Let’s Look at the Logs: Low-Impact Runtime Verification

Alex Groce
Jet Propulsion Laboratory
California Institute of Technology
alex.d.groce@jpl.nasa.gov

Klaus Havelund
Jet Propulsion Laboratory
California Institute of Technology
klaus.havelund@jpl.nasa.gov

Margaret Smith
Jet Propulsion Laboratory
California Institute of Technology
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.

ABSTRACT
Runtime verification as a field faces many challenges, perhaps most obviously the need to keep instrumentation overheads low (especially in the real-time critical systems that might be most important to monitor), and the general difficulty of devising expressive but user-friendly specification languages. In this paper, we show that for many systems, in-place logging and telemetry provides a satisfactory basis for “runtime” verification, where the overhead is already included in system design. While this approach prevents the autonomous reaction to problems possible with traditional runtime monitoring, it provides a powerful tool for test automation, control, and debugging — in our case, analysis of spacecraft telemetry by ground operations teams. The “event-based” viewpoint natural to log analysis helps us address the second challenge, with an expressive pattern language inspired and developed in collaboration with test engineers at NASA’s Jet Propulsion Laboratory. We present case studies from use of our tool by flight software test engineers at NASA’s Jet Propulsion Laboratory. We present case studies from use of our tool by flight software test engineers for the Mars Science Laboratory mission at JPL.

Categories and Subject Descriptors
D.2.1 [Software Engineering]: Requirements/Specifications;
D.2.1 [Software Engineering]: Testing and Debugging—
Monitors, Tracing; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—Temporal logic

1. INTRODUCTION
Runtime verification encompasses the discipline of checking execution traces against formal specifications. Two of the key problems are: (1) how to obtain execution traces with minimal impact on observed software, and without overburdening the user with the need to implement instrumentation and (2) how to design a specification language that is expressive yet user-friendly. This paper reports on an experiment addressing both these issues, a combination that we call “low-impact” (in terms of impact on the software and burden on users) runtime verification. In summary, the paper argues that analyzing log information removes the instrumentation burden in the sense that logs usually are already produced by most critical software systems. A log is a recorded sequence of events. We argue that systematizing logging can be of additional benefit. Second, we introduce a textual temporal-logic-inspired pattern language over events for specifying properties of logs. This language supports data parameterization, essential for monitoring logs (events typically carry data). Patterns are translated into data parameterized \( \forall \)-automata forming an interesting and useful subset of the textual RULER language [6, 8, 7]. Users can mix patterns with more expressive automata in a specification. Parameterized automata are visualized using the GraphViz tool [13]. Our system additionally offers preliminary support for automatically constructing (learning) specifications from example logs. The resulting monitoring framework, LogScope, is written in Python, and is developed to support engineers testing the flight software for NASA’s next Mars rover mission, the Mars Science Laboratory (MSL) [1], developed at the Jet Propulsion Laboratory (JPL). In this sense the work represents an instance of the often sought, but rarely found, marriage between theory and practice in formal methods.

The MSL mission’s goal is to put the so-far largest rover (the size of a compact car) on Mars for continued exploration of the red planet. It is scheduled to launch in 2011. A flight software team peaking at approximately 30 programmers develops the software for controlling the Rover Compute Element (RCE), which controls all stages of the integrated 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. LogScope was developed to support this team, and is the result of their requirements as well as our research ideas. The MSL flight software produces rich log information, which is stored in SQL data bases (one database per log). A log is in essence a sequence of 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). Such logs are usually far too large to be effectively analyzed by humans. Traditionally, these logs have been analyzed by writing scripts, with prop-
errs, properties coded up in **Python**. Such scripts are time consuming to produce and result in difficult-to-read “specifications” that hinder communication, maintenance, and specification-sharing and reuse. Runtime verification of logs using formal specifications is a solution that has proven to be a “Trojan horse” for introducing formal methods into a high-profile NASA flight project.

1.1 Contributions and Related Work

The contribution of this work is a data-parameterized temporal pattern language, as well as its translation into parameterized *∀*-automata (also usable in specification). A second contribution is our method for injecting this technology into a NASA flight mission. The LogScope system has specifically been influenced by the Ruler system [6, 8, 7]. In particular, the automata language conceptually forms a subset of *∀*-automata. LogScope can be seen as an adaptation of Ruler to the specific needs of JPL’s MSL project: adding temporal logic, making some adjustments, and reprogramming in **Python**. LogScope has, however, also been influenced by the state machine-based systems *RCAT* [20] and *RMOR* [14]. A substantial amount of work has been done on runtime verification [2] within the last decade, e.g. [16, 12, 15, 5, 3, 10, 11], much of which has guided the design of Ruler as well as LogScope. The design of the PSL [22] language is also relevant, in its ability to sugar-coat temporal specifications and mix different notations.

