Formal Analysis of Log Files

Howard Barringer[1]
University of Manchester, Manchester, UK

Alex Groce[2]
Oregon State University, Corvallis, Oregon, USA

Klaus Havelund[3] and Margaret Smith[4]
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, USA

Runtime verification as a field faces several challenges. One key challenge is how to keep the overheads associated with its application low. This is especially important in real-time critical embedded applications, where memory and CPU resources are limited. Another challenge is that of devising expressive and yet user-friendly specification languages that can attract software engineers. In this paper, we show that for many systems, in-place logging provides a satisfactory basis for post-mortem "runtime" verification of logs, where the overhead is already included in system design. While this approach prevents on-line reaction to detected errors, possible with traditional runtime verification, it provides a powerful tool for test automation and debugging — in our case, analysis of spacecraft telemetry by ground operations teams at NASA's Jet Propulsion Laboratory (JPL). The second challenge is addressed in the presented work through a temporal specification language, designed in collaboration with JPL test engineers. The specification language allows for descriptions of relationships between data-rich events (records) common in logs, and is translated into a form of automata supporting data parameterized states. The automaton language is inspired by the rule-based language of the RULER runtime verification system. We present a case study illustrating the use of our LOGSCOPE tool by software test engineers for the 2011 Mars Science Laboratory mission.

[1] School of Computer Science, howard.barringer@manchester.ac.uk
[2] School of Electrical Engineering and Computer Science, alex@eecs.oregonstate.edu
[3] Laboratory for Reliable Software, klaus.havelund@jpl.nasa.gov
[4] Software System Engineering, margaret.h.smith@jpl.nasa.gov

I. 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 over-burdening 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" 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. Consequently, also no additional impact on the running software is imposed beyond the already implemented logging. 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 specification language over events for specifying properties of logs. This language supports data parameterization, essential for monitoring logs (events typically carry data). Specified properties, called *patterns*, are translated into data parameterized automata forming an interesting and useful subset of the textual RULER language [6, 8–10]. Users can mix patterns with more expressive automata in a specification. Parameterized automata are visualized using the GRAPHVIZ tool [24]. Our system additionally offers preliminary support for automatically constructing (learning) specifications from example logs. The log analysis framework, LOGSCOPE, is developed to support engineers testing the flight software for NASA's next Mars rover mission, the Mars Science Laboratory (MSL) [33], developed at the Jet Propulsion Laboratory (JPL). In this sense the work represents an instance of the often sought 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 databases (one database per log). A log is in essence a sequence of time-stamped events, where an event can be one of several forms, corre-

sponding 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 analyzed effectively by humans. Traditionally, these MSL logs have been analyzed by writing scripts, with properties coded up in PYTHON. Such scripts are time consuming to produce and result in difficult-to-read "specifications" that hinder communication, maintenance, specification-sharing, and reuse. Based on our experience with LOGSCOPE, runtime verification of logs using formal specifications has excellent potential as a route to introduce lightweight formal methods into a high-profile NASA flight project.

A. Contributions and Related Work

The contribution of this work is the design of a simple, user friendly, and yet for practical purposes sufficiently expressive data-parameterized temporal specification language, with a translation into data parameterized automata, which are themselves usable for specification. A second contribution is the injection of this technology into a NASA flight mission. The LOGSCOPE system has specifically been influenced by the RULER system [6, 8–10]. In particular, the automaton language conceptually forms a subset of RULER. LOGSCOPE can be seen as an adaptation of RULER to the specific needs of MSL: adding temporal logic, enhancing usability, and implementing it in PYTHON to integrate with the existing scripting culture. LOGSCOPE has also been influenced by the state machine-based systems RCAT [36] and RMOR [26]. The relationship between LOGSCOPE and RULER is illustrated in [6], whereas the process that lead to LOGSCOPE is presented in [25].

Within the last decade, a substantial body of work has been produced on runtime verification [35], e.g. to mention just a few approaches, over and above the work referenced above, consider [2, 5, 15, 17, 20–23, 27, 29, 38]. Each of these is characterized by several dimensions, such as the specification logic they support, the way in which they connect with the application being monitored, the way in which monitors are generated for the propositional (data-free) case, and the way in which they handle data values. LOGSCOPE offers a user friendly logic, avoiding some of the pitfalls of LTL (such as the need for nested until-formulas for expressing sequencing). By analyzing log files, the issue of code instrumentation becomes a separate problem. Data parameterization is handled through parameterized automata: a state machine state can carry a vector of data values. The approach is a restricted form of parameterization mechanisms in the RULER

system, see Section V D, however, the basic idea is a special case of the monitor state being a formula including data. It is also used in our own EAGLE system [5], which offers a very sophisticated logic based on minimal and maximal fixpoints. The sophistication of this logic caused the implementation to be highly complex. The HAWK system [17] augmented EAGLE with an event definition language specialized for Java, and monitors were generated as ASPECTJ [28] aspects. JLO [13, 38] generates monitors in a similar way from parameterized LTL formulas. As explained in [13], the monitor state is a formula including instantiated data values. For example, consider the LTL formula: $\Box(p(x) \Rightarrow \Diamond q(x))$ (*"always: when $p(x)$ is observed, where x is a binding variable name, then eventually $q(x)$ must be observed"*). This formula, upon an observed event $p(1)$, will be rewritten into: $\Box(p(x) \Rightarrow \Diamond q(x)) \wedge \Diamond q(1)$. JLO events are defined by pointcuts inspired by aspect oriented programming, and monitors are generated as ASPECTJ aspects.

Often, generality of parameterization mechanisms is sacrificed for optimality. An example is the TRACEMATCHES system [2], which carries the connection to ASPECTJ to the extreme, and extends ASPECTJ with regular expressions. Data values are handled by annotating state machine states with constraints, and updating these as events are observed. Chen and Roşu [15] implement a range of traditional logics, such as state machines, temporal logic, regular expressions, and grammars, as different plugins in the MOP system. Data (parameters) are handled uniformly in MOP by separating them out from the logic plugins, which are all propositional. Monitors are instead indexed by data values. This makes designing a new logic very easy since only the propositional case needs to be defined. It also makes monitoring very efficient. Events are generated by definition of pointcuts, and monitors are generated as ASPECTJ aspects. As already alluded to, the logics supported by these different systems have drawbacks. For example, state machines are verbose (one has to mention intermediate states), and LTL as well as regular expressions can be difficult to use.

Software testing researchers [3, 14] have also explored formal specifications for checking logs. Chang and Ren [14] provide a specification language just based on regular expressions. The properties are formulated over actual textual logs, in contrast to LOGSCOPE properties, which are formulated over an abstraction of the log: a sequence of records, each of which is a mapping from field names to values. This makes their approach less adaptable to changes in the log format. Chang and Ren provide interesting empirical data on application to a large telecommunications system (though it is not clear whether specifications are written by the testers or the research team). Andrews and Zhang [3] share our focus on lightweight test oracles. They of-

fer an elegant parameterized state machine framework similar to LOGSCOPE parameterized state machines. Their state machines yield errors in case events occur for which there is no out-going transition. Although we support this concept, we also support state machines where one can wait in a state until a certain event occurs. This has turned out to be useful in practice for large log files and occasionally leads to less verbose specifications in the case studies we have looked at. Their state machines are compiled into PROLOG, preventing an online application of monitoring in case this should be desired. They do not provide a temporal logic, a concept that we found important for acceptance by engineers.

