: in hot end state:

state S3(x,y) {
  EVR{Number:y,Success:x} =>
  S6(x,y)
}
with bindings:
  {'y':'18', 'x':'RUN_COMMAND'}

--- error trace: ---

COMMAND 9626 {
  Args := ['set_device(1)','TRUE']
  Number := "18"
  Stem := "RUN_COMMAND"
  Time := 51708372934400
  Type := "FSW"
}
dynamic typing

- checking that specification format is “consistent” with log format
  - fields in events misspelled in spec
  - types of fields in spec inconsistent with log
- can of course be handled by static declaration of log event format – however, there may be hundreds of different events making it unpractical
- tool does its best to detect problems
specification learning

• writing specs is time consuming
• often hard come up with properties
• one approach is to use already generated log files to “get ideas”
• in the extreme case, specifications can be automatically generated from log files
architecture

learner → spec

monitor → spec

logs

yes

no:
...
...

spec
import logscope

log1 = ... ; log2 = ... ; log3 = ... ; log4 = ...

learner = logscope.ConcreteLearner("P")
learner.learnlog(log1)
learner.learnlog(log2)
learner.dumpSpec(sfile)

learner = logscope.ConcreteLearner("P", sfile)
learner.learnlog(log3)
learner.dumpSpec(sfile)

obs = logscope.Observer(sfile)
obs.monitor(log4)
the learned spec

using step and success states
case study

• tool design focused on test engineers
• dispatch/success pattern checked on 400 command scenarios part of pre-defined regression test suite: leading to discovery of double successes.
• one engineer auto-generated specs, leading to further error discoveries.
• one engineer used learning capability.
methodology observation

requirements:

pattern P1:
COMMAND{Type:"FSW", Stem:x, Number:y} => EVR{Success:x, Number:y}

integration of event-based requirements and logging
LogScope’s Inspirations

- **RuleR**: for the automata language, including how to handle parameters
- **RCAT**: for the emphasis on state machines and for hot states
- **RMOR**: for the lexical representation of state machines (and automated code instrumentation experience)
RuleR

• \( R_1(x) : R_2, R_3(y), a(x) \rightarrow R_4(x+1), R_5 \mid R_6(10) \)

• includes state machines
• states as well as events parameterized
  – with data: what we need
  – with rules: one can define temporal operators
• Petri-net semantics: lhs = conjunct
• AND + OR nodes: rhs = disjunct of conjuncts
requirements capture and analysis

RCAT
formal Requirements Capture & Analysis Tool

integration with SPIN verification and RMOR monitoring

a sample behavioral design requirement
```c
file = openfile("file42");
if (file != NULL) {
    process_file(file);
    closefile(file);
} else
    error_handling();
```
conclusions

• introduced temporal quantified patterns
• translated to quantified universal automata
• applied to log analysis: easy access
• suggestion: combine requirements engineering and event logging
• future work includes: GUI and better output, learning, merge with RuleR system: unifying patterns and automata
end