From Scripts to Specifications

The Evolution of a Flight Software Testing Effort

Alex Groce
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR
alex@eecs.oregonstate.edu

Klaus Havelund
Laboratory for Reliable Software
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA
klaus.havelund@jpl.nasa.gov

Margaret Smith
Software System Engineering
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA
margaret.h.smith@jpl.nasa.gov

The research described in this publication was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The first author was previously a member of the Laboratory for Reliable Software at the Jet Propulsion Laboratory.

ABSTRACT

This paper describes the evolution of a software testing effort during a critical period for the flagship Mars Science Laboratory rover project at the Jet Propulsion Laboratory. Formal specification for post-run analysis of log files, using a domain-specific language, LOGSCOPE, replaced scripted real-time analysis. Log analysis addresses the key problems of on-the-fly approaches and cleanly separates specification and execution. Mining the test repository suggested the inadequacy of the scripted approach, and encouraged a partly engineer-driven development. LOGSCOPE development should hold insights for others facing the tight deadlines and reactionary nature of testing for critical projects. LOGSCOPE received a JPL Mariner Award for "improving productivity and quality of the MSL Flight Software" and has been discussed as an approach for other flight missions. We note LOGSCOPE features that most contributed to ease of adoption and effectiveness. LOGSCOPE is general and can be applied to any software producing logs.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.5 [**Software Engineering**]: Testing and Debugging - Monitors, Tracing

General Terms

Languages, Verification

Keywords

Testing, test infrastructure, development practices, runtime verification, logs, temporal logic, space flight software, Python

ICSE '10, May 2-8 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-719-6/10/05 ...$10.00.

1. INTRODUCTION

One of the most difficult challenges in any large software development project is system testing. Even when most modules are well-specified, designed, and unit-tested, the integration of components produces a myriad of new problems, and a degree of complexity that is daunting to even experienced test engineers. One critical problem that arises at this point in the life-cycle of a software system is test evaluation and understanding. In part, this is a problem of specification: even when the behavior of individual modules is reasonably well specified, the expected behavior of the full system may be difficult to define. The problem is also more general than specification: in some cases, the emergent behavior of components is sufficiently complex that simply *understanding what the system does* in a particular test run is a problematic prerequisite to deciding if the software's behavior is correct. Moreover, ability to understand emergent behavior depends on the degree of visibility designed into the system.

This paper describes the evolution of the approach to test evaluation and understanding used in a large-scale software testing effort for the flight software (which executes in space and on the Martian surface) of the Mars Science Laboratory, a major planetary rover mission in development at NASA's Jet Propulsion Laboratory [1]. We show how a test team, with the assistance of verification researchers, moved from a purely script-based approach to test execution and evaluation to an approach using a formal specification language and log analysis tool, LOGSCOPE, to evaluate tests and aid test engineer understanding of system behavior. Although the framework has been developed specifically to support a space mission, it is fully general, and can be applied to any system producing logging information.

The goal of the Mars Science Laboratory (MSL) project is to put the thus-far largest rover (the size of a compact car) on Mars for continued exploration of Martian geology and climate, following on from the highly successful Mars Exploration Rovers, Spirit and Opportunity [12]. MSL is scheduled to launch in 2011. A flight software team peaking at around 30 programmers develops software for the Rover Compute Element (RCE), which controls all stages of the integrated spacecraft, from launch to roving on the Martian surface. Flight software is multi-tasking software that executes on the RCE on a VxWorks operating system platform, and provides much (but not all) of the embedded processing performed on the spacecraft. A testing team of approximately 10 engineers (the Flight software Internal Test, or FIT, team) is responsible for functional testing of the flight software. During 2008, the FIT team engaged members of JPL's Laboratory for Reliable Software (LaRS), a group specializing in automated software verification technology research, charged with the design and development of infrastructure for test automation. The LogScope tool was created as a response to challenges faced by the FIT team, and is an example of the often-sought marriage between formal methods research and practical software problem-solving. The evolution described in this paper was guided by research concepts (particularly in the design of the core language) and the needs of test engineers, as determined both by discussion and analysis of test library usage patterns. The structure of the specification language was suggested by a sketch of a log description, produced by a test team member. The testing effort benefitted from a fortunate coincidence of engineer needs with maturing research ideas.

1.1 Testing the MSL Flight Software

The MSL flight software produces rich log information, which is stored in SQL data bases (one database per log). From the point of view of a test engineer in the FIT team, testing the MSL software is very similar to commanding the spacecraft from JPL ground operations. A test issues commands via the JPL ground control software, which radiates the commands to the "spacecraft" (in the case of testing, a workstation simulation or actual flight hardware in the testbed). The ground system software then receives telemetry from the spacecraft, producing a log of the test execution. A log is, in essence, a sequence of (multiply) time-stamped events, where an event can be one of several forms, corresponding to input to and output from the system, as well as internal state transitions and state readings. Each event is, in essence, a mapping from field names to values (a record). There are four primary categories of events. After an operations team on the ground issues (1) **commands** to the spacecraft, the spacecraft responds to commands by changing its state (transitions, which are not, themselves, visible). The responses (and autonomous behaviors) of the spacecraft are observed via three kinds of telemetry: (2) **EVR**s (EVent Reports), essentially `printf`s in flight software used to indicate state changes (3) EHA **channels** that provide a snapshot of current spacecraft state and (4) **data products**, the "outputs" of the spacecraft downlinked to the ground, including engineering telemetry, images, and science instrument data.

MSL flight software logs are usually far too large to be effectively analyzed by humans (e.g., they will often contain hundreds of thousands of events). Understanding a test execution involves discriminating the important events relevant to a test from other necessary spacecraft operations. For example, when testing commands to fire pyros, the relatively small command sequence and response must be evaluated against a background of spacecraft events including initialization, communications, regular health and system state updates, and (particularly in early stages of testing) "noise" from faulty modules and health monitoring. All of this takes place under considerable time pressure, both in terms of a tight schedule for testing of each flight software release and, in many cases, a very limited period of access to testbed flight hardware (often scheduled during very late or very early hours). Simply configuring and running the flight software or troubleshooting difficulties with either the testbed installations or workstation simulation is often very time consuming, and time spent reading logs is time not spent developing new tests, running tests, or communicating results to flight software developers.

