Experience with Rule-Based Analysis of Spacecraft Logs[⋆]

Klaus Havelund and Rajeev Joshi

Jet Propulsion Laboratory
California Institute of Technology
California, USA

Abstract. One of the main challenges facing the software development as well as the hardware communities is that of demonstrating the correctness of built artifacts with respect to separately stated requirements. Runtime verification is a partial solution to this problem, consisting of checking actual execution traces against formalized requirements. A related activity is that of humans attempting to understand (or *comprehend*) what the system does when it executes, for validation purposes, or for simply operating the system optimally. For example, a key challenge in operating remote spacecraft is that ground operators must rely on the limited visibility available through spacecraft telemetry in order to assess spacecraft health and operational status. In this paper we illustrate the use of the rule-based runtime verification system LogFire for supporting such log comprehension. Specifically, LogFire is used for generating abstract events from the concrete events in logs, followed by a visualization of these abstract events using the D3 visualization framework.

1 Introduction

1.1 Motivation

Demonstrating the correctness of a software or hardware artifact is a challenging problem. In the ideal case we want to prove the artifact correct for all possible input. Unfortunately, full verification is still cost-prohibitive for complex systems (especially those with tight deadlines), so practitioners typically use less formal, but cheaper, alternatives to build confidence in their systems. One such alternative is runtime verification, which checks a particular execution against a formal specification, which in this case becomes the test oracle. However, runtime verification systems can be used during deployment as well, to monitor the actual execution of the system in the field. Such monitoring can happen *online*, as the system executes, or *offline* by analyzing log files generated by the running system. Violations of the formal specification can be flagged by the runtime verification system, either leading to automated behavior modification in the case

[⋆] The work described in this publication was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

of online monitoring, or human driven systems modification in the case of offline monitoring (as examples).

In this paper we suggest yet a slightly different use of a runtime verification system: namely that of *system comprehension*. One of the key challenges in operating remote spacecraft is that the only knowledge ground operators have of the spacecraft behavior is the telemetry sent down to earth. Such telemetry typically consists of logs of system events and sensor measurements (such as battery voltage or probe temperature). A log may be viewed as a sequence of time-stamped records with named fields. Current practice at NASA's Jet Propulsion Laboratory (JPL) is to develop ad-hoc tools using various scripting languages, resulting in a growing collection of scripts that are hard to maintain and modify, which becomes a concern for long-running missions that last many years. A more desirable solution is a specification-based approach where *comprehension rules* are formulated in a human readable DSL (Domain Specific Language). In this paper, we present such an approach applied to the telemetry received from the Curiosity rover currently on Mars, and part of the MSL (Mars Science Laboratory) mission [31].

1.2 Contribution

More concretely, we illustrate the application of the LogFire runtime verification system [25] and the D3 visualization system [14] to support *human comprehension* of logs sent down to earth from Curiosity. Although illustrated on such logs, the approach is fully general. LogFire is a rule-based monitoring framework, a concept extensively studied within the artificial intelligence community. It is implemented in the Scala programming language as an internal DSL (essentially an API), and its core algorithm is a modification of the Rete algorithm [19], to support event processing as well as fast indexing, as described in [25]. Rete is one of the original algorithms for rule-based systems, optimizing rule-evaluation, and known for its complexity.

Rules have the form: $condition_1, \ldots, condition_n \Rightarrow action$. The state of a rule-system can abstractly be considered as consisting of a set of *facts*, referred to as the *fact memory*, where a fact is a mapping from field names to values. A condition in a rule's left-hand side can check for the presence or absence of a particular fact. A left-hand side matching against the fact memory usually requires unification of variables occurring in conditions. In the case that all conditions on a rule's left-hand side match (become true), the right-hand side action is executed, which can be any Scala code, including adding and deleting facts, or generating error messages. The DSL allows domain specific constructs to be mixed with Scala code, making the notation very expressive and convenient for practical purposes, one of the reasons that LogFire is used daily on the MSL mission.