The PSL specification language [39] combines temporal logic and regular expressions, and in this way has commonalities with the LOGSCOPE language. However, PSL is designed for hardware verification and does not support data parameterization to the degree that LOGSCOPE does. The LOGSCOPE temporal specification language is characterized by integrating temporal logic and a limited form of regular expressions in a very simple manner (there are for example no explicit temporal operators). The LOGSCOPE specification language is much simpler than PSL. Engineers were able to write properties in less than one hour. Finally, LOGSCOPE offers parameterized state machines, whereas this concept is not directly supported in PSL. The SALT specification language [11] has, as PSL and LOGSCOPE, been developed to make specification writing easier for industrial use. SALT is an impressive collection of temporal operators, which are translated to LTL, or timed LTL in case time constraints are involved. SALT offers operators similar to LOGSCOPE's sequencing operator and scope operator (`upto`, which in SALT is `rejecton` and `accepton`). The goals of PSL and SALT are important. The distinctive feature of LOGSCOPE is its simplicity compared to those languages, although still attempting to improve usability of LTL, and the fact that LOGSCOPE handles data parameters over very large (or infinite) data domains, which is not the case for neither PSL nor SALT.

Mateescu and Thivolle [32] introduce a model checking language (MCL), which augments the modal μ-calculus with high-level operators aimed at improving expressiveness and conciseness of formulas. The main MCL ingredients are data parameterized fixed points, action patterns extracting data values from events, modalities on transition sequences described using extended regular expressions and programming language constructs. The attempt in [32] is similar to the work described in [5] on extending the μ-calculus with data and a sequencing (chop) operator for runtime monitoring. However, although very expressive we do not believe that formulas in MCL are necessarily easy to write for engineers. Our framework is simpler and

directly applicable to formulate the kinds of properties needed for our case studies.

Cohen et al. [16] introduce the PDL specification language for stating properties about programs, to be verified using symbolic execution. The specification language has some similarities with our pattern language in providing a form of sequencing operator and allowing for negation of events. However, PDL is limited in the alternation between positive and negative events allowed. More importantly, events can only be program labels. There is no support for data parameterization.

Dillon et al. introduce in [18] the Graphical Interval Logic GIL for specifying the behaviour of concurrent systems. The logic is quite powerful and addresses the problem of usability of propositional temporal logic. However, the GIL logic is considerably more complex than the logic presented here, making fast learning by test engineers less likely. The LOGSCOPE language is textual as opposed to GIL, which is a graphical notation. It has been our experience that test engineers work faster with a good textual notation rather than a graphical notation. It also makes it easier to auto-generate specifications. Finally, GIL does support some form of data quantification. However, there are no algorithms provided for monitoring such unless the type quantified over is finite and known [19].

The work in [1] describes a set of specification patterns often encountered in work on verification. Each pattern focuses on one specific property format, such as for example the *response* pattern (*P* should always be followed by *Q*). A central part of these specification patterns are *scopes*, which LOGSCOPE to some degree has implemented in terms of the `upto` construct. Our experience is that our general language is simple enough for engineers to work with, and that special patterns are not necessarily needed. This is in fact the strength of the proposed language in our belief. The specification patterns in [1] do not deal with data, and it would likely be a research topic to extend the patterns with such.

B. Outline of Paper

Section II describes the process that lead to the design of the LOGSCOPE specification language and system. Section III describes how log files are constructed for the MSL flight software, forming the foundation of a through-going example. Section IV introduces the temporal pattern language, while Section V introduces the automaton language, and illustrates how patterns are translated to automata. Section VI briefly describes the learning feature. Section VII describes the results of the MSL case study in more detail. Section

VIII finally concludes the paper and suggests future work. Appendix IX contains the complete grammar for the LOGSCOPE specification language.

II. Tool Development Process

The LOGSCOPE tool and pattern language grew out of the needs of engineers. In fact, the earliest effort to "mockup" a specification came from an MSL test engineer. The "specification" was a comment in a PYTHON script describing the purpose of the script as follows.

```
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,*)
```

The script represents the following property to be checked: when an issued `DRILL_DMP` command is observed in the log, then the events with `positive` as an argument should follow in any order, and the events with `negative` as an argument 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; and there should not be an `evr` reporting failure (`CMD_COMPLETED_FAILURE`) of the command. Then follows some requirements on samplings of the state (`channel` events). For example, there should be a sampling of state variable `CMD-0004` that, informally stated, *contains opcode of last immediate command*. Finally the flight software should downlink a `DrillAll` product (a `prod` event) to ground informing about the status of the drill. "*To downlink*" is NASA terminology for sending information from space craft to ground.

The core elements of our temporal 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,

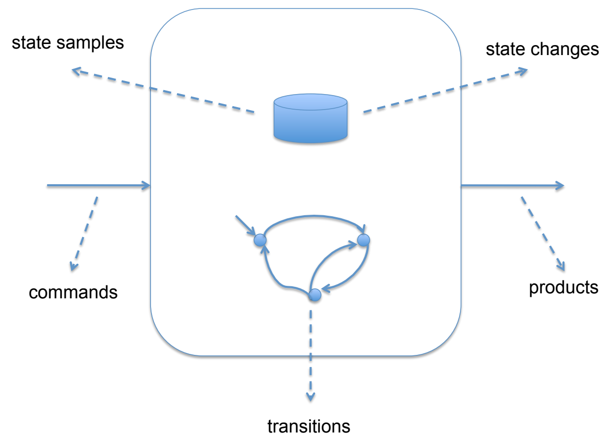

Fig. 1 Observable events of a system

e.g. DRILL_DMP in the sample specification. The spacecraft software responds to commands by causing **transitions**. Transitions in turn cause the local state to be updated. State changes can be observed either in terms of "random" **state samples** that provide a snapshot of current spacecraft state, or more precisely by observing the actual **state changes** when they occur. Finally, data **products** are the "outputs" of the spacecraft downlinked to the ground, including engineering telemetry, images, and science instrument data. These different event kinds are referred to in the logs (see Figure 2) as respectively: COMMAND, EVR (EVent Report = transition), CHANNEL (state sampling), CHANGE (state change) and PRODUCT. Although these MSL concepts are very general (should cover any system) LOGSCOPE can be adjusted to work with other forms of events.