Prior to our involvement, test results (logs) had been analyzed on-the-fly, during test execution, by the test execution scripts, with properties coded in Python as queries for telemetry matching correct responses to commands issued. E.g., to test file uplink a script might check for an increment to a "files stored" counter (a telemetry channel), check another channel to determine if the number of free bytes of NVM had decremented, watch for the appearance of an EVR indicating the file of the particular name had been stored on NVM (non-volatile memory), and issue a command to list the contents of the target directory, causing a data product containing the filename to be generated and downlinked. Such scripts are time consuming to produce and result in difficult-to-read "specifications" that hinder communication, maintenance, and specification-sharing and reuse. Hand-scripted analysis also seldom provides any assistance in understanding a complex log, as the effort required to generalize analysis beyond the minimum required for a specific test is too high: in practice, understanding of test runs was largely obtained by slow, inefficient interaction with the ground system GUI view of telemetry. Moreover, because telemetry may arrive out-of-order due to different priorities, and is not visible until the ground system database populates, scripts that process telemetry tend to execute very slowly. While high levels of human interaction were, perhaps, required in very early testing, to improve understanding of the system, lack of automation resulted in very slow test turn-around and inefficient use of limited hardware testbed time.

These, and other problems, discussed below, motivated a (partial to date) move towards post-execution analysis of test run logs, using a separate, domain-specific language designed to facilitate specification and understanding of completed test executions. Figure 1 shows how the specification-driven LogScope tool replaced the portion of the script that evaluated a test. Test execution and evaluation are now separated. A test engineer instead develops a test script, which focuses only on commanding the flight software, and a specification (or set of specifications), which describes correct behavior. A logging tool (named LogMaker) automatically produces a complete log of test execution, which is "monitored" (in a post-mortem fashion, after test execution is complete) with respect to the specification(s) by the LogScope tool. The tool produces error traces for all specification violations, and a summary of the test execution for the engineer to examine. Furthermore, various statistics are produced to aid test understanding. A test engineer may also use LogScope "in reverse": as a *learning* tool to automatically generate a specification from one or more execution logs. The specification can subsequently be used for checking future logs. The key point to take away is the separation between test execution (which may be controlled by a test script, as usual, or even performed by hand at the command console), and the analysis of results, performed after a test run is complete. Previously, test evaluation relied on script-controlled feedback between the flight software and the ground operations software, significantly increasing the amount of traffic and timing dependencies.

1.2 Contributions and Related Work

The contribution of this work is the story of injection of an automated formal methods approach to test evaluation into a NASA flight mission's testing process, and an analysis of how the test process in question evolved from an approach relying purely on interactive test scripts to a specification-based post-run log analysis approach. The use of light-weight data mining of test scripts to focus infrastructure development effort by understanding actual test engineer practices is a feature that may also be of interest to other test teams. The LogScope language was originally introduced in [4]. We refer readers to this publication and to [3] for further details of the syntax and semantics, and how this system relates to other runtime verification approaches. LogScope is specifically influenced by the Ruler system [5, 6], but adds a user friendly form of temporal logic, and integration with Python, both critical to MSL requirements. The success of a temporal-logic-inspired specification language might appear somewhat surprising, as state

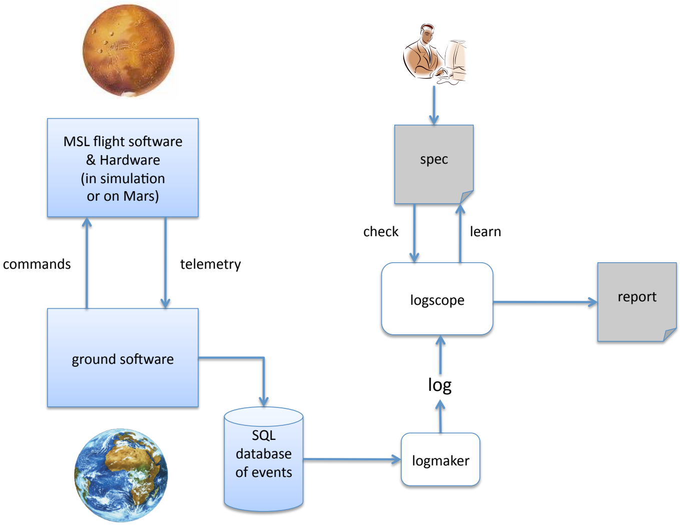

Figure 1: Specification-based log analysis.

machines are often considered more acceptable for adoption in engineering settings. It is our belief that the temporal pattern language strikes a good balance between traditional temporal logic and regular expressions. Also relevant is the literature of log analysis in testing, particularly the work of Andrews and Zhang [2]. The similarities and critical differences between LOGSCOPE and other temporal specification languages, including PSL [13], MCL [11], PDL [8], and GIL [9] are described in our previous publications.

In previous work [7] we developed a PYTHON API for commanding and monitoring the space shuttle launch platform at Kennedy Space Center during preparation of a launch. This work was part of NASA's Constellation Launch Control System (LCS) project, a part of NASA's Constellation project developing the replacement for the space shuttle. The API has similarities to the PYTHON scripts developed for testing the MSL flight software. Monitoring was online, synchronized with the launch preparation period, supporting detection of property violations as they took place.

1.3 Paper Outline

The paper is organized as a presentation of three stages along the road to the final LOGSCOPE system. Section 2 presents stage one: how test scripts were written initially by test engineers. Section 3 presents stage two: the introduction of a specialized so-called *registry* for recording events both prohibited and required as a consequence of submitted commands during a test. Section 4 examines the registry solution and leads to the final specification-based LOGSCOPE solution presented in Section 5. Section 6 provides further details on how the LOGSCOPE specification language presented in the previous section evolved as an interaction between test engineers and researchers.

2. STAGE ONE: AD HOC SCRIPTS

The primary concern in this paper in the transition from a script-based imperative approach to execution evaluation to a declarative approach based on a separate log-analysis tool. Before investigating the log-based approach, however, the FIT team first attempted to improve test infrastructure support for script-based on-the-fly evaluation of tests.

The testing approach used by the FIT team relies on a number of layers of software mediating interaction with the MSL flight code. First, the JPL ground control software system is responsible for sending commands to and receiving telemetry from the spacecraft. Access to this system in scripts is provided by a PYTHON library also produced by the ground systems team. The ground access library provides only a low level interface to telemetry and commanding, and is seldom directly called by test scripts. Another PYTHON library, referred to as "the FIT tool," provides a more convenient interface for issuing commands, receiving telemetry, and other core test features. In addition to providing command and telemetry features and limited test evaluation, the FIT tool automatically labels and numbers test steps and produces an annotated, readable "log" of test execution, including information as to steps that fail or pass (in some cases, as in command dispatch, automatically). While it lacks the detailed information of all test events required for analysis, this log (distinct from the ground systems log) provides a readable record of test activity. In the remainder of the paper, the term *log* refers to the ground systems log of a complete test run, not the human-readable "log" produced by the FIT tool.