LogFire was originally developed for verifying execution traces against formal specifications. A main focus was monitoring of events carrying data, what is also sometimes referred to as data parameterized monitoring. The purpose was to understand how well rule-based systems fare for this form of task. In the work presented here, we instead use LogFire for generating abstract facts from low level events occurring in a log. Such facts are generated as a result of executing actions of rules triggered by lower-level events and facts. The rule-based approach is particularly suited for this form of fact generation, compared to other forms of runtime verification logics, such as temporal logics, regular expressions or state machines. The collection of facts built in this manner is then fed into a visualization tool implemented using the D3 library. The system is currently in use by the Curiosity operations team. A main core message of this work is that runtime verification as a field should embrace a wide range of technologies for not only verifying systems but also for learning and comprehending their behavior.

1.3 Related work

In [26] we describe an attempt to build a DSL on top of LogFire in order to make it even easier to formulate abstraction rules for log comprehension and visualization. That work, however, is still a research prototype, and not (yet) used in mission operations, as is the case with the work presented here.

Numerous systems have been developed over the last decade for supporting monitoring of parameterized events, using various formalisms, such as state machines [21, 32, 11, 5, 2], regular expressions [1, 32], variations over the μ-calculus [4], temporal logics [4, 32, 5, 22, 8, 9, 15], grammars [32], and rule-based systems [7, 25]. LogFire itself was in part inspired by the Ruler system [7]. Other rule-based systems include Drools [17], Jess [27] and Clips [13]. Standard rule systems usually enable processing of facts, which have a life span. In contrast, LogFire additionally implements events, which are instantaneous. Drools supports a notion of events, which are facts with a limited life span, inspired by the concept of *Complex Event Processing* (CEP), described by David Luckham in [28].

Two other rule-based internal DSLs for Scala exist: Hammurabi [20] and Rooscaloo [33]. Hammurabi, which is not Rete-based, achieves efficient evaluation of rules by evaluating these in parallel, assigning each rule to a different Scala actor. Rooscaloo [33] is Rete based, but is not documented in any form other than experimental code. The Drools project has an effort ongoing, defining functional programming extensions to Drools [18]. In contrast, by embedding a rule system in an object-oriented and functional language, as done in LogFire, we can leverage the already existing host language features.

TraceContract [5] and Daut (Data automata) [23, 24] are internal Scala DSLs for trace analysis based on state machines. They allow for multi-transitions without explicitly naming the intermediate states, which allows for temporal logic like specifications, in addition to data parameterized state machines. TraceContract was deployed throughout the LADEE mission [29], checking command sequences (similar format as logs) sent to the spacecraft, as documented in [6] at an early stage of that project.

1.4 Contents

The paper is organized as follows. Section 2 introduces the rule-based system LogFire, illustrating how it can be used for verifying the correctness of program/system executions. The example is that of general deadlock potential detection between any number of tasks, chosen since it illustrates the expressive power of rules. Section 3 presents the application of LogFire to abstract and visualize telemetry from the Curiosity rover. Finally, Section 4 concludes the paper.

2 The LogFire runtime verification system

LogFire [25] is an API in the Scala programming language, also here referred to as an internal DSL, created for writing rule-based runtime monitors. A monitor is specified as a set of rules, each of the form: $lhs \Rightarrow rhs$, which operate on a database of facts, called the *fact memory*. Rule left-hand sides test on incoming events, as well as presence or absence of facts in the fact memory. Right-hand sides (actions) can add facts to the fact memory, delete facts, issue error messages, and generally execute any Scala code. A monitor takes as input a sequence of events, consumed one at a time, and for each event executes the actions of those rules whose left-hand sides evaluate to true. Monitors can be used to analyze the execution of a program as it executes or to analyze logs produced by the program. LogFire is an implementation of the Rete algorithm [19], specifically as it is described in [16], modified to process instantaneous events (in addition to facts that have a life span), and to perform faster lookups in the fact memory. We will illustrate LogFire using the example of detecting deadlock potentials [10] in a program by just analyzing a single execution trace generated by an instrumented version of the program. This example illustrates the flexibility of using a rule-based system. The reader is referred to [25] for more details about the implementation of LogFire.