2. TOOL DEVELOPMENT PROCESS

The LogScope tool and pattern language grew out of the needs of engineers. In fact, the earliest effort to “mock-up” a specification came from an MSL test engineer, not the runtime-verification research team:

```plaintext
look:DRILL_DMP\ evr(CMD_EVR_VC1_CMD派遣,positive)\ evr(CMD_EVR_CMD派遣成功,positive)\ evr(CMD_EVR_CMD派遣失败,positive)\ chan(id:CMD-0004,positive,0x03e0,contains the opcode of the last immediate command dispatched)\ chan(id:CMD-0007,positive)\ chan(id:CMD-0001,negative)\ chan(id:CMD-0009,negative)\ prod(name:DrillAllParms,1,*)
```

The core elements of our current pattern language — the **events**, all originate in the telemetry system of the Mars Science Laboratory flight and ground software. These events correspond to the generalized view of a “system” shown in Figure 1. An operations team on the ground issues **commands** to the spacecraft, e.g. DRILL_DMP in the sample specification. The spacecraft responds to commands by changing its state (transitions, which are not, on their own, visible). The responses of the spacecraft are observed via three kinds of telemetry events: (1) **EVRs**, essentially **printfs** in flight software used to indicate state changes (2) **EHA channels** that provide a snapshot of current spacecraft state and (3) **data products**, the “outputs” of the spacecraft downlinked to the ground, including engineering telemetry, images, and data from science instruments (these are shown as evr, chan, and prod).

Figure 1: Observable events of a system

To a large extent, the development of the pattern language that followed was a process of taking this early draft and preserving its essential elements while expanding expressive power and readability, guided by usage, engineer requests, and language design considerations. For example, we preserved the structure of a guard which when triggered requires a series of expected event responses, an $A \Rightarrow B$ form, where $A$ is an event and $B$ is a sequence of events. The clear need for more complex specifications inspired elaboration in our first prototype: in the original mock-up, the positive events in $B$ were required to occur in order (allowing all non-negative events in between). The first implementation prototype introduced a way to distinguish ordered lists from unordered sets of events, which proved critical to real specifications.

The original test infrastructure, as suggested by the FIT team, provided an event registry, a **Python** library with methods allowing test scripts to create, alter, and delete online monitors for telemetry events. In principle, the registry provided a sufficient set of tools for analyzing tests at runtime, a specification and debugging mechanism well-integrated with established testing procedures. In practice, the registry approach was seldom adopted by test engineers (and proved frustrating when it was used) for two primary reasons:

1. **Ordering and timing of events**: Events on the spacecraft are not downlinked to the ground system in a strict FIFO ordering. Different channels, data products, and EVRs have different priorities, and the “Earth receive time” ordering of two events will often be the opposite of their ordering by spacecraft time. Test engineers were forced to either introduce lengthy delays after each test step, to avoid confusion...
of results, or to build very complicated logic to disentangle events arriving out-of-order and "recreate" a linear chronology of events (which is sometimes only possible after test termination). A strategy of pausing long enough to allow telemetry to arrive proved brittle, as timeouts fluctuated with each software release.

2. Confusion of test execution and test evaluation: One of the original motivations for the online event registry was that it seamlessly integrated with the idea of tests as PYTHON scripts with logic, looping, and other programming-language features. Unfortunately, test scripts that were clearly readable when their task was limited to commanding the spacecraft became extremely difficult to follow when test execution was interlaced with test evaluation.

Neither of these problems is particular to MSL. The first problem is generic to any distributed system in which constructing an event timeline is non-trivial before event queues clear, and the second problem is a general observation about test case readability. Test engineers reacted to these problems by making little use of the registry, despite having originally requested its features! An automated analysis of all test scripts developed by the team, conducted shortly before the adoption of the LogScope approach, showed that most registry features were never used and others were only used in the most basic fashion, or after test termination. How, then, were test engineers evaluating tests? In many cases, they were hand-scripting a post-test analysis of all telemetry (ad-hoc construction of very limited logs). A few simple checks were routinely performed online, but more complex analysis was often delayed until the test was complete, and chronological confusion could be partially avoided by counting the total expected responses.

Team management (also active in writing tests) recognized that building ad-hoc test-specific "log analysis" systems was not an effective use of mission time, and that scripting logic only partially addressed both problems with the registry. The DRILL_DMP mock-up emerged after a series of team meetings and informal discussions, informed by our ideas of log analysis. MSL flight software management additionally suggested that a unified logging scheme would be useful to developers outside the test team. After an initial trial run in which the research team replicated the hand-scripted results of random command regressions, the FIT team began use of prototype versions of the logging system, replacing hand-scripted test evaluation code, as described in more detail in Section 7. The research team suggested the use of square and curly braces to indicate lists and sets of events, respectively, and the use of bindings for data values. Other features, including indexed data field (e.g. bit-vectors), SQL query filtering of events, up to scoping of specifications (Section 4.5), and concrete learning (Section 6.1), were requested by the test team. Starting with a syntax and informal semantics designed by the test team ensured that the language would be powerful enough to support test needs and simple enough that it would actually be used.

3. LOGS

The first step in analyzing the behavior of the MSL flight system is to produce a log — an ordered sequence of events.