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 its expressive power and readability, guided by usage, engineer requests, and language design considerations. For example, we preserved the structure of properties, each having the form: $E \Rightarrow C$, where E is an event and C is a consequence, for example a sequence of events. However, more complex specifications were required. In the original mockup, the events specified positive in C were allowed to occur in any order in the log. 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, which was a PYTHON library with methods allowing test scripts to create, alter, and delete *online* monitors for telemetry events. In principle, the registry provided a set of tools sufficient for analyzing tests at runtime, and a specification and debugging mechanism well-integrated with established testing procedures. In practice, the registry approach was seldom adopted by engineers (and proved frustrating when used) for two primary reasons:

1. **Ordering and timing of events:** Events on the spacecraft are not downlinked to the ground system in chronological order. 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 event times. Test engineers were forced to either introduce lengthy delays after each test step or 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:** A motivation for the event registry was that it seamlessly integrated with the idea of tests as 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 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 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 requested its features. 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 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 expected responses.

Team management (also active in writing tests) recognized that building ad-hoc test-specific "log anal-

ysis" systems was not an effective use of mission time, and that scripting logic only partially addressed both problems with the registry. The `DRILL_DMP` mockup emerged after a series of team meetings and discussions, informed by our ideas of log analysis. MSL flight software management 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 VII. 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 fields (e.g. bit-vectors), SQL query filtering of events, scoping of specifications (Section IV F), and learning (Section VI), were requested by the test team. A specification language emerged, starting with a syntax and informal semantics inspired by the original mockup, powerful enough to support test needs and simple enough to actually use.

III. Logs

The first step in analyzing the behaviour of the MSL flight system is to produce a *log* — an ordered sequence of events. A log is extracted from a database of 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 used both in MSL hardware testbeds and software simulation. The ground system stores events of each system execution as entries in an SQL database specific for that execution (an approach also considered in runtime verification literature [30]).

The log extracted from such a database is simply an ordered list of named records (effectively a sequence of *dictionaries* in PYTHON). All MSL-specific aspects of the log are handled by the log extractor tool (LOGMAKER), making LOGSCOPE easily adaptable to *any* system producing logs: one just has to replace the log extractor. 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 extracted 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.

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

EVR 7309 {
  message := "Dispatched immediate command
    POWER_HOUSEKEEPING: number=4,
    seconds=789006392, subseconds=1073741824."
  Dispatch := "POWER_HOUSEKEEPING"
  Time := 51708322925696
  name := "CMD_DISPATCH"
  level := "COMMAND"
  Number := "4"}
...
EVR 7311 {
  name := "POWER_SEND_REQUEST"
  Time := 51708322925696
  message := "power_queue_card_request-
    sending request to PAM 0."
  level := "DIAGNOSTIC"}

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

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

CHANNEL 7314 {
  channelId := "PWR-3049"
  DNChange := 67
  dnUnsignedValue := 1600
  type := "UNSIGNED_INT"
```

Figure 2 shows a simplified print of a log by our tool after testing of the power sub-system. Details have been modified in order to not reveal the exact nature of the log. The example includes `COMMAND`, `EVR`, and `CHANNEL` events. The fields of the events are shown within curly 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. A derived field is computed from fields in the data base, yielding additional useful information. Field names 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 time information in the message of the subsequent dispatch EVR event — the `Time` field of `COMMAND 7308` is extracted from the message in `EVR 7309`. 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.

IV. 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*:

$$\langle \textit{specification} \rangle \rightarrow \langle \textit{monitorspec} \rangle^{+}$$

$$\langle \textit{monitorspec} \rangle \rightarrow \langle \textit{pattern} \rangle \mid \langle \textit{automaton} \rangle$$

In this section we focus on temporal patterns, which are automatically translated into parameterized automata.

A. 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:

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

This pattern (**pattern** is a keyword) has the name `P1` and states that *if* a command is observed in the log 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, 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) [34] with quantification over data variables:

$$\forall x, y \bullet$$
$$\Box(\texttt{COMMAND\{Type:"FSW", Stem:x, Number:y\}} \Rightarrow$$
$$\Diamond(\texttt{EVR\{Success:x, Number:y\}}))$$

The temporal operators $\Box$ and $\Diamond$, as well as the universal quantification $\forall$, are implicit in LOGSCOPE's pattern notation. The pattern has the form:

$\langle pattern \rangle \rightarrow$

  pattern $\langle NAME \rangle$ `":"` $\langle event \rangle$ `"=>"` $\langle consequence \rangle$

⟨*consequence*⟩ →

⟨*event*⟩

| `"!"` ⟨*event*⟩

| `"["`⟨*consequencelist*⟩`"]"`

| `"{"`⟨*consequencelist*⟩`"}"`

⟨*consequencelist*⟩ →

⟨*consequence*⟩ (`","`⟨*consequence*⟩)*

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.

An 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 binding 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.

B. 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` *field having the value* `"FSW"`*, the* `Stem` *field having some value x, and the* `Number` *field having some value y, then there should thereafter not*

occur an EVR*, with the field* Failure *mapped to x and the* Number *field mapped to y"*.

We express this by the property (which the log from Section III satisfies):

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

C. Composite Consequences

We have seen that the consequence of a pattern can be an event, as in pattern P1, or the negation of an event, as in 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 then exactly one success of that command. Before the dispatch there should be no dispatch failure, and in between the dispatch and the success there should not be a failure"*.

Requirement R_3 is in Figure 3 illustrated as a timeline, much like in TIMEEDIT [37], showing events that could/should occur, and constraints on events that should not occur. Note, however, that TIMEEDIT does not support data parameterization, only temporal ordering of atomic events. The property can be stated formally as follows, in a form that very closely reflects the time line.

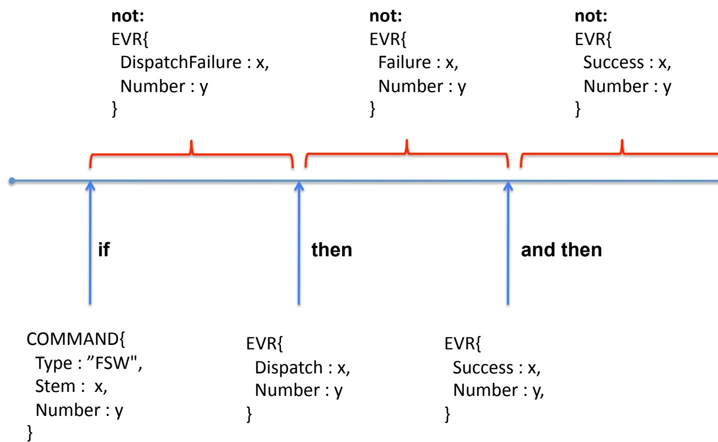

Fig. 3 Requirement R_3 expressed as a timeline

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

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 represent what should *not* happen in between the non-negated events.

The following first order LTL formula represents this property. It is suggestive as to the extra complication that LTL's until operator ('$\mathcal{U}$') introduces, and why the pattern language might be easier to use for

engineers:

$$
\begin{array}{l}
\forall x, y \bullet \\
\quad \Box(\texttt{COMMAND\{Type:"FSW",Stem:x,Number:y\}} \Rightarrow \\
\qquad\quad \neg\texttt{EVR\{DispatchFailure:x,Number:y\}}\ \mathcal{U} \\
\qquad\qquad (\texttt{EVR\{Dispatch:x,Number:y\}} \\
\qquad\qquad \wedge(\neg\texttt{EVR\{Failure:x,Number:y\}}\ \mathcal{U} \\
\qquad\qquad\qquad (\texttt{EVR\{Success:x,Number:y\}} \\
\qquad\qquad\qquad \wedge \bigcirc\Box\neg\texttt{EVR\{Success:x,Number:y\}})))
\end{array}
$$

Linear time temporal logics have by several authors been extended with *chop* operators (‘;’), for example [7]. This eliminates the nesting problem of the until operator, although a re-coding of the given example (and adding quantification over data variables) is still non-trivial:

$$
\begin{array}{l}
\forall x, y \bullet \\
\quad \Box(\texttt{COMMAND\{Type:"FSW",Stem:x,Number:y\}} \Rightarrow \\
\qquad\quad (\texttt{!EVR\{DispatchFailure:x,Number:y\}}\,\mathcal{U}\,(\texttt{EVR\{Dispatch:x,Number:y\}} \wedge \bigcirc \boldsymbol{fin}); \\
\qquad\quad\ \texttt{!EVR\{Failure:x,Number:y\}}\,\mathcal{U}\,(\texttt{EVR\{Success:x,Number:y\}} \wedge \bigcirc \boldsymbol{fin}); \\
\qquad\quad\ \bigcirc\Box\texttt{!EVR\{Success:x,Number:y\}}))
\end{array}
$$

The ***fin*** predicate is true on the empty trace. The chop (in [7]) is overlapping, hence the need to apply the $\bigcirc$ operator to the final $\Box$ term. As an example of an unordered arrangement of events, consider the following (perhaps less meaningful) relaxation of the above stated property:

> *R_4: “Whenever a flight software command is issued, there should follow a dispatch of that command and exactly one success of that command, in no particular required order. There should be no dispatch failure and no failure at any time”.*

We formalize this as:

```
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,Number:y},
     !EVR{Failure:x,Number:y}
    }
```

The curly brackets `{...}` indicate an unordered collection of consequences (a Boolean '$\wedge$' effectively). This corresponds in this case to four parallel timelines, all of which must be satisfied. The fact that they are unordered 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.

D. Event Predicates

Events can be constrained with predicates. Consider 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* $z \leq y$*"*.

The following pattern formalizes this requirement.

```
pattern P5 :
  EVR{Success:_,Number:y} =>
    !EVR{Success:_,Number:z}
        where {: z <= y :}
```

Note that this formula ignores which commands succeed (the underscore '_'). The constraining predicate is: `{: z<=y :}`, and is a general PYTHON expression enclosed by the symbols `{: ... :}`. This expression can refer to all of PYTHON's directly available features, in addition to a LOGSCOPE-specific pre-defined library of predicates. Should this not suffice, it is possible to import PYTHON libraries or directly define PYTHON 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 at most `max` time units apart.

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


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

PYTHON code must occur at the beginning of the specification file and be delimited by the symbols `{: ... :}`. This predicate is then used to check that power commands (having names starting with `"PWR"`) succeed within maximally 10000 time units. The `x.startswith(y)` method is a PYTHON built-in, returning true if the string `y` is a prefix of the string `x`. Note that it is possible to call a predicate directly without embedding it within the more general code brackets `{: ... :}`. Predicates can be composed using the traditional Boolean operators: **or**, **and**, **not**.

As a matter of convenience there are shorthands for certain predicates that occurred frequently during testing of the MSL flight software, such as interval checks and indexing operations, in particular bit-indexing in bit-vectors. The following example illustrates this.

```
pattern P7 :
  COMMAND{Type:"FSW",Stem:"PICT"} =>
    [
      CHANNEL{Status1:{0 : 1, 4 : x}},
      CHANNEL{Status8:{2 : x}},
      PRODUCT{ImageSize:[1000,2000]}
    ]
```

The requirement states that after a flight software picture command, there should be a channel reading where the `Status1` field denotes an integer, where bit number 0 is 1 (counted from the right), and where bit 4 is bound to some value `x` (0 or 1). Thereafter should follow another channel reading where the `Status8` field denotes an integer, where bit 2 has the value `x`. Finally should follow a product, where the field `ImageSize` is in the interval 1000 to 2000. The form of indexing shown for the status fields (`Status1` and `Status8`) generally works on any data object that is indexable, such as bitvectors (integers), lists and strings (in both latter cases indexes must be integers, counting from the left), and dictionaries (maps, allowing for indexing also with strings).

E. Event Actions

As a more experimental feature (not used by the test engineers), LOGSCOPE allows events to be associated with actions to be executed when the events match. As an example, consider the following specification, which is a weakening of property `P1`. It states that only the first command must be followed by a success `EVR`.

```
{:
counter = 0


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


def amongstfirst(limit):
  return counter < limit
:}


pattern P8 :
  COMMAND{Stem:x}
    where amongstfirst(1) do count() =>
      EVR{Success:x}
        do {: print x + " succeeded" :}
```

The specification first introduces a PYTHON code section: a global integer `counter`, a `count()` function, and an `amongstfirst(limit)` function. If a command still matches the **where** predicate (it is amongst the first 1 commands), the **do** construct is executed, causing the `count()` function to be called, counting up the global `counter` variable. Just as predicates can be referred to by name (without enclosing in `{: ... :}`), so can actions as in this example. The general format of an event is:

$\langle event \rangle \rightarrow$

  $\langle type \rangle$ `"{"` $\langle constraints \rangle$ `"}"`

    [**where** $\langle predicate \rangle$]

    [**do** $\langle code \rangle$]

If the user-defined PYTHON code contains the definition of a method `final()` (without aguments), then this method will be called at the end of monitoring the properties. This function can for example compute and print statistics based on the state of the global variables introduced. Each specification file has its own PYTHON state. Hence, it is possible to declare a state local to a subset of properties.

F. 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. Scopes were introduced in the specification patterns described in [1], and also exist for example in SALT [11] (specifically what there is referred to as *exceptions*: `rejecton` and `accepton`). 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 IV C. 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 `COMMAND{Type:"FSW"}`). This is done by adding the scope delimiter **upto** `COMMAND{Type:"FSW"}` as follows:

```
pattern P9 :
  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,Number:y},
     !EVR{Failure:x,Number:y}
    }
    upto COMMAND{Type:"FSW"}
```

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 for patterns is now expanded to include an optional scope-specification:

$\langle pattern \rangle \rightarrow$

pattern $\langle NAME \rangle$ `":"` $\langle event \rangle$ `"=>"` $\langle consequence \rangle$

[**upto** $\langle event \rangle$]

V. The Automaton Language

LogScope also allows testers to write properties as parameterized finite-word $\forall$-automata, a more expressive, but lower-level language than the pattern language. A $\forall$-automaton is a non-deterministic automaton for which accepted sequences have to be accepted in all possible runs. In traditional non-deterministic $\exists$-automata an accepted sequence only has to be accepted in one possible run. Patterns are automatically translated to automata. The automaton language forms a subset of the powerful rule-based Ruler specification language [6, 8–10], with some modifications. The automaton language has been designed to be sufficiently expressive to support the actual encountered needs of the MSL testing engineers, and yet as sim-

ple as possible to minimize learning effort. RULER is a much more comprehensive specification language, resembling a functional stream programming language. The differences between LOGSCOPE and RULER are explained at the end of this sub-section.

In LOGSCOPE an automaton is expressed in terms of states and transitions between states triggered by events. Events are exactly as defined for patterns. Just as events can be parameterized with values, so can states be parameterized, carrying values produced by incoming transitions. Automata, hand-written as well as translated from patterns, can be visualized with GRAPHVIZ [24]. 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 automaton language by presenting the automata for patterns `P3` and `P4`.

A. Automaton for Pattern `P3`

The automaton corresponding to pattern `P3` is shown in Figure 4, and visualized in Figure 5. The automaton consists of four states: `S1`, `S2`, `S3` and `S4`, where the first mentioned state is the initial state (can also be identified via an **initial** 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 be left (disappear) before the end of the log is analyzed. If not it is regarded as an error. They are used to model that some event must occur before the end of the log. It is also possible to declare states hot with a separate **hot**-state declaration, as in: '**hot** `S2, S3`'. In this case, state `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` and a `Number` field with a value equal to the parameter `y`. Similarly for the second transition. There are two special states: the **error** state (indicating

```
automaton A_P3 {

  always S1 {

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

        S2(x,y)

  }


  hot state S2(x,y) {

    EVR{DispatchFailure:x,Number:y} => error

    EVR{Dispatch:x,Number:y} => S3(x,y)

  }


  hot state S3(x,y) {

    EVR{Failure:x,Number:y} => error

    EVR{Success:x,Number:y} => S4(x,y)

  }


  state S4(x,y) {

    EVR{Success:x,Number:y} => error

  }

}
```

Fig. 4 The automaton A_P3 for property P3

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 a no further successes after the first, for a particular command `x` with number `y`.

Automaton visualization uses different symbols for the different states: a grey state represents an initial state, a white circle represents a normal state, a black circle represents an error state, and a downwards-pointed pentagon represents a hot state. The '`@`' symbol in state `S1` indicates an **always**-state.

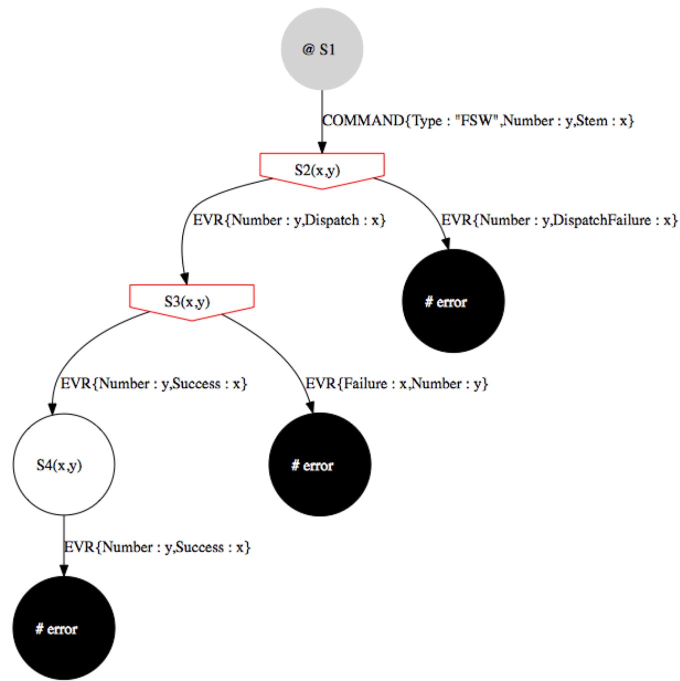

Fig. 5 Visualization of automaton A_P3

B. Automaton for Pattern `P4`

The pattern `P4` is translated into the automaton in Figure 6, and 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 triangle (similar to a '∧') to describe a transition with multiple targets. Note that **done** is visualized as a white circle (a normal state) labelled `#done`.

C. Other Language Concepts

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

```
automaton A_P4 {
  always S1 {
    COMMAND{Type:"FSW",Stem:x,Number:y} =>
        S2(x,y), S3(x,y), S4(x,y), S5(x,y)
  }

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

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

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

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

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

Fig. 6 The automaton A_P4 for property P4

terminated. This is not an error, unless it means that some success state is not reached. This concept is specifically used during learning.

The language supports both C-style multi-line comments (`/*...*/`) and PYTHON-style single-line trail-

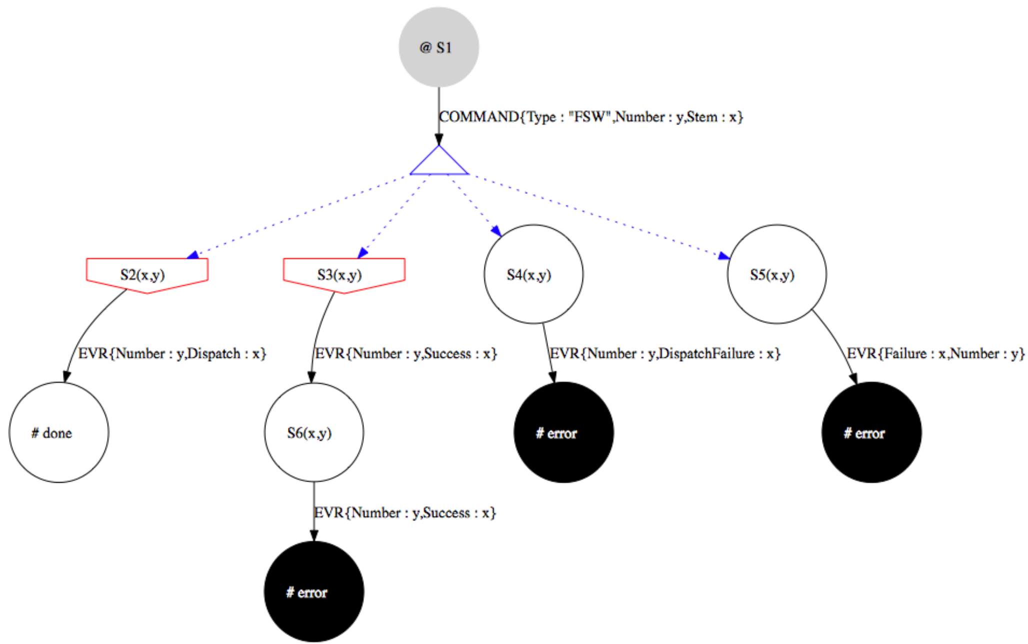

Fig. 7 Visualization of automaton A_P4

ing comments (#...). Furthermore, it is possible to ignore a pattern or automaton by prefixing the declaration with the keyword **ignore**. This is useful for switching specifications on and off, as an alternative to commenting them out.