2.1 The deadlock potential detection problem

Deadlock potentials in a program can very easily be detected by analyzing single execution traces generated by an appropriately instrumented program. We consider traces that only contain two kinds of events: $lock(t, l)$, representing that task t takes the lock l; and $unlock(t, l)$, representing that task t releases lock l. As described in [10], the standard technique for detecting deadlock potentials is to build a *lock graph*, where nodes are locks and where there is an edge between two nodes (locks) l_1 and l_2, labelled with task id t, if task t at some point holds lock l_1 while taking lock l_2. Nodes and edges are only added to the graph (never deleted). If at some point the graph contains a cycle it indicates the potential for deadlock, although not necessarily an actual deadlock. The algorithm can typically be made efficient in practice since it only needs to check for deadlock *potentials*, in contrast to, say, a model checker, which typically has to search the reachable state space for *actual* deadlocks.

Figure 1 shows an example illustrating how the algorithm works. The left-hand side shows three tasks (say, threads in a multi-threaded program), each taking two locks, then entering a critical section, and then releasing the locks. The locks are taken in a circular manner: a deadlock can occur if the tasks are scheduled such that each gets to take their first lock, but not the second. After that point none of the tasks can take the second lock since it is already held by one of the other tasks. Testing may not reveal this deadlock which only happens with certain schedules. For example, if we run these tasks in a sequential manner (first task T_1, then T_2, and then T_3), no deadlocks will occur. However, we can record the lockings and unlockings in a graph, as shown on the right of Figure 1. Each node is a lock, and an edge is drawn from a lock l_x to a lock l_y, labelled with task T_z if task T_z at some point holds lock l_x while taking lock l_y. If this graph ends up containing a cycle, as in this case, we have detected the potential for a deadlock.

$T_1 : \mathit{lock}(l_1);\ \mathit{lock}(l_2);$
$\quad\langle \mathit{critical\ section}\ 1\rangle$
$\quad\mathit{unlock}(l_2);\ \mathit{unlock}(l_1)$

$T_2 : \mathit{lock}(l_2);\ \mathit{lock}(l_3);$
$\quad\langle \mathit{critical\ section}\ 2\rangle$
$\quad\mathit{unlock}(l_3);\ \mathit{unlock}(l_2)$

$T_3 : \mathit{lock}(l_3);\ \mathit{lock}(l_1);$
$\quad\langle \mathit{critical\ section}\ 3\rangle$
$\quad\mathit{unlock}(l_1);\ \mathit{unlock}(l_3)$

$$\begin{tikzcd}
l_1 \arrow[rr, bend left, "T_1"] & & l_2 \arrow[dl, bend left, "T_2"] \\
& l_3 \arrow[ul, bend left, "T_3"] &
\end{tikzcd}$$

Fig. 1. Example illustrating three tasks T_1, T_2, and T_3, taking three locks l_1, l_2, and l_3 in a cyclic manner, opening for a deadlock potential. This is detected as a cycle in the corresponding lock graph

Traditional implementations of such deadlock-potential checkers are coded as algorithms in a programming language [10]. An alternative is to formulate such a checker in a logic as a monitor specification, expressing that there must be no such cycles. The general case involves a cycle between any number n of tasks. It turns out, however, that traditional temporal logic is not expressive enough for the case where n is unknown (it can vary at execution time). Temporal logic solutions for exactly two tasks are shown in [3, 34]. For example, the solution provided in [34] has the following form (with some minor changes for presentation purposes) expressed in linear temporal logic (LTL) extended with data, and stating the property that no cycles should exist between two tasks and locks:

$$
\begin{array}{l}
\forall t_1, t_2 : \mathit{Task}, l_1, l_2 : \mathit{Lock} \bullet \\
\quad \mathbf{G}\ (\\
\qquad \neg \mathit{lock}(t_1, l_2)\ \mathbf{U}\ (\mathit{lock}(t_1, l_1) \wedge (\neg \mathit{unlock}(t_1, l_1)\ \mathbf{U}\ \mathit{lock}(t_1, l_2))) \\
\qquad \rightarrow \\
\qquad \mathbf{G}\ \neg(\neg \mathit{lock}(t_2, l_1)\ \mathbf{U}\ (\mathit{lock}(t_2, l_2) \wedge (\neg \mathit{unlock}(t_2, l_2)\ \mathbf{U}\ \mathit{lock}(t_2, l_1)))) \\
\quad)
\end{array}
$$

This formula can be read as follows: always (**G**), **if** task t_1 does not take lock l_2 until (**U**) it takes lock l_1, and from then on does not release l_1 until (**U**) it takes l_2, **then** always (**G**), it is not the case that task t_2 follows the opposite pattern. Besides being cumbersome to read, it only captures the situation for two tasks and two locks. As we show in the next section, using a rule-based logic makes it possible to express the property for an arbitrary number of tasks.