Figure 2: A simplified log
Our log is based on a database view of commands and spacecraft telemetry maintained by JPL ground software. The ground software will be used by mission operations to communicate with the spacecraft and rover, and is now used both in MSL hardware testbeds and software simulation. The ground system stores events as entries in a SQL database (an approach also considered in runtime verification literature [17]). Different kinds of events in the database provide different fields (e.g., EVRs have a message but no value, while channels have a value but no message). Rather than forcing a single representation on these event types, we allow different events in the unified log to have varying fields, but ensure that each event has (1) a type and (2) a timestamp. The type is used in specifications as shown below, and the timestamp allows us to order the event sequence.

Figure 2 simplifies part of a log produced during testing of the power sub-system. The example includes commands, EVRs, and a channel value. The fields of the events are shown in brackets, except for the object type, which appears before the bracket. In addition to fields present in the original database, our system 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 always derived. Events that take place on the spacecraft include a spacecraft event time (omitted here) that establishes a canonical order. However, command events originate from the ground, and include only a transmission time. We establish a uniform chronology by extracting the time a command is dispatched on the spacecraft from the EVR events — the Time field of COMMAND 7308 is extracted from the message in EVR 7309. This leads to a feedback between logging and telemetry design on the spacecraft. Use of our tool by test engineers has increased to a feedback between logging and telemetry design on the spacecraft from the EVR events — the Time field of COMMAND 7308 is extracted from the message in EVR 7309. This leads to a feedback between logging and telemetry design on the spacecraft. Use of our tool by test engineers has increased to a feedback between logging and telemetry design on the spacecraft.

From the point of view of LogScope, a log is simply an ordered list of structures (implemented as a list of dictionaries in Python). All MSL-specific aspects of the log are encapsulated in the logging tool, making the later tools in the chain easily adaptable to any system producing such a log. We expect this to be most useful for other JPL missions, but it should also be applicable to operating systems, web servers, and any other systems that produce event-based logs.

4. THE PATTERN LANGUAGE
Monitors are written in the LogScope specification language. A specification consists of one or more specification units, each of which is either a temporal logic pattern, or a parameterized automaton:

spec ::= spec-unit+
spec-unit ::= pattern | automaton

In this section we focus on temporal logic patterns. Patterns are automatically translated into parameterized automata.

4.1 Simple Response Patterns
Consider as an example the following informally stated property: \( R_1 \): “Whenever a flight software command is issued, then eventually an EVR should indicate success of that command”. Before we can formalize this property, it needs to be refined to refer to the specific fields of events. The following is such a refinement:

\( R'_1 \): “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 \( y \).”

The Number field is a sequence number indicating the order in which commands are received by the spacecraft for dispatch. Subsequent commands should have increasing numbers. Events related to command execution have the number of the command. In our language, this property reads:

\[
\begin{align*}
\text{pattern P1:} \\
\text{COMMAND(Type:"FSW", Stem:x, Number:y} \} => \\
\text{EVR\{Success:x, Number:y\}}
\end{align*}
\]

This pattern (pattern is a keyword) has the name P1 and states that if a command is observed in the log file at a position \( i \), with the Type field having the exact string value "FSW", the Stem field having some value \( x \), and the Number field having some value \( y \); then later in that log file, at a position \( j > i \), an EVR should occur with a Success field having \( x \) as value and a Number field having \( y \) as value. Informally, we can explain the semantics of this formula in terms of the following formula in a Linear Temporal Logic (LTL) [19] with quantification over data variables:

\[
\forall x, y \bullet \square \text{COMMAND(Type:"FSW", Stem:x, Number:y)} \Rightarrow \diamond \text{EVR\{Success:x, Number:y\}}
\]

The temporal operators \( \square \) and \( \diamond \), as well as the universal quantification \( \forall \), are implicit in LogScope’s pattern notation. The pattern has the form:

\[
\begin{align*}
\text{pattern ::=} \\
\text{'pattern' NAME ':' event '=>' consequence}
\end{align*}
\]

\[
\begin{align*}
\text{consequence ::=} \\
\text{event} \\
| '!' event \\
| '[' consequence_1,...,consequence_n ']', \\
| '{' consequence_1,...,consequence_n '}'
\end{align*}
\]

This example shows the simplest case, where the consequence is an event. The other alternatives will be explained
in the following subsections. The event triggering the pattern is a command event here, but can be any kind of event. Each event is constrained (between {...} brackets) by zero or more constraints, each consisting of a field name (without quotes), and a range specification. We saw two forms of range specifications: the string "FSW" for the field Type and the names x and y for the other fields. A string constant represents a concrete constraint: the field in the event has to match this value exactly (by PYTHON equality ==). One can also provide an integer as such a concrete range constraint. Ranges can also be multi-valued, such as integer intervals, as will be shown below.

A unquoted name (x and y in this case) occurring as a range indicates an (at specification time) unknown value. The first occurrence of such a name in a pattern is binding: it will be bound to the value of the corresponding field in the matching event. Any subsequent occurrences of this name in the pattern are now constraining: for a match to occur, the corresponding fields now have to have the values these names were bound to by the triggering event. The log in Figure 2 satisfies this specification for the first command, as the command is matched by a success. The property fails for the second command.