As mentioned earlier, the automaton language is more expressive than the pattern language. Recall that a pattern has the form: *event* => *consequence*. That is, the antecedent is a single event. It is therefore currently not possible to write a pattern for example of the form: $[e_1, e_2, e_3]$ => e_4, with the intended meaning that if events e_1, e_2 and e_3 occur, then e_4 should occur, despite this property being easily expressed in the automaton language. The pattern language can, of course, be extended to also allow this form of property (work in-progress).

D. Relationship to RULER

As mentioned above, the LOGSCOPE automaton language forms a simplified subset of the RULER language needed for translating the pattern language. As such it represents an interesting simple but powerful language, focusing on data parameterization, for writing monitors. The RULER concepts that have been ignored are the following. Whereas LOGSCOPE supports state machines where a transition in one state is

trigged by one event and leads to a conjunction of resulting states, RULER is indeed a rule-based system where a transition can be triggered by a conjunction of states, events, and negations of such; and where the result is a disjunction of conjunctions of states and negations of such (negating a state means removing it). Disjunction is in RULER used to represent alternative ways of achieving success. Note that what in LOGSCOPE is referred to as a *state*, is in RULER referred to as a *rule activation*. RULER in addition has a notion of super state, similar to that in statecharts/UML. In RULER, templates (what correspond to LOGSCOPE automata) can themselves be parameterized with data as well as with other templates, making RULER higher order. Furthermore, various combinators exist for composing templates (sequential compositon, conditionals, parallel composition, etc.).

E. Type Checking

In addition to the types of events, we must consider the types of event fields. User specifications are statically type-checked to ensure consistencies within a specification; we omit the details of this very standard (and, in this case, fairly trivial) process. Unfortunately, static checking does not detect the most common and serious type errors: inconsistencies between a specification and a log. Because we provide a very flexible specification language, agnostic as to the deep structure of events, we must handle these *dynamic type errors* at run-time, when we can compare with a log. Dynamic type errors can be divided into two classes: field errors and range errors. In field errors, a field is mentioned in the specification but is not present in the log, often due to misspellings or (less commonly) undocumented changes to the log database structure by the MSL ground system or flight software. Range errors generally concern a case where the specification expects one type (i.e., a string, integer, or list) and the log provides a different type object for the field. Another form of range error is when a specification indexes an event field, but the index is out-of-bounds (negative or larger than the size of the object). Our (somewhat ad hoc) solution is to provide users with a list of all patterns and automata that never trigger (which may indicate an error in a field's name, or simply an event that did not occur) and all range violations after checking a specification. In order to improve usability (and in the spirit of PYTHON) we also provide limited automatic conversions (e.g., of strings containing numbers to numbers) while still reporting the range error in question.

VI. Learning

Writing specifications can be difficult and time consuming. The key problem consists of identifying what properties to check — even formulated in English prose. An approach is to generate logs during one or more runs, and then look at (eyeball) the logs while trying to extract *correct-behaviour* patterns. This process can be supported by tools in various ways. One extreme approach is to attempt to learn automatically a specification from one or more presumably "*well behaved*" logs, and then later turn this specification into a monitor to check subsequent logs 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 behaviour, 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 behaviour, 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.

LOGSCOPE can learn exact logs 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 with respect to these fields. The learner module, when given several logs, will build an automaton which represents the set of all logs seen. Common prefixes of the logs 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 automaton learned so far to persistent memory, and then later further refine via learning or use it for monitoring. This form of learning results in typically very large automata. The objective is in this case similar to that of comparing logs 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 learning of logs, what we term *abstract learning*.

The learning facility has not been fully applied, but addresses a need by test engineers to (i) run a set

of nominal test runs in their office, (ii) check by hand that the generated logs conform to expectations, and (iii) at a later stage go to a software/hardware testing laboratory and rerun the tests, this time automatically checking that the results match those of previous runs.

VII. Usage and Case Study

A. 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:

```
import logscope


log = ... # create a log

observer = logscope.Observer("$MSL/spec")

observer.monitor(log)
```

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

Our original log in Figure 2 violates some of the properties presented in this paper. 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    :  0

P8    :  0

P9    :  3 errors

A_P3  :  1 error

A_P4  :  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 `A_P4` presented in Section V B, and visualized in Figure 7. This property is similar to `P4`. `A_P4` is violated 3 times: once because a failure occurs, and twice because neither a dispatch nor a success occurs. The error messages for the dispatch failure and the lack of dispatch are listed below.

```
============================

     RESULTS FOR A_P4:

============================


*** violated: by event 9627 in state:


  state S4(x,y) {

    EVR{DispatchFailure:x,Number:y} => error

  }

  with bindings:

    {'x':'RUN_COMMAND','y':'18'}
```

```
by transition 1 :
EVR{'DispatchFailure':'RUN_COMMAND','Number':'18'} =>
  error

--- error trace: ---

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

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

*** violated: in hot end state:

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

```
  }
  with bindings:
    {'x':'RUN_COMMAND','y':'18'}

--- error trace: ---

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

The first error message explains the 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 `RUN_COMMAND` 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 lack of a dispatch. The *hot* state `S2` is where the monitor rests at the end of the log, indicating a violation. The only event that moved the monitor with respect to this particular violation is the initial `RUN_COMMAND` command, which is the only event listed in the error trace.

B. Application in MSL Testing

As described, our tool and language development was guided generally by the needs of the MSL test engineers. Our first trial run, initiated at the request of MSL software management, checked the core behaviour of the command dispatch and success protocol shown above as pattern `P3`, for 400 issued commands (an automated regression test pre-dating our tool). Using LOGSCOPE enabled us to more easily experiment with the specification compared to what a test engineer could do using PYTHON. This lead to the discovery of

a previously uncaptured error, namely, 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 behaviour. Enabling this automatic generation was a primary motivation for some new language features, including pattern scopes. Again, log analysis revealed several faulty behaviours 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 sends files via an uplink 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 learning capabilities of the tool.

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 improved the test team's productivity, and could result in better-tested flight software.