At the time LaRS became involved in MSL testing, the FIT tool provided only very simple methods for issuing commands. In general, the tool supported a purely synchronous approach to testing: issue a command, and wait until all expected telemetry arrives before proceeding to another command. Given the varying response times to different commands and the frequent need to repeatedly check for expected changes in spacecraft state, many scripts were forced to introduce very long delays between commands and clutter test execution steps with repeated queries for expected but variably timed spacecraft state changes.

Figure 2 shows a (simplified) portion of code, typical of such FIT test scripts during the first stage of the testing effort. This code

```
old_val_0001 = FIT.get_eha("CMD-0001",timeout=60)
old_val_0007 = FIT.get_eha("CMD-0007",timeout=60)
old_val_0009 = FIT.get_eha("CMD-0009",timeout=60)
...
FIT.send_fsw_command("DRILL_DMP")
FIT.pause("Press <ENTER> to continue test.")
FIT.wait_eha("CMD-0004", eha_dn=FIT.get_opcode("DRILL_DMP"),
             timeout=60)
FIT.wait_eha("CMD-0007",eha_dn=old_val_0007+1,timeout=60)
val_0001 = FIT.get_eha("CMD-0001",timeout=60)
if (val_0001 > old_val_0001):
   FIT.fail("Dispatch failure")
val_0009 = FIT.get_eha("CMD-0009",timeout=60)
if (val_0009 > old_val_0009):
   FIT.fail("Validation failure")
FIT.wait_data_product("DrillAll",1,None)
```

Figure 2: Original FIT approach to test evaluation.

issues one command to flight software, DRILL_DMP, using the FIT library call send_fsw_command. This call pauses test execution until certain standard responses to command execution (*command dispatch* and *command success* EVRs) have been received in the telemetry stream, using default timeout values (which can be overridden during the call to send_fsw_command). The script then includes a manually controlled pause, so that the test engineer can continue the script once the GUI display of ground telemetry shows that responses to the command have arrived. Such manual pauses were common in early test scripts, in order to avoid multiple queries for telemetry responses with variable arrival time. The script then receives and evaluates values for four command-related channels, CMD-0004, CMD-0007, CMD-0001 and CMD-0009[1]. In this case, the test waits until new values arrive for two channels, CMD-0004 and CMD-0007, with the test failing if (1) no values arrive within 60 seconds or (2) the values received do not match the specified "dn" (data numbers), respectively the opcode of the command issued, and the previous value of the command channel incremented by one. The second check uses a previously stored value for the channel, old_val_0007, captured at some prior point (shown at the top of the example code) during the test. For channels CMD_0001 and CMD_0009, the script uses a get_eha call rather than waiting for a new value. This call returns the last value for the channel to appear in the telemetry stream, however far back in test execution that value may have been. In this case, the assumption is that enough time has passed that current values for these channels will be available, and there is no need to wait for another periodic update. These channels store counts for two different forms of command failure. Explicit PYTHON code compares the obtained values to previous values, and fails the test (with an appropriate error message) if either channel has incremented. Finally, the script waits until a data product, the file produced by the DRILL_DMP command, arrives on the ground.

The key point to observe is that this script is both slow and brittle. Both problems derive from the generally synchronous nature of the library calls: the test script may spend many seconds waiting for telemetry to arrive, and if the expected order and timing of events is off, it may miss critical events, or make use of outdated channel values when evaluating results (e.g., consider the case in which the latest value of CMD_0001 is delayed and does not arrive until after the get_eha call). The variance in delays is such that a manual delay has been introduced (a call of FIT.pause), preventing automated test execution without human involvement.

```
old_val_0001 = FIT.get_eha("CMD-0001",timeout=60)
old_val_0007 = FIT.get_eha("CMD-0007",timeout=60)
old_val_0009 = FIT.get_eha("CMD-0009",timeout=60)
...
FIT.send_fsw_command_and_proceed("DRILL_DMP")
FIT.register_EHA("CMD_0007", eha_dn = old_val_007+1,
                 timeout=60)
FIT.register_EHA_negative("CMD_0001", eha_dn = old_val_0001+1,
                          timeout=60)
FIT.register_EHA_negative("CMD_0009", eha_dn = old_val_0009+1,
                          timeout=60)
FIT.register_data_product("DrillAll",1,None)
```

Figure 3: Using the registry.

3. STAGE TWO: THE REGISTRY

In order to address the need for asynchronous interaction with telemetry, the FIT tool infrastructure team added support for an event *registry*, a database of expected or prohibited events, updated from the test script as calls to certain functions in the FIT library *registry API* when commands were issued. The FIT tool would then automatically query the registry at each step of a test, to check whether prohibited events had occurred (indicating failure, to be reported) or whether expected events had occurred (indicating success, that could now be discharged). Rather than pausing test execution after each command, a script could now issue a command, register the telemetry related to success and failure, and proceed.

Figure 3 shows a similar script, now using registry calls in place of synchronous telemetry queries. The key differences are:

1. The send_fsw_command call has been replaced by the asynchronous call send_fsw_command_and_proceed. This call issues the DRILL_DMP command and places all the standard required events, including dispatch, success, and the requirement that CMD-0004 update to contain the command's opcode, in the registry. Test execution immediately proceeds to the next step, without waiting for these events to occur.

2. Expressing checks for what events to appear or not appear in the telemetry stream is now simply a matter of registering these in the registry.

We call this script *similar* to the first example, rather than *equivalent* because the semantics is potentially slightly different: some tests may pass in this case that fail in the previous case, and vice-versa, due to the precise intricacies of timing (when the registry queries telemetry vs. the original script).

However, the second version of the test is considerably simpler (at least on the surface; we will return to this issue later) than the first version, and is to some extent less brittle. It is also much faster: there is no manual pause, and the registered events can succeed or fail in the background while other commands are issued and results evaluated. In addition to the features shown here, the registry supports explicit *barriers*, forcing test execution to pause until certain critical events are discharged from the registry (useful, for example, when one command's results are pre-requisites for issuing another command).