### 4.2 Negation of Events
A consequence can also be the negation ('!') of an event. Suppose we want to state the following property: $R_2$: "Whenever a command is issued with the type FSW, the stem field having some value $x$, and the number field having some value $y$, then there should after not occur an EVR, with the field failure mapped to $x$ and the number field mapped to $y". We express this by the property (the log from Section 3 satisfies):

\[
\text{pattern P2:} \quad \text{COMMAND} \{ \text{Type:"FSW"}, \text{Stem}:x, \text{Number}:y \} \Rightarrow \\
\neg \text{EVR} \{ \text{Failure}:x, \text{Number}:y \}
\]

### 4.3 Composite Consequences
We have seen that the consequence of a pattern can be an event (pattern P1) or the negation of an event (pattern P2). There are two more forms: ordered and unordered sequences of consequences (a recursive definition). The square brackets [...] indicate that the consequences should occur in exact order, while the curly brackets {...} indicate that they may occur in any order. The symbols are chosen for their frequent use in literature to represent ordered lists and unordered sets. As an example, consider the following requirement: $R_3$: "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 (with that number), and no failure before that, and no more successes of that command (exactly one success)"

Requirement $R_3$ is in Figure 3 illustrated as a timeline, much like those in Timeedit [21], showing events that could/should occur, and constraints on events that should not occur. This property can be stated formally as follows, in a form that very closely reflects the time line.

![Figure 3: Requirement $R_3$ expressed as a timeline](image-url)

The consequence consists of a sequence (in square brackets [...] of (sub) consequences, in this case events and negations of events. The ordering means that as a response to the command the dispatch should occur before the success, and the negations state what should not happen in between the non-negated events.

The following quantified LTL formula represents this property. It is suggestive as to the extra complication that LTL's until operator introduces, and why the pattern language might be easier to use for engineers:

\[
\forall x,y. \\
\Box \text{COMMAND} \{ \text{Type:"FSW"}, \text{Stem}:x, \text{Number}:y \} \Rightarrow \\
\neg \text{EVR} \{ \text{Success}:x, \text{Number}:y \} \\
\wedge \Box \neg \text{EVR} \{ \text{Failure}:x, \text{Number}:y \} \\
\wedge \Box \neg \text{EVR} \{ \text{DispatchFailure}:x \} \\
\wedge \Box \text{EVR} \{ \text{Dispatch}:x, \text{Number}:y \} \\
\wedge \Box \neg \text{EVR} \{ \text{Failure}:x, \text{Number}:y \} \\
\wedge \Box \neg \text{EVR} \{ \text{Success}:x, \text{Number}:y \}
\]

As an example of an un-ordered arrangement of events, consider the following (perhaps less meaningful) relaxation of the above stated property: $R_3$: "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 suc-
cess there should not follow another success for that same command and number". We formalize this as:

\[
\text{pattern P4 :}\quad \text{COMMAND}\{\text{Type: }\text{"FSW"}, \text{ Stem: } x, \text{ Number: } y\} \Rightarrow \\
\{ \text{EVR}\{\text{Dispatch: } x, \text{ Number: } y\}, \\
[ \text{EVR}\{\text{Success: } x, \text{ Number: } y\}, \\
! \text{EVR}\{\text{Failure: } x, \text{ Number: } y\} ], \\
! \text{EVR}\{\text{DispatchFailure: } x\}, \\
! \text{EVR}\{\text{Success: } x, \text{ Number: } y\} \}
\]

The curly brackets \{...\} indicate an un-ordered collection of consequences (a Boolean \(\lor\) effectively). This corresponds in this case to four parallel timelines, all of which must be satisfied. The fact that they are un-ordered means that the non-negated events can occur in any order (but must occur eventually), and negations have to hold to the end of the log. Nested inside the unordered \{...\} construct there is, however, an ordered sequence expressing that after a success there should not occur another success. As the grammar for consequences suggests, ordered and unordered collections of consequences can be mixed arbitrarily.

### 4.4 Event Predicates

Events can be constrained with predicates. The following pattern expresses the requirement \(R_5\): “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 \(z \leq y\).”

\[
\text{pattern P5 :}\quad \text{EVR}\{\text{Success: } _, \text{ Number: } y\} \Rightarrow \\
! \text{EVR}\{\text{Success: } _, \text{ Number: } z\} \text{ where } |z \leq y|
\]

Note that this formula ignores what commands succeed (the underscore \_). The constraining predicate is: \(|z \leq y|\), and is a general PYTHON expression enclosed by the symbols |...|. This expression can refer to all of Python’s immediately available concepts in a addition to a library of predicates. Should this not suffice, it is possible to import PYTHON libraries or directly define predicates in the specification. The following specification defines a Python predicate within(t1,t2,max) which checks that the two time points t1 and t2 are less than max time units apart.