VIII. 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, primitive and composite (lists, maps). The automaton language forms conceptually a subset of the RULER language, 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 automaton language is more expressive than the pattern language and can be used in cases where this extra expressiveness is required. Data parameterized patterns are automatically translated to data parameterized automata. A point of particular interest is that engineers find it very effective to write specifications in the pattern language and check their precise semantics observing the automaton visualizations. The pattern language itself is 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 probably turned out to 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, no 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, specifying pointcuts in aspect oriented programming [31] 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 [41]. Our recommendation is, therefore, to apply formal runtime verification to logs when automated code instrumentation becomes impractical. Even unstructured logs can be made useful, though structured logging, with events as records, is generally preferable. We experimented with extracting events from text logs, using regular expressions to define events, e.g., in this example where `command` is bound to the number matched by `(\d+)`:

```
event success(command) =
  'COMMAND ((\d+)) \(.*?\) SUCCEEDED'
```

We also hope to establish a connection between requirements engineering and logs, 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 those events. The reactions we received during MSL testing suggest that event-sequences may be a very natural way for systems engineers (not limited to software engineers) working on requirements to formulate properties.

Further future work includes merging a more expressive version of the LOGSCOPE temporal logic with the RULER system [8]. A merge should, however, diminish the distinction between temporal logic and rules (rather than translating one into the other), while still providing the benefits of both. Another line of work is to explore learning techniques. The learning techniques implemented are rather simple (just learning the traces seen, abstracted to certain events and fields). It would be desirable 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 [40]. For example, 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, 12], though the challenge here is considerable: the language of input symbols is very large, and it is not feasible for example to make precise language-inclusion queries as required by Angluin's algorithm. A final goal is to give LOGSCOPE more exposure inside JPL and exploit its potential in other flight missions.

Appendix

IX. LogScope Grammar

A. Lexical Elements

⟨*CODE*⟩ → `{:` ... Python code ... `:}`

⟨*NAME*⟩ → `[a-zA-Z_][a-zA-Z0-9_.]*`

⟨*NUMBER*⟩ → `[0-9]+`

⟨*STRING*⟩ → " ... "

⟨*COMMENT*⟩ → ⟨*COMMENT*$_1$⟩ | ⟨*COMMENT*$_2$⟩

⟨*COMMENT*$_1$⟩ → `/*` ... `*/`

⟨*COMMENT*$_2$⟩ → `#` ... `\n`

B. Grammar

1. Specifications

⟨*specification*⟩ → [⟨*CODE*⟩] ⟨*monitor*⟩$^+$

⟨*monitor*⟩ → [**ignore**] ⟨*monitorspec*⟩

⟨*monitorspec*⟩ → ⟨*pattern*⟩ | ⟨*automaton*⟩

2. Patterns

⟨*pattern*⟩ →

pattern ⟨*NAME*⟩ `":"` ⟨*event*⟩ `"=>"` ⟨*consequence*⟩

[**upto** ⟨*event*⟩]

⟨*consequence*⟩ →

⟨*event*⟩

| `"!"` ⟨*event*⟩

| `"["`⟨*consequencelist*⟩`"]"`

| `"{"`⟨*consequencelist*⟩`"}"`

⟨*consequencelist*⟩ →

⟨*consequence*⟩ (`","`⟨*consequence*⟩)*

3. Automata

⟨*automaton*⟩ →

automaton ⟨*NAME*⟩ `"{"`

⟨*state*⟩*

[**initial** ⟨*actions*⟩]

[**hot** ⟨*names*⟩]

[**success** ⟨*names*⟩]

`"}"`

⟨*state*⟩ →

[⟨*modfier*⟩*] ⟨*statekind*⟩ ⟨*NAME*⟩ [⟨*formals*⟩] `"{"`

⟨*rule*⟩*

`"}"`

⟨*formals*⟩ → `"("`⟨*names*⟩`")"`

⟨*modifier*⟩ → **hot** | **initial**

⟨*statekind*⟩ → **always** | **state** | **step**

⟨*rule*⟩ → ⟨*event*⟩ `"=>"` ⟨*actions*⟩

⟨*actions*⟩ → ⟨*action*⟩ (`","` ⟨*action*⟩)*

⟨*action*⟩ →

⟨*NAME*⟩ [`"("`⟨*arguments*⟩`")"`]

| **done**

| **error**

⟨*arguments*⟩ → [⟨*argument*⟩ (`","` ⟨*argument*⟩)*]

⟨*argument*⟩ → ⟨*NUMBER*⟩ | ⟨*STRING*⟩ | ⟨*NAME*⟩

⟨*names*⟩ → ⟨*NAME*⟩ (`","` ⟨*NAME*⟩)*

4. Events

⟨*event*⟩ →

⟨*type*⟩ `"{"` ⟨*constraints*⟩ `"}"`

[**where** ⟨*predicate*⟩]

[**do** ⟨*code*⟩]

⟨*constraints*⟩ →

[⟨*constraint*⟩ (`","` ⟨*constraint*⟩)*]

⟨*type*⟩ →

`"COMMAND"`

| `"EVR"`

| `"CHANNEL"`

| `"CHANGE"`

| "PRODUCT"

⟨*constraint*⟩ → ⟨*NAME*⟩ ":" ⟨*range*⟩

⟨*range*⟩ →

⟨*NUMBER*⟩

| ⟨*STRING*⟩

| "[" ⟨*NUMBER*⟩ "," ⟨*NUMBER*⟩ "]"

| "{" ⟨*indexes*⟩ "}"

| ⟨*NAME*⟩

| "_"

⟨*indexes*⟩ → ⟨*index*⟩ ("," ⟨*index*⟩)*

⟨*index*⟩ → ⟨*value*⟩ ":" ⟨*range*⟩

⟨*value*⟩ → ⟨*NUMBER*⟩ | ⟨*STRING*⟩

⟨*predicate*⟩ →

⟨*code*⟩

| ⟨*predicate*⟩ **or** ⟨*predicate*⟩

| ⟨*predicate*⟩ **and** ⟨*predicate*⟩

| **not** ⟨*predicate*⟩

| "(" ⟨*predicate*⟩ ")"

⟨*code*⟩ →

⟨*CODE*⟩

| ⟨*NAME*⟩ "(" ⟨*arguments*⟩ ")"

Acknowledgments

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Britain's *Royal*

Academy of Engineering furthermore provided an award to support this work. We would finally like to thank members of the MSL team: Chris Delp, Dave Hecox, Gerard Holzmann, Rajeev Joshi, Cin-Young Lee, Alex Moncada, Cindy Oda, Glenn Reeves, Lisa Tatge, Hui Ying Wen, Jesse Wright, and Hyejung Yun.

References

[1] H. Alavi, G. Avrunin, J. Corbett, L. Dillon, M. Dwyer & C. Pasareanu: *Specification Patterns, SAnToS Laboratory, Kansas State University*. http://patterns.projects.cis.ksu.edu.

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan & J. Tibble (2005): *Adding Trace Matching with Free Variables to AspectJ*. In: *OOPSLA'05*. ACM Press.

[3] J. H. Andrews & Y. Zhang (2003): *General Test Result Checking with Log File Analysis*. *IEEE Transactions on Software Engineering* 29(7), pp. 634–648.

[4] D. Angluin (1987): *Learning Regular Sets from Queries and Counterexamples*. *Inf. Comput.* 75(2), pp. 87–106.