The registry approach was developed in close cooperation with test engineers, as a response to their needs as execution time became a driving concern. During the requirements-gathering stage,

[1] The details of channelized telemetry are quite complex: channel values in the telemetry stream are sometimes automatically produced when the channel value changes, and at other times produced as part of a periodic general downlink of current channel values, as defined by a fairly rich set of criteria.

it became clear that the registry would be a fairly complex system, with a non-trivial semantics, due to the need to handle a wide variety of timeout and ordering issues in different testing sub-domains.

Once registry functionality was implemented, the infrastructure team expected to receive bug reports, feature requests, and calls for help. Surprisingly, while the FIT team acknowledged the release and asked a few simple questions, there was very little response. The development and preliminary testing of the registry code had suggested a different story. The high variability in event timing had made setting timeout parameters for expiration of registry events a difficult art. In some cases, the developer's unit test scripts failed occasionally due to missing telemetry, which required an examination of the GUI displays of events to confirm that the expected messages were not present in the logs, rather than missed by the low-level tools or improperly handled by the registry code. While registry development was not difficult, the delivered state of the tool was not expected to immediately meet all engineer needs.

The semantics of registry events was considerably more complicated than at first apparent. In order to understand tests using the registry it was important to know the answers to numerous questions not visible in the test script code itself. In particular, the ordering of events was non-trivial. Consider the case of a simple channel (EHA) value added to the registry. Is the check for `CMD-0007` satisfied by *any* appearance of the desired value in the telemetry stream? This is not the case. Rather, the registered event is only satisfied if the value appears in the stream *after* the last command issued by flight software, since test evaluation concerns itself with responses to commands. Unfortunately, when a command is issued this is a *ground* event, and there is no uniform timestamp to provide a (partial) ordering of command issue with respect to spacecraft events. The registry uses the *dispatch* event on the spacecraft as a canonical timestamp for the command event. Similarly, a user must understand the semantics of a registry timeout: is a timeout with respect to test execution time, or spacecraft event time? This question is particularly complicated when tests execute on the workstation simulation, in which case test time may proceed at a rate six times faster than "wall-clock" time. The FIT tool development team expected to hear many variations of these, and other questions from test engineers as they began to use the registry.

Why were users either not encountering these problems, or not requesting assistance with them, and demanding a more detailed set of examples and semantic documentation? Discussions in meetings continued to produce minor feature requests for the registry, but did not suggest significant frustration with the state of the tools. It seemed possible but unlikely that the greater knowledge of the test engineers made the timeout problem and other troubleshooting less onerous.

4. RE-EXAMINING THE REGISTRY

4.1 Mining The Tests

The lack of help requests and bug reports for the registry code left the infrastructure team with sufficient time to turn to a long-overdue refactoring and tightening of the FIT tool code. FIT team management agreed that an overhaul to remove duplicated functions, improve documentation, check for exposure of "hidden" internal functionality, and generally improve the readability and maintainability of the core tool code was in order. However, the refactoring would need to attempt to minimize impact on in-use test scripts; breaking legacy scripts no longer frequently executed was acceptable, but for the test areas currently underway, work would need to be postponed or designed around current usage patterns.

The infrastructure team hypothesized that the largest barrier to tool readability was a large weight of (nearly) dead code. In order to plan the refactoring process, and confirm this hoped-for prevalence of easily removed code, the team wrote a tool to analyze the entire repository of test script code from all engineers. For each function in the FIT tool code base, the analysis reported all calls in all test scripts, as shown in Figure 4. The tool also reported calling chains, noting functions that were called through other FIT functions, and reported on internal-use-only functions improperly called in test scripts. The infrastructure team was able to remove a moderate number of unused (mostly older) functions, and alerted one test engineer of an alternative public function to call in place of an internal pretty-print function. The most important information, however, was the full set of calls to registry functions, remarkable in two ways:

```
Calls for FIT._print:
  WARNING:  Internal use only!
  throughput.py: FIT._print ("Radiate commands.")
Calls for FIT.barrier:
  downlink2.py:   FIT.barrier(timeout=60,failTest=True)
Calls for FIT.register_EHA:
  fvs.py:         FIT.register_EHA("CMD-056", delta=3,
                                   timeout=200)
Calls for FIT.register_EVR:
  throughput.py:  FIT.register_EVR("UPLINK_RECV")
  pyros.py:       (indirect) FIT.send_fsw_cmd
                                 ("FIRE_PYRO,11",
                                 extra_EVR="PYRO_FIRE",
                                 timeout=False)
...
```

Figure 4: Sample output of mined usage from the repository.

1. There were *very* few calls to registry functions. More than half of the available functions were never called; all but three were called at most twice in the entire repository.

2. The functions that were called were the simplest functions, and were called without any optional arguments (including timeouts), resulting in the simplest possible behavior.

4.2 Limitations of the Registry

In general, test engineers were using the registry to code up simple post-processing of events without timeouts, rather than exploiting its ability to provide asynchronous on-the-fly evaluation of test results. Why? Discussion with individual developers revealed two primary reasons and one secondary problem:

1. **Timing and ordering of events:** Events are not downlinked to the ground system in chronological order. Different events have different priorities, and the "Earth receive time" ordering of two events will often be the opposite of their event times on the spacecraft. Test engineers were, even when using the registry, forced to introduce lengthy delays after test steps (resulting in even slower tests) or build very complicated logic to "recreate" a linear chronology of events, in order to avoid confusion of similar responses to different commands. The registry's "automatic" ordering of events by registration of command execution was too simplistic to handle the out-of-order arrival of command responses. Pausing long enough to allow telemetry from one command to arrive before proceeding with potentially confused test operations proved brittle, as timeouts fluctuated with each code release. Asynchrony alone simply did not address the problem of constructing an unambiguous event ordering. It was

easiest for engineers to make very simple registry calls, ensure that all events were discharged by test termination, and handle oddly ordered events or delays long enough to indicate bugs by hand or with *ad hoc* code for (essentially) post-processing the event log.

2. **Difficult to read and re-use scripts:** A key motivation for the event registry was that it integrated with the idea of tests as scripts with logic, looping, and other programming language features, and the FIT toolkit as a “Swiss army knife” for MSL software testing. Unfortunately, test scripts that were readable when their task was limited to *commanding* the spacecraft became almost impossible to follow when test execution was interlaced with test evaluation. Even the very simple form of registry use found in scripts sometimes doubled the amount of text found in a simple commands-only script, by mixing commands with required event responses and channel value queries. Many properties were used in more than one script, but coded up differently in each case because of the need to take the context of a command into account. Examination of the source repository mining results made it particularly clear that the registry approach and *ad hoc* post-processing forced engineers to continually re-invent the wheel, as multiple scripts by the same engineer (much less different engineers) would often feature subtly different implementations of the same behavioral specification.