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

\[
\text{pattern P6:}\quad \text{COMMAND}\{\text{Type: }\text{"FSW"}, \text{ Stem: } x, \text{ Number: } y, \text{ Time: } t1\} \\
\text{where } \{x\text{.startswith("PWR_")}\} \\
\Rightarrow \\
\text{EVR}\{\text{Success: } x, \text{ Number: } y, \text{ Time: } t2\} \\
\text{where within(t1,t2,10000)}
\]

This means that positive events such as the dispatch have to occur before the next flight software command, and negative events, such as failures, are only checked for (forbidden) up to the next flight software command. The syntax is now expanded to include an optional scope-specification:

```python
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{Success: x, Number: y} }
```

This predicate is then used to check that power commands (having names starting with "PWR") succeed within maximally 20000 time units. The \(x\text{.startswith(y)}\) method is one of PYTHON’s built-in string methods, returning true if the string \(y\) is a prefix of the string \(x\).

Predicates can be composed using the traditional Boolean operators: and, or, not. It should briefly be mentioned that there are shorthands for certain predicates that have occurred frequently, such as intervals and bit operations. It is thus possible to write an event of the form: EVR{Number1 : [1000,2000], Number2 : {0:0,1:0}}. Number1 should be an integer in the interval 1000 to 2000, and Number2 should be a bit-vector, where bit 0 is 0 and bit 1 is 0. This form of indexing also works on list values, strings (in both cases indexes would be integers, counting from the left) and dictionaries (maps).

### 4.5 Scopes

In some cases one may want to limit the scope in which a pattern is checked, by providing an additional scope-terminating event. Without such limitations a pattern holds from the point at which its trigger event is matched until the end of the log. As an example, one may want to check that a particular command results in a particular set of events to occur, and some other events not to occur, \(up to the next command being fired\). Consider for example pattern P4 in Section 4.3. According to the semantics whenever a flight software command is detected in the log, the consequence is checked on the rest of the log, to its end. That is, any required event such as the dispatch can occur anywhere in the rest of the log, and negative events, such as failures are checked on the remaining log. We might, however, limit these checks to be performed up to the next flight software command (satisfying \(\text{COMMAND}\{\text{Type: }\text{"FSW"}\}\)). This is done by adding the scope delimiter ‘upto COMMAND{Type: "FSW"}’ as follows:

```python
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} }
```

5. THE AUTOMATON LANGUAGE

LOGSCOPE also allows testers to write properties as parameterized $\forall$-automata, a more expressive, but lower-level, language than the pattern language. Patterns are automatically translated to automata. The automata language forms a subset of the powerful rule-based RULER specification language [6, 8, 7], with some modifications (of a syntactic flavor). An automaton is expressed in terms of states and transitions between states triggered by events. Events are exactly as in patterns. Just as events can be parameterized with values, states can be parameterized, carrying values produced by incoming transitions. Also, an automaton can be in several states at the same time, all of which have to lead to success. Automata, hand-written as well as translated from patterns, can be visualized with GraphViz [13]. Automata are stored in GraphViz’s dot format for diagrams. Automata visualization has proven useful for users of the pattern language when trying to ensure their patterns express what is intended — we believe that textual development with graphic visualization is a very effective method for experienced test engineers.

We illustrate the automata language by presenting the automata for patterns P3 and P4. The automaton corresponding to pattern P3 is shown in Figure 4, visualized in Figure 5. The automaton consists of four states: S1 – S4, where the first mentioned state is the initial state (also indicatable via keyword, as there may be several initial states). There is one transition exiting S1: this transition is triggered by a flight software command, binding x and y, and entering state S2(x,y) with x and y now bound to the actual values in the matching event — an example of a state parameterized with data. The S1 state is an always-state, meaning that it is always active, waiting for any command observed. As such several instances of S2(x,y) can be active at any point in time, each binding different values to x and y.

States S2 and S3 are so-called hot states, meaning that instances of these states should disappear before the end of the log is analyzed. If not it is regarded as an error. They are used to model (bounded) liveness properties: that some event must occur before the end of the log. In this case, S2 for example represents the property that a dispatch should occur (with no previous dispatch failure). The first exiting transition is matched by any EVR with a DispatchFailure field with a value equal to the parameter x. Similarly for the second transition. There are two special states: the error state (indicating error) and the done state (indicating successful termination of a branch of the automaton). The state S4 is a normal state (not an always-state and not a hot state), meaning that it is acceptable to finish monitoring in that state. Its purpose is to check that there are no further successes after the first, for a particular command x with number y.

Automaton visualization uses different symbols for the different states: a gray state represents an initial state, a white circle represents a normal state, a black circle represents an error state, and a downwards-pointed red (in case colors are available) pentagon represents a hot state. The $\exists$ symbol in state S1 indicates an always-state.