[5] H. Barringer, A. Goldberg, K. Havelund & K. Sen (2004): *Rule-Based Runtime Verification*. In: *Proc. of Fifth International VMCAI conference (VMCAI'04)*, *LNCS* 2937. Springer.

[6] H. Barringer, K. Havelund, D. Rydeheard & A. Groce (2009): *Rule Systems for Runtime Verification: A Short Tutorial*. In: S. Bensalem & D. Peled, editors: *Proc. of the 9th International Workshop on Runtime Verification (RV'09)*, *LNCS* 5779. Springer, pp. 1–24.

[7] H. Barringer, R. Kuiper & A. Pnueli (1984): *Now You May Compose Temporal Logic Specifications*. In: *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing, (STOC'84), Washington, D.C., USA*. pp. 51–63.

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

[9] H. Barringer, D. Rydeheard & K. Havelund (2008): *Rule Systems for Run-Time Monitoring: from Eagle to RuleR*. *Journal of Logic and Computation, doi: 10.1093/logcom/exn076* .

[10] H. Barringer, D.E Rydeheard & K. Havelund (2008): *RuleR: A Tutorial Guide*. Available at: http://www.cs.man.ac.uk/~howard/LPA.html.

[11] A. Bauer, M. Leucker & J. Streit (2006): *SALT - Structured Assertion Language for Temporal Logic*. In: *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, *LNCS* 4260. Springer, pp. 757–775.

[12] A. W. Biermann & J. A. Feldman (1972): *On the Synthesis of Finite-State Machines from Samples of their Behaviour*. *IEEE Transactions on Computers* 21, pp. 592–597.

[13] E. Bodden (2005): *J-LO - A Tool for Runtime-Checking Temporal Assertions*. Master's thesis, RWTH Aachen University.

[14] F. Chang & J. Ren (2007): *Validating System Properties Exhibited in Execution Traces*. In: *Automated Software Engineering*. pp. 517–520.

[15] F. Chen & G. Roşu (2007): *MOP: An Efficient and Generic Runtime Verification Framework*. In: *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*.

[16] A. Coen-Porisini, G. Denaro, C. Ghezzi & M. Pezzè (2001): *Using Symbolic Execution for Verifying Safety-Critical Systems*. In: *ESEC/FSE-9: Proc. 8th European Software Engineering Conference*. ACM Press, pp. 142–151.

[17] M. D'Amorim & K. Havelund (2005): *Runtime Verification for Java*. In: *Workshop on Dynamic Program Analysis (WODA'05)*.

[18] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith & Y. S. Ramakrishna (1994): *A Graphical Interval Logic for Specifying Concurrent Systems*. *ACM Transactions on Software Engineering and Methodology* 3(2), pp. 131–165.

[19] L. K. Dillon & Y. S. Ramakrishna (1996): *Generating Oracles from Your Favorite Temporal Logic Specifications*. *ACM SIGSOFT Software Engineering Notes archive* 21(6), pp. 106–117.

[20] D. Drusinsky (2000): *The Temporal Rover and the ATG Rover*. In: *SPIN Model Checking and Software Verification*, *LNCS* 1885. Springer, pp. 323–330.

[21] D. Drusinsky (2006): *Modeling and Verification using UML Statecharts*. Elsevier. ISBN-13: 978-0-7506-7949-7, 400 pages.

[22] S. Eckmann, G. Vigna & R. A. Kemmerer (2001): *STATL Definition*. Reliable Software Group, Department of Computer Science, University of California, Santa Barbara, CA 93106.

[23] Y. Falcone, J.-C. Fernandez & L. Mounier (2009): *Runtime Verification of Safety-Progress Properties*. In: S. Bensalem & D. Peled, editors: *Proc. of the 9th International Workshop on Runtime Verification (RV'09)*, *LNCS* 5779. Springer, pp. 40–59.

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

[25] A. Groce, K. Havelund & M. Smith (2010): *From Scripts to Specifications - The Evolution of a Flight Software Testing Effort*. In: *32nd International Conference on Software Engineering (ICSE'10), Cape Town, South Africa, May 2-8, 2010, Proceedings, ACM SIG*.

[26] K. Havelund (2008): *Runtime Verification of C Programs*. In: *Proc. of the 1st TestCom/FATES conference*, *LNCS* 5047. Springer, Tokyo, Japan.

[27] K. Havelund & G. Roşu (2004): *Efficient Monitoring of Safety Properties*. *Software Tools for Technology Transfer* 6(2), pp. 158–173.

[28] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm & W. G. Griswold (2001): *An Overview of AspectJ*. In: J. Lindskov Knudsen, editor: *European Conference on Object-oriented Programming*, *LNCS* 2072. Springer, pp. 327Ð–353.

[29] M. Kim, S. Kannan, I. Lee & O. Sokolsky (2001): *Java-MaC: a Run-time Assurance Tool for Java*. In: *Proc. of the 1st International Workshop on Runtime Verification (RV'01)*, *Electronic Notes in Theoretical Computer Science* 55(2). Elsevier.

[30] D. Kortenkamp, T. Milam, R. Simmons & J. L. Fern (2001): *Collecting and Analyzing Data from Distributed Control Programs*. In: *Proc. of the 1st International Workshop on Runtime Verification (RV'01)*, *Electronic Notes in Theoretical Computer Science* 55(2). Elsevier, pp. 133–151.

[31] R. Laddad (2003): *AspectJ in Action*. Manning.

[32] R. Mateescu & D. Thivolle (2008): *A Model Checking Language for Concurrent Value-Passing Systems*. In: *The 15th international symposium on Formal Methods (FM 2008)*, *LNCS* 5014. Springer. Turku, Finland.

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

[34] A. Pnueli (1977): *The Temporal Logic of Programs*. In: *Proc. of the 18th IEEE Symposium on Foundations of Computer Science*. pp. 46–77.

[35] `http://www.runtime-verification.org`.

[36] M. Smith & K. Havelund (2008): *Requirements Capture with RCAT*. In: *16th IEEE International Requirements Engineering Conference (RE'08)*, IEEE Computer Society. Barcelona, Spain.

[37] M. Smith, G. Holzmann & K. Ettessami (2001): *Events and Constraints: a Graphical Editor for Capturing Logic Properties of Programs*. In: *5th Int Sym. on Requirements Engineering*, 55(2). Toronto, Canada, pp. 14–22.

[38] V. Stolz & E. Bodden (2006): *Temporal Assertions using AspectJ*. *Electronic Notes in Theoretical Computer Science* 144(4). Elsevier, pp. 109–124.

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

[40] J. Yang, D. Evans, D. Bhardwaj, T. Bhat & M. Das (2006): *Perracotta: Mining Temporal API Rules from Imperfect Traces*. In: *International Conference on Software Engineering*. pp. 282–291.

[41] A. Zeller (2005): *Why Programs Fail: a Guide to Systematic Debugging*. Morgan Kaufmann.