3. **The registry was overly complex:** Understanding the usage of the more complex features of the registry required adding a number of new concepts to a test engineer’s understanding of test execution. In addition to tracking the commands sent to the flight system and the telemetry received from the system, it was now necessary to understand the synchronization of registry checks, barriers, a set of timeouts (some explicit and some implicit), and different methods used for handling events and channels in the registry. The registry functions all provided numerous (optional) arguments, some with fairly complex implications for testing. Even the developer of the registry found the library difficult to use. The lack of questions was only due to a general inability to even begin using the more semantically involved registry features.

These problems are not particularly tied to the MSL flight software. The first problem appears in *any* distributed system in which constructing an event timeline is non-trivial, and the second problem is a general observation about test case readability and code re-use. The registry system failed in part because it was designed and adopted just as engineers (most of whom were new to the MSL testing task) began to face more complex testing tasks: the needs of testing forced the development of a stop-gap solution before the problem was well understood. While each engineer, in isolation, might feel that the registry was not solving the key problems of testing, the problems were not insurmountable. An apparent (unspoken) general assumption was that, while the registry might not precisely fit *my* testing task, it was solving problems for other engineers, and at least tests were now executing much faster. Only looking at the complete body of test code made it evident that the registry was (other than, perhaps the problem of test speed) not solving the critical problems of evaluation and understanding. We believe that similar very lightweight data mining may be useful to other teams facing usability problems in a testing context; because test software is not an external deliverable item, or subject to independent quality assurance, it is often difficult to discover usability or design failures in test infrastructure code, unless test engineers can discover more widespread usage patterns. Individual engineers are often resourceful enough to work around tool limitations, and may easily assume that problems are specific to their particular tests only. Such “universal workarounds” may become common practice without ever being discussed at group meetings or codified in test infrastructure documentation. Only the actual test code embodies the actual practice of engineers.

```
look:DRILL_DMP\
  evr(CMD_DISPATCH,positive)\
  evr(CMD_COMPLETED_SUCCCESS,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,*)
```

Figure 5: The DRILL_DMP specification mock-up.

Due to the inherent problems of on-the-fly evaluation and the shortcomings of the registry library, users were, in the more complicated and automated tests, hand-scripting post-test analysis of all telemetry (ad-hoc construction of very limited logs). A few checks were routinely performed online, but complex analysis was often delayed until the test was complete, when chronological confusion could be partially avoided by simply *counting* expected responses. In some cases, engineers were still hand-examining telemetry after test execution.

FIT team management (also active in writing tests), test engineers, and LaRS all concluded that building numerous test-specific “log analysis” systems was not an effective use of team time, and that simply improving the registry tool and clarifying its semantics would not address the first two problems. After a series of team meetings and discussions, informed by the LaRS team’s research interest in log analysis, a test engineer produced a specification mock-up, a starting point for LOGSCOPE, shown in Figure 5.

The text represents a property to be checked: when a `DRILL_DMP` command is observed in the log, then the events tagged `positive` should follow in any order, and the events tagged `negative` should not occur. For example, the following `evr` (event report) events should follow: a report of the dispatch (`CMD_DISPATCH`) of the command; a report of the success (`CMD_COMPLETED_SUCCCESS`) of the command. Additionally, there should *not* be an `evr` reporting failure (`CMD_COMPLETED_FAILURE`) of the command. This event set is followed by requirements on samplings of the state (channel events). For example, there should be a sampling of state variable `CMD-0004` that contains the opcode of the last immediate command (obviously at this point the precise syntax was not fully determined). Finally, the flight software should downlink a `DrillAll` product (a `prod` event) to ground informing about the status of the drill. This mock-up is, of course, a new version of the specification embedded in our previous test script examples! This specification is very different in style from the previous versions: it declares the contents of a final log, to be examined after test execution, rather than providing a recipe for querying the system during test execution. The details of commanding the spacecraft are left to a test script, and the details of constructing an event order and collecting relevant telemetry are delegated to another tool, no longer the responsibility of the test engineer. By *specifying* contents of a final log, after all events have been downlinked to the ground, the new approach sidesteps the difficult and confusing issue of attempting to produce a canonical ordering of events on-the-fly. Any on-the-fly ordering of events is inherently unstable, so only post-test analysis of a *unified log* can really solve the ordering problem.

Using the example mock-up specification in Figure 5 as inspiration, we designed the LOGSCOPE specification language, and implemented a system for monitoring log files against such specifications. This language and monitoring system will be briefly presented in Section 5. The final specification language resulted from an interaction with the test engineers, as will be explained in Section 6. Starting with a syntax and informal semantics designed by the test team ensured that the language would be usable; enhancing the resulting language with concepts from formal methods research ensured that the language would be expressive enough to meet future testing needs.

Top MSL flight software management suggested that a specification-based approach would also be useful to developers outside the test team[2], and introduced us to the ground software team responsible for defining the logging system. This made it possible for us to move from querying the ground system or parsing logs produced by the ground software to directly using the SQL event databases used by ground software. This allowed us to, e.g., filter events through SQL queries. After an initial trial run, in which the LaRS team replicated hand-scripted results of random command executions, showing the suitability of log analysis for basic testing tasks, FIT team members began use of prototype versions of LOGSCOPE, replacing hand-scripted post-test evaluation code, as described in more detail in sub-section 5.5.

5. STAGE THREE: LOGSCOPE

The LOGSCOPE system allows us to separate commanding of the flight software from the specification of its expected behavior. Commanding is performed using PYTHON test scripts as before, whereas the specification of expected behavioral properties is now expressed in the LOGSCOPE specification language. The PYTHON test script will in principle just consist of code that submits commands to the flight software. The following script submits the drill dump command:

```
FIT.send_fsw_command_and_proceed("DRILL_DMP")
```

Running the script will cause the flight software to attempt an execution of the command, while the ground system stores execution events in an SQL database created for the run. The LOGSCOPE system takes as input the execution log extracted from the database, and a formal specification of the expected contents of the log file (possibly represented in several different files), and produces a report enumerating every violation of the specification.