The pattern P4 is translated into the automaton in Figure 6, visualized in Figure 7. This automaton expresses that after a command, we want to see a dispatch (state S2) and a success (state S3), and after that no further success (state S6), but in no particular order, and that there should not at any time be a dispatch failure (state S4) or a failure (state S5). The transition of state S1 enumerates several target states, all of which become active when the transition fires, and all of which must lead to success: no error states should be entered and at the end no hot states should be active. The states

\[
\text{pattern ::= 'pattern' NAME ':': event '=>' consequence ['upto' event]}
\]

\[
\text{automaton A3 \{ always S1 { COMMAND\{Type:"FSW", Stem:x, Number:y\} => S2(x,y) \}}
\]

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

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

\[
\text{state S4(x,y) \{ EVR\{Success:x, Number:y\} => error \}}
\]

\[
5. \text{THE AUTOMATON LANGUAGE}
\]

\[
\text{LOGSCOPE also allows testers to write properties as parameterized } \forall \text{-automata, a more expressive, but lower-level, language than the pattern language. Patterns are automatically translated to automata. The automata language forms a subset of the powerful rule-based RULER specification language [6, 8, 7], with some modifications (of a syntactic flavor). An automaton is expressed in terms of states and transitions between states triggered by events. Events are exactly as in patterns. Just as events can be parameterized with values, states can be parameterized, carrying values produced by incoming transitions. Also, an automaton can be in several states at the same time, all of which have to lead to success. Automata, hand-written as well as translated from patterns, can be visualized with GraphViz [13]. Automata are stored in GraphViz’s dot format for diagrams. Automata visualization has proven useful for users of the pattern language when trying to ensure their patterns express what is intended — we believe that textual development with graphic visualization is a very effective method for experienced test engineers.}
\]

\[
\text{We illustrate the automata language by presenting the automata for patterns P3 and P4. The automaton corresponding to pattern P3 is shown in Figure 4, visualized in Figure 5. The automaton consists of four states: S1 – S4, where the first mentioned state is the initial state (also indicatable via keyword, as there may be several initial states). There is one transition exiting S1: this transition is triggered by a flight software command, binding x and y, and entering state S2(x,y) with x and y now bound to the actual values in the matching event — an example of a state parameterized with data. The S1 state is an always-state, meaning that it is always active, waiting for any command observed. As such several instances of S2(x,y) can be active at any point in time, each binding different values to x and y.}
\]

\[
\text{States S2 and S3 are so-called hot states, meaning that instances of these states should disappear before the end of the log is analyzed. If not it is regarded as an error. They are used to model (bounded) liveness properties: that some event must occur before the end of the log. In this case, S2 for example represents the property that a dispatch should occur (with no previous dispatch failure). The first exiting transition is matched by any EVR with a DispatchFailure field with a value equal to the parameter x. Similarly for the second transition. There are two special states: the error state (indicating error) and the done state (indicating successful termination of a branch of the automaton). The state S4 is a normal state (not an always-state and not a hot state), meaning that it is acceptable to finish monitoring in that state. Its purpose is to check that there are no further successes after the first, for a particular command x with number y.}
\]

\[
\text{Automaton visualization uses different symbols for the different states: a gray state represents an initial state, a white circle represents a normal state, a black circle represents an error state, and a downwards-pointed red (in case colors are available) pentagon represents a hot state. The $\exists$ symbol in state S1 indicates an always-state.}
\]

\[
\text{The pattern P4 is translated into the automaton in Figure 6, visualized in Figure 7. This automaton expresses that after a command, we want to see a dispatch (state S2) and a success (state S3), and after that no further success (state S6), but in no particular order, and that there should not at any time be a dispatch failure (state S4) or a failure (state S5). The transition of state S1 enumerates several target states, all of which become active when the transition fires, and all of which must lead to success: no error states should be entered and at the end no hot states should be active. The states}
\]
can be said to “execute in parallel”. The visualization of this automaton uses an upward pointing blue triangle (similar to an and-symbol $\land$) to describe a transition with multiple targets.

As mentioned earlier, the automaton language is more expressive than the pattern language. Recall that an event pattern has the form: \texttt{event => consequence}. That is, the antecedent is a single event. It is currently not possible to write a property of the form: \texttt{[E1,E2,E3] => E4}, with the intended meaning that if events \texttt{E1}, \texttt{E2} and \texttt{E3} occur, then \texttt{E4} should occur. This can, however, easily be expressed in the automaton language. The pattern language can of course be extended to also allow this form of property (work in-progress).

The specification language offers a number of additional features. These include a notion of success states, a dual to hot states. Occasionally it can be more convenient to express a property using success states: at least one of these must be reached by the end of the log analysis. States can also be defined as “step” states, meaning that in case the next event does not trigger one of the transitions, monitoring along that path is terminated. This is not an error, unless it means that some success state is not reached. This concept is specifically used during learning.

6. LEARNING
Writing specifications can be difficult and time consuming. The key problem consists of identifying what properties to check — even formulated in English prose. A working approach is to generate logs during one or more runs, and then look at (eyeball) the logs while trying to extract correct-behavior patterns. This process can be supported by tools in various ways. One extreme approach is to attempt to automatically learn a specification from one or more presumably “well behaved” log files, and then later turn this specification into a monitor to check subsequent log files as part of a regression test suite. If these differ, for example due to later code modifications, warnings highlighting the discrepancies are issued. This approach assumes that a user can judge whether an initial set of logs (those to be learned from) reflect correct program behavior, and is, of course not a generally sound approach, especially if logs contain thousands of events. If the initial logs are the result of erroneous program behavior, the learner will learn a wrong specification.