5.1 The Log

A run of the drill dump command could result in a log of the form illustrated in Figure 6, expressed as a PYTHON *sequence* (list), the format in which logs are processed by LOGSCOPE. The log contains a sequence of events, each represented by a PYTHON *dictionary*: a mapping from fields to values (a *record* using traditional programming terminology). Each event has a `OBJ_TYPE` field, indicating what kind of event it is: a command (`COMMAND`) representing input to the system, an event report (`EVR`) representing some internal transition, a state variable/channel change (`CHANGE`), or a product (`PRODUCT`) representing output from the system. Each event also has a canonical time stamp, `Time`. Commands and event reports have a command number so that event reports can be related to the commands that cause them. Each change event indicating a change of state variable carries the old value (the field `Dn_old`) as well as the new value (the field `Dn`).

```
log =
  [
    {
     "OBJ_TYPE" : "COMMAND",
     "Time" : 3700393,
     "Stem" : "DRILL_DMP",
     "Number" : "1",
     "Type" : "FSW" },
    {
     "OBJ_TYPE" : "EVR",
     "Time" : 5030468,
     "Dispatch" : "DRILL_DMP",
     "message" : "Dispatched DRILL_DMP",
     "Number" : "1" },
    {
     "OBJ_TYPE" : "CHANGE",
     "Time" : 22736937,
     "Id" : "CMD-0004",
     "Dn" : 42,
     "Dn_old" : 41 },
    {
     "OBJ_TYPE" : "CHANGE",
     "Time" : 44937474,
     "Id" : "CMD-0009",
     "Dn" : 101,
     "Dn_old" : 100 },
    {
     "OBJ_TYPE" : "PRODUCT",
     "Time" : 320378725,
     "Name" : "DrillAll",
     "message" : "drill product",
     "Count" : 1 },
    {
     "OBJ_TYPE" : "EVR",
     "Time" : 320378950,
     "Success" : "DRILL_DMP",
     "message" : "Completed DRILL_DMP",
     "Number" : "1" }
  ]
```

Figure 6: Example log.

In addition to fields present in the original database, the log creation tool *annotates* events with *derived fields* that ease readability and specification. Field IDs from the MSL database begin in lowercase, while derived fields (e.g., `Dispatch`) begin in uppercase. The `Time` field, used to order events, is the most critical derived field. Events that take place on the spacecraft include a spacecraft event time `scet` that establishes a canonical order. However, command events originate from the ground, and include only a ground transmission time. The log maker establishes a uniform chronology for a log by extracting the time a command is actually dispatched on the spacecraft from the EVR notifying ground of the dispatch. This leads to a feedback between logging and telemetry design on the spacecraft (in that as the spacecraft changes its notifications, we modify our logging system, and as we change our logging, engineers may learn more about needed telemetry). In fact, use of our tool by test engineers has increased the MSL software team's awareness of ambiguities in timing of events originating on the spacecraft, suggesting that improved synchronization is needed between modules responsible for different types of telemetry. The timestamp on a channel value may derive from the beginning of a pass over all channels, while the value for a channel is stored later, leading to instances of a command's effect apparently taking place

[2] The FIT tool itself was already used by some developers and hardware testbed team members.

before the command was issued. While very rare, such a timeline is very confusing for test engineers and developers.

To LogScope, a log is simply an ordered list of records with named fields. All MSL-specific aspects of the log are encapsulated in the logging tool, making later tools in the chain easily adaptable to *any* system producing such a log. This approach is applicable to operating systems, web servers, and any other systems producing event-based logs. Even when a system produces logs in a purely textual format, a translation into such a canonical format may be critical to *maintenance* of a log analysis framework: adapting hand-built regular expressions to changes in logging output is time-consuming and error-prone (indeed, it has some of the worst features of hand-scripted imperative evaluation). Constructing specifications that largely rely on *derived* fields encapsulates all parsing and interpretation of log events in one maintainable, independent tool.

5.2 The Specification

The example LogScope specification of what it means for a log to be well-formed consists of two properties, a general property that is expected to hold for all command executions, and a drill-command specific property. The two properties together represent the properties checked in test scripts shown in earlier sections, although now logically representing the correct versions of these properties, and not the formulations engineered for easy implementation in scripts. The LogScope specification in its entirety is presented in Figure 7.

```
{: import FIT :}

pattern DISPATCH_SUCCESS :
  COMMAND{Stem: st, Number : nr} =>
  [
   EVR{Dispatch:st, Number:nr},
  !EVR{Failure:st, Number:nr},
   EVR{Success:st, Number:nr}
  ]
  upto COMMAND{Type:"FSW"}

pattern DRILL_CONSEQUENCES :
  COMMAND{Stem:"DRILL_DMP"} =>
  {
    [
     CHANGE{Id:"CMD-0004", Dn:dn}
       where
       {: dn == FIT.get_opcode("DRILL_DMP") :},
     CHANGE{Id:"CMD-0007", Dn_old:dn_old, Dn:dn}
       where
       {: dn == dn_old + 1 :},
     PRODUCT{Name:"DrillAll", Count:1}
    ],
    !CHANGE{Id:"CMD-0001"},
    !CHANGE{Id:"CMD-0009"}
  }
```

Figure 7: LogScope specification.

The specification starts with the import of the `FIT` Python library, within special `{: ... :}` brackets. The `FIT` library provides, among other features, the function `get_opcode`, also used in the scripts presented earlier. This function is called in the specification. Note that it is generally permitted to introduce arbitrary Python definitions (imports, variables, functions, classes, objects) in between `{: ... :}` brackets and refer to these within the properties in special `where`-clauses, as shown in Figure 7. This feature makes the specification language flexible and powerful, a key criteria for acceptance by the test team.

The specification consists of two properties, or *patterns* as they are called, `DISPATCH_SUCCESS` and `DRILL_CONSEQUENCES`. Each pattern is of the form: *event* => *consequence*, with the informal interpretation that if the *event* to the left of => is observed in the log, then the *consequence* must be observed in the remainder of the log. The property `DISPATCH_SUCCESS` expresses that if a command with a name (stem) `st` and number `nr` is observed, then:

1. an `EVR` must be observed, indicating a dispatch of the command with name `st` and number `nr`. Note that the command event binds the variables `st` and `nr` to actual values in the log, and these now become constraints in the consequence.

2. subsequently, no `EVR` event indicating failure of that command should be observed (`!` is negation).

3. finally an `EVR` indicating success of that command should be observed.

The square brackets `[...]` indicate that the ordering matters: the dispatch should come before the success, and there should be no failure in between the two. The consequence of the command is checked upto the next flight software command, at which point missing events are reported as violations. The `upto` constraint limits the scope of a property.

The `DRILL_CONSEQUENCES` pattern states that after a drill command (a command where the field `Stem` has the value `"DRILL_DMP"`) there should:

1. in order (indicated by the inner `[...]` brackets):
   (a) occur an update (change) to the state variable/channel named `CMD-0004`, where the new value (the value `dn` of the `Dn` field) should be equal to the value of the Python expression: `FIT.get_opcode("DRILL_DMP")`. The `E where {: P :}` clause allows an event `E` to be constrained by the Python predicate `P`.
   (b) occur an update to the `CMD-0007` state variable such that its new value is an increment of its old value.
   (c) occur a drill dump product.

2. be no updates in the remainder of the log to the state variables `CMD-0001` and `CMD-0009`. The `{ ... }` brackets at the outermost level indicate that there is no ordering imposed among the two event negations and the ordered `[ ... ]` sequence.

5.3 Specification Visualization

LogScope patterns are translated to data parameterized automata, as described in [3, 4]. This automaton language forms a subset of the more general rule-based Ruler language [5, 6, 4]. These automata can be visualized using GraphViz [10]. Figure 8 shows the visualization of the automaton for the second property. The interpretation should be self-explanatory with the following remarks: states can be parameterized with data, downwards arrow pointed states indicate non-final states that have to be left before the end of the log is reached, black states are error states, and the upwards pointed triangle represents an AND-node: all the sub-trees have to lead to success. The visualization of the automata generated from LogScope patterns turned out to be very useful as a mechanism for test engineers to understand the exact meaning of their patterns. We find that using a textual, as opposed to graphic, language makes it easier for testers who are often experienced coders to quickly formulate properties (and to auto-generate them from other sources, such as spreadsheets and engineering files), but that visualization is critical for quick understanding of specification semantics.

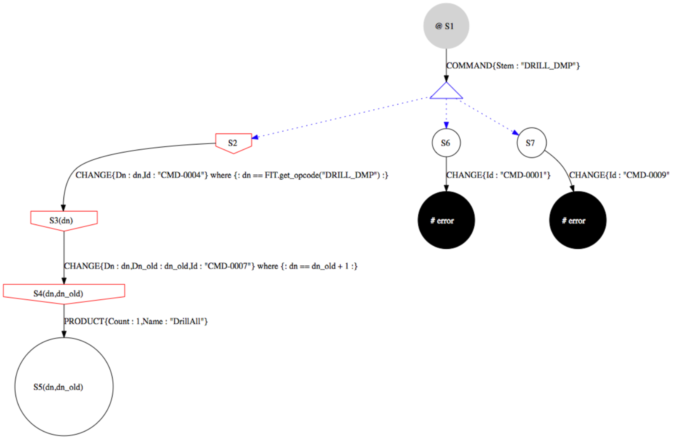
@ S1
COMMAND{Stem : "DRILL_DMP"}
S2
S6
S7
CHANGE{Dn : dn,Id : "CMD-0004"} where {: dn == FIT.get_opcode("DRILL_DMP") :}
CHANGE{Id : "CMD-0001"}
CHANGE{Id : "CMD-0009"
# error
# error
S3(dn)
CHANGE{Dn : dn,Dn_old : dn_old,Id : "CMD-0007"} where {: dn == dn_old + 1 :}
S4(dn,dn_old)
PRODUCT{Count : 1,Name : "DrillAll"}
S5(dn,dn_old)

Figure 8: Automaton for DRILL_CONSEQUENCES.

5.4 Running LogScope

The application of LOGSCOPE to the log in Figure 6 and the specification in Figure 7 causes two violations the `DRILL_CONSEQUENCES` property to be detected, both stemming from the fact that a change occurred to the state variable `CMD-0009` (which was not allowed - error 1) instead of to the state variable `CMD-0007` (which was required - error 2). The error message for error 1 is shown in Figure 9. The error message indicates that a `CHANGE` of state variable `CMD-0009` was observed while the monitor was in state `S7` (see Figure 8). An error trace consisting of events involved in the error is listed. These are the events that caused the monitor to transition between its states; the error trace excludes events not relevant for the error.

```
*** violated: by event 4 in state:

state S7 {
  CHANGE{Id : "CMD-0009"} => error
}
with bindings: {}

by transition 1 :
  CHANGE{'Id': "CMD-0009"} => error

--- error trace: ---

COMMAND 1 {
  OBJ_TYPE := "COMMAND"
  Stem := "DRILL_DMP"
  Type := "FSW"
  Number := "1"
  Time := 3700393
}

CHANGE 4 {
  Dn := 101
  OBJ_TYPE := "CHANGE"
  Dn_old := 100
  Id := "CMD-0009"
  Time := 44937474
}
```

Figure 9: Error message.

As additional information, LOGSCOPE prints out some statistical information for each property describing what events that were evaluated in order to determine its correctness. In the case of the `DRILL_CONSEQUENCES` property it concerns one command and two change events:

```
Statistics {
  COMMAND :
      {'Stem': 'DRILL_DMP'} -> 1
  CHANGE :
      {'Dn': 42, 'Id': 'CMD-0004'} -> 1
      {'Id': 'CMD-0009'} -> 1
}
```

5.5 Testing the MSL Software

As discussed above, the first trial of LOGSCOPE, initiated at the request of MSL software management, checked the behavior of an enhanced version of the command dispatch and success protocol shown above, for commands issued in randomly generated groups of 400. This was an attempt to duplicate the results of a hand-coded basic command system regression pre-dating the LOGSCOPE tool. Using LOGSCOPE enabled us to experiment with a more thorough specification of command behavior, leading to the discovery of a previously unknown fault in flight software that resulted in duplicated success EVRs.

The next major application, generation of a specification from tests of the power module, was initiated by a test engineer. The test engineer replaced previously unreliable on-the-fly queries to the ground software with code to generate a specification for each tested behavior, producing hundreds of custom specifications for as many flight software calls. Enabling this automatic generation was a primary motivation for some language features, including `upto` scopes. Again, log analysis revealed several faulty behaviors in the power and command modules, and increased awareness of subtle issues with the timing of channel telemetry. Test engineers responsible for the file verification system (FVS, used when ground uplinks files to the spacecraft) and the PYRO system (used to fire pyros) also replaced hand-scripted test code with LOGSCOPE specifications and helped to design (and then used) a concrete learning facility in the case of the PYRO subsystem. Concrete learning gives LOGSCOPE the ability to take a test and generate a specification from the test, abstracting away the precise timing of events but preserving ordering.