However, as long as error reports are considered with care, this technique is effective for catching bugs and “characterizing” software that is in a constant state of flux. \textsc{LogScope} provides some preliminary functionality supporting learning of specifications from logs. Two forms of learning can be considered: \textit{concrete learning} and \textit{abstract learning}. Only concrete learning has been implemented so far, but we will briefly discuss both.

6.1 Concrete Learning
During concrete learning \textsc{LogScope} learns the exact log files it observes up to equivalence on a set of field names that is provided to the learner, either from a default set or a set (for each kind of event) provided by the user. Two events are considered equivalent if they are equal \texttt{w.r.t.} these fields. The learner module, when fed several logs, will build an automaton which represents the set of all logs seen. Common
prefixes of the log will result in a single path through the automaton, which will eventually form a tree-shape (no loops). The leaf-states are success states (at least one success state must be reached before the end of the log), and the remaining states are step states (states that must be left in the next step, hence forcing complete conformance to the automaton, no extra events are allowed). It is possible to create a new automaton, learn from one or more logs, write back the so-far learned automaton to persistent memory, and then later further refine via learning or use it for monitoring. Concrete learning results in typically very large automata. The objective is in this case is similar to that of comparing log files with UNIX's `diff` command. Here one would possibly also attempt to remove irrelevant event fields, for example with UNIX's `grep` command. Using an automata-based approach, however, opens up the possibility of more advanced comparisons of logs.

### 6.2 Abstract Learning

In abstract learning mode one would want, for example, to learn high level properties about consequences of individual commands, yielding typically small, human readable, specification units, an idea similar to the approach of Perracotta [23]. E.g., for each command one can learn what subsequent events always occur between the command and the next command. This corresponds to introducing a scope for learning the consequences of a single command. One can learn the specific order in which subsequent events occur or ignore the order. One can learn minimal and maximal time periods in between events. We also hope to incorporate classic automata learning results [4, 9], though the challenge here is considerable: the language of input symbols is very large, it is not feasible to make precise language-inclusion queries as required by Angluin's algorithm, and we believe that the SAT problems for Biermann's approach might be too large to solve.

### 7. USAGE AND CASE STUDIES

#### 7.1 Running LogScope

LogScope is called from a Python application as in the following script, which first creates a log (typically by reading in a log generated by a running program), then creates an instance of an Observer object (given the location of the specification(s), possibly in a list, as parameter), and then applies the monitor method to the log:

```python
import logscope

log = ... # create a log
observer = logscope.Observer("$ISSTA/spec")
observer.monitor(log)
```

Executing the above Python script will generate a file with results as well as a set of dot-files visualizing the generated automata. The results of the monitoring can also be accessed from within the Python script by a `getResult()` method, so that they can be processed, e.g. as part of a regression test harness.

---

Our original log file in Figure 2 violates all properties presented in this paper, except P2, P5, and P6. This is summarized by the system as follows:

```
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
```

All these violations are caused by the dispatch failure (and subsequent lack of a dispatch and a success) of the last command issued (event number 9626). To simplify this presentation we will focus on property A4 presented in Section 5, and visualized in Figure 7. This property is similar to P4 and P7. A4 is violated 3 times: once for a safety property (a failure occurs) and twice for a (bounded) liveness property (neither a dispatch nor a success occurs). The error messages for the safety violation and the first liveness property (lack of dispatch) are listed below.

```
RESULTS FOR A4:

*** violated: by event 9627 in state:
state S4(x,y) {
  EVR{DispatchFailure:x} => error
} with bindings:
  {'y': '18', 'x': 'PTY_RUN_IMMEDIATE'}
by transition 1:
  EVR{DispatchFailure:'PTY_RUN_IMMEDIATE'} => error
--- error trace: ---
COMMAND 9626 {
  OBJ_TYPE := "COMMAND"
  Args := ['pwr_set_device(91,1,0)', 'TRUE']
  Number := "18"
  Stem := "PTY_RUN_IMMEDIATE"
  Time := 51708372934400
  Type := "FSW"
}

EVR 9627 {
  OBJ_TYPE := "EVR"
  name := "CMD_EVR_DISPATCH_VALIDATION_FAILURE"
  level := "COMMAND"
  Number := "18"
}
```
DispatchFailure := "PTY_RUN_IMMEDIATE"
Time := 51708372934499
message := "Validation failed for command
PTY_RUN_IMMEDIATE: number=18."
}

*** violated: in hot end state:

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

--- error trace: ---
COMMAND 9626 {
   OBJ_TYPE := "COMMAND"
   Args := ['pwr_set_device(91,1,0)', 'TRUE']
   Number := "18"
   Stem := "PTY_RUN_IMMEDIATE"
   Time := 51708372934400
   Type := "FSW"
}

The first error message explains the safety violation: that the dispatch failure event number 9627 triggers the error transition in the state S4, which is listed, including the values that are bound to the state parameters x and y. The error trace only contains the events that have made the automaton move: the PTY_RUN_IMMEDIATE command event, and the fatal dispatch failure event for that command. Since events are numbered it is easy to locate these events in the real log, which is also stored with event numbers.