Developing the tool in collaboration with the test engineers (and the larger MSL flight software team) has maximized LOGSCOPE's utility, and can serve as a model for the introduction of formal specification methods in other software efforts. More importantly, it seems clear that the tool can improve the test team's productivity and result in better-tested flight software, as suggested by the interest of some JPL managers in using this approach for other projects, and the receipt of the Mariner Award.

6. FROM MOCK-UP TO LOGSCOPE

The following is a concise narrative of LOGSCOPE's evolution from the original mock-up specification in Figure 5, provided by engineers.

6.1 The Mock-up

In the original script a property conceptually has the following form: *event* ⇒ *consequence*, meaning: if an *event* occurs in the log at a certain position, then the remainder of the log shall match the pattern indicated by the *consequence*. The *consequence* itself is a list of events, each indicated via an event argument as either positive (shall occur) or negative (shall not occur). There is no intended ordering of the events — it is essentially an $\wedge$ (and) of event constraints. Events can carry arguments with some informal indication of a relation between arguments (as in: '`contains opcode` `of last immediate command`'). Parameters can be referred to by name (as in '`id:CMD-0007`'), although some parameters are referred to by position (as '`1`' and '`*`' in: '`prod(name:DrillAll,1,*)`', the last argument indicating a parameter of no interest).

6.2 LogScope Version 1

The first version of the LOGSCOPE language introduced the distinction between ordered and unordered consequences, using square brackets to represent an ordered collection of events (including negated events), and curly brackets to represent un-ordered collection (a logical $\wedge$ essentially) corresponding to the original intention in the mock-up. The distinction is common in temporal logic and regular expressions, although our use of brackets enclosing a list of events for ordered as well as un-ordered is less traditional. The uniformity might make specification writing easier. We avoided the general temporal operators (such as *always*, *eventually* and *until*) as they are not "engineer-friendly." The sequencing operator `[...]` (inspired by regular expressions) seems easier to use than the nested *until* expressions of temporal logic. Note, however, that there is no need for expressing *"don't care"* constraints for events in between events of interest. Hence we can write '$\texttt{[}e_1\texttt{,}e_2$ `]`' instead of '$e_1\texttt{;true*;}e_2$'. We only allowed negation of events, with a prefix operator replacing the positive and negative arguments in the mock-up to be better aligned with logic. Event parameters were formalized, and only referenced by name, in order to handle "big" events with many parameters. During the initial design engineers also asked for the possibility to express certain constraints on the values of fields, such as testing whether a bit in a bit-field was set or not. Special indexing constructs were introduced to handle this. Properties were finally translated into parameterized state machines, which themselves could be used for specification. This translation as well as the data parameterization was inspired by the RULER system [5, 6]. A parameterized automaton is really a simplified RULER system.

6.3 LogScope Version 2

After the first version engineers asked for two critical improvements. The first was the ability to limit the scope of a property, e.g., to express that some events should or should not occur until some other event (typically the issuing of another command). This lead to the `upto` construct. The second was the ability to test that a string value of a parameter contained a particular substring. We introduced a general solution for testing and relating contents of parameters: the `where` construct, which calls a PYTHON predicate, and furthermore allows PYTHON program text to be defined in specifications. This was a generalization of the indexing construct mentioned above. An option for executing PYTHON code with side effects when certain events occur was also introduced, inspired by early requests to be able to count and perform various statistics.

7. CONCLUSIONS

A central message of this paper is that automated post-run log analysis using formal specifications can be a light-weight approach to introducing formal methods in a software development effort even when engineers are wary of the time commitment required by formal approaches. Using logs already produced by the software makes such an entry much easier, as additional code instrumentation is not required. The LOGSCOPE specification language appeared easy to learn and was sufficient for expressing many realistic properties of a very complex software system. The language is general since it is based on the simple assumption that a log is a sequence of events, where an event is a named record: this can be made to hold for essentially any system. Future work includes further development of the LOGSCOPE language and tools, as well as additional applications. Concerning the specification language, our aim is to integrate with the RULER language [5, 6, 4] and optimize the way in which the monitoring algorithm handles data values.

Thanks are due to members of the MSL team, including Chris Delp, Dave Hecox, Gerard Holzmann, Rajeev Joshi, Cin-Young Lee, Alex Moncada, Cindy Oda, Glenn Reeves, Lisa Tatge, Hui Ying Wen, Jesse Wright, and Hyejung Yun.

8. REFERENCES

[1] `http://mars.jpl.nasa.gov/msl`.

[2] J. H. Andrews and Y. Zhang. General test result checking with log file analysis. *IEEE Transactions on Software Engineering*, 29(7):634–648, 2003.

[3] H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *AIAA Journal of Aerospace Computing, Information and Communications*, 2010. To appear.

[4] H. Barringer, K. Havelund, D. Rydeheard, and A. Groce. Rule systems for runtime verification: A short tutorial. In S. Bensalem and D. Peled, editors, *Proc. of the 9th International Workshop on Runtime Verification (RV'09)*, volume 5779 of *LNCS*, pages 1–24. Springer, 2009.

[5] H. Barringer, D. Rydeheard, and K. Havelund. Rule Systems for Run-Time Monitoring: from Eagle to RuleR. In *Proc. of the 7th International Workshop on Runtime Verification (RV'07)*, volume 4839 of *LNCS*, pages 111–125, Vancouver, Canada, 2007. Springer.

[6] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *Journal of Logic and Computation, doi: 10.1093/logcom/exn076*, 2008.

[7] M. Bennett, R. Borgen, K. Havelund, M. Ingham, and D. Wagner. Development of a prototype domain-specific language for monitor and control systems. In *IEEE Aerospace Conference, Big Sky, Montana*, March 2008.

[8] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze. Using symbolic execution for verifying safety-critical systems. In *European Software Engineering Conference/Foundations of Software Engineering*, pages 142–151, 2001.

[9] L. Dillon, G. Kutty, L. Moser, P. M. Melliar-Smith, and Y. S. RamaKrishna. A graphical interval logic for specifying concurrent systems. 3(2):131–165, 1994.

[10] GraphViz. `http://www.graphviz.org`.

[11] R. Mateescu and D. Thivolle. A model checking language for concurrent value-passing systems. In *The 15th international symposium on Formal Methods (FM 2008)*, volume 5014 of *LNCS*. Springer, May 2008. Turku, Finland.

[12] S. Squyres. *Roving Mars: Spirit, Opportunity, and the Exploration of the Red Planet*. Hyperion, 2005.

[13] M. Vardi. From Church and Prior to PSL. In *25 Years of Model Checking: History, Achievements, Perspectives*, 2008.