The second error message explains the first liveness violation: that a dispatch does not occur. The state S2 is where the monitor rests at the end of the log. Since this is a hot state, this indicates a violation. The only event that moved the monitor w.r.t. this particular violation is the initial PTY_RUN_IMMEDIATE command, the only event listed in the error trace.

7.2 Application in MSL Testing
As described, our tool and language development was generally guided by the needs of the MSL test engineers. Our first trial run, initiated at the request of MSL software management, checked the core behavior of the command dispatch and success protocol shown above, for commands issued in randomly generated groups of 400 (an automated regression effort pre-dating our tool). Using LOGScope enabled us to experiment with the specification more than the test engineer using a hand-scripted analysis, leading to the discovery of a previously uncaptured error, duplicated success EVRs.

A more extensive application, initiated by the test engineer responsible, was the automatic generation of a specification from tests of the power module. The test engineer replaced previously unreliable on-the-fly queries to the ground software with code to generate a specification for each tested behavior. Enabling this automatic generation was a primary motivation for several language features, including pattern scopes. Again, log analysis revealed several faulty behaviors in the power and command modules, and revealed subtle issues with the timing of channel telemetry. The sample log in Figure 2 is taken from a power test (the full log contains over 11,000 events, including 107 flight software commands).

The 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. In the case of the PYRO module, the test engineer needed the capability to learn a canonical log and compare other logs against the known-correct result, a primary motivation for the concrete learning capabilities of the tool.

MSL test team management has proposed that all scripted test evaluation should be converted to use LOGScope, in order to simplify reaction to future changes in flight software, improve test maintainability, and enable automatic overnight regressions of flight software behavior. We believe that developing the tool in collaboration with the test engineers (and the larger MSL flight software team) has maximized the tool’s adoption and 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 has improved the test team’s productivity, and will result in better-tested flight software.

8. CONCLUSIONS AND FUTURE WORK
We have in this paper presented a log analysis specification language offering two forms of specifications: temporal logic patterns and \( \forall \)-automata, both of which can be parameterized with data, composite (lists, maps) or primitive. The automaton language forms conceptually a subset of the RULER language [6, 8, 7], limited to the concepts needed by MSL. In this sense, the automaton language presented here suggests a useful, and yet simple subset of RULER. The automata language is more expressive than the pattern language and can be used in rare cases where this extra expressiveness is required. Patterns are automatically translated to automata. To the best of our knowledge the translation from parameterized patterns to parameterized automata is a novel approach. A point of particular interest is that engineers find it very effective to write specifications in the pattern language and check their precise semantics in the automaton visualizations. The pattern language itself is also new, and interesting in the sense that engineers are able to learn it fairly quickly, and found that it was useful for expressing most realistic properties.

The decision to analyze logs turned out to likely be crucial to the adoption of formal specifications by a testing team at JPL. Logs were already produced by the system under test, and hence no instrumentation effort on behalf of the flight software engineers was required (and, therefore, problem with breaking expected real-time performance). This meant that the testing team could work more or less in isolation, without slowing the flight software engineers down. A great amount of research in runtime verification involves instrumentation issues. Automated code instrumentation is indeed important and interesting. However, it might not be a simple matter for software engineers to decide what to monitor and how to specify it. For example, specify-
ing point-cuts in aspect oriented programming [18] can be a challenge. On the other hand, inserting print statements in code is the world’s most common method for debugging and understanding programs [24].

Our recommendation is, therefore, to apply formal runtime verification to logs when automated code instrumentation becomes unpractical. Even un-structured printf logs can be made useful, though structured logging, with events as records, is generally preferable. We experimented with extracting events from un-structured logs, using regular expressions to define events from a text stream, e.g., in this example where command is bound to the number denoted by \( d+ \):

\[
\text{event success(command)} = \\
\text{\textquoteleft COMMAND (d+) \textquoteleft(.*\?\textquoteleft SUCCEEDED\textquoteright)}
\]

A further thought is to establish a formal connection between requirements engineering and logging, such that requirements become testable through runtime verification of logs. The common element is the event: requirements should be expressed as predicates on sequences of events, and logging should produce such sequences of events. Several of the JPL test engineers we interacted with came from a system engineering background, and their reactions suggest that event-sequences may be a very natural way for high-level engineers (not limited to software engineers) working on requirements to formulate many properties. Further future work includes merging the LogScope system with the RULER system [6] to obtain a unified system, in collaboration with other members of the RULER team\(^3\). On the application side, our goal is to give LogScope more exposure inside JPL and exploit its potential in other flight missions (and in MSL development as well as testing).

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

9. REFERENCES


\(^3\)The British Royal Academy of Engineering has provided an award to facilitate this collaboration.

Runtime Verification (RV’07), volume 4839 of LNCS, Vancouver, Canada, 2007. Springer.