2.2 Formulating deadlock detection in LogFire

Assume that our traces contain the two events: lock(t,l) and unlock(t,l). The cycle detection property (that no cycles should exist) is shown in Figure 2. The main component of LogFire is the **trait**[1] *Monitor*, which any user-defined monitor must extend to get access to the constants and methods provided by the rule DSL. The *events* lock and unlock are short-lived instantaneous observations about the system being monitored, those submitted to the monitor. In contrast, *facts*, in this case Locked and Edge, are long-lived pieces of information stored in the fact memory of the rule system, generated and deleted explicitly by the rules. The monitor contains five rules. Each rule has the form:

$$\mathit{name}\ \texttt{--}\ \mathit{condition}_1\ \&\ldots\&\ \mathit{condition}_n\ \longmapsto\ \mathit{action}$$

Event and fact names, as well as parameter names are values of the Scala type *Symbol*, which contains quoted identifiers such as 't. The rules read as follows. The first rule, named `lock`, states that on observation of a lock('t, 'l) event we insert a Locked('t, 'l) fact in the fact memory, representing the fact that task t holds the lock l. The second rule, named `unlock`, states that if a task t holds a lock l (represented by Locked('t, 'l)), and an unlock('t, 'l) event is observed, then that Locked fact is removed from the fact memory. The third rule, named `edge`, states that if a task t holds a lock l1 (represented by Locked('t, 'l1), and a lock('t, 'l2) event is observed, then an edge from l1 to l2 is drawn. The fourth rule, named `close`, performs the transitive closure of the edge-relation. Note that LogFire for each event first evaluates all left-hand sides, recording which evaluate to true. Then it deletes the event from the fact memory, evaluates all the corresponding right-hand sides, and continues evaluating rules until a fixed point is reached (infinite loops are possible to program by a mistake). Only

[1] A **trait** in Scala is a module concept closely related to the notion of an *abstract class*, as for example found in Java.

```
class NoLockCycles extends Monitor {
  val lock, unlock = event
  val Locked, Edge = fact

  "lock" −− lock('t,'l) ⟼ insert(Locked('t,'l))

  "unlock" −− Locked('t,'l) & unlock('t,'l) ⟼ remove(Locked)

  "edge" −− Locked('t,'l1) & lock('t,'l2) ⟼ insert(Edge('l1,'l2))

  "close" −− Edge('l1,'l2) & Edge('l2,'l3) & not(Edge('l1,'l3)) ⟼
    insert (Edge('l1,'l3))

  "cycle" −− Edge('l1,'l2) ⟼ {
    if (get('l1) == get('l2)) fail ("cycle detected on " + get('l1))
  }
}
```

Fig. 2. No-lock-cycles property in LogFire

hereafter is the next event is consumed. This special handling of events is one difference wrt. the original Rete algorithm described in [19, 16]. The last rule, named `cycle`, detects cycles in the graph. It states that if there is an edge from a lock to itself then it is considered a deadlock potential. Symbols representing bindings of parameter values must be accessed with special get functions.

A monitor can be applied as shown in Figure 3. Since the trace exposes a deadlock potential, an error trace is produced as shown in Figure 4. Each entry in the error trace shows the number of the event, the event, the fact that it causes to be generated, and the rule that triggers.

2.3 Improving the specification

The deadlock potential detection specification shown in Figure 2 can be improved in three ways. Firstly, it can yield false positives. It will for example report a deadlock potential for a single task that accesses locks in a cyclic manner, although a single task cannot deadlock on its own (assuming reentrant locks). In order to exclude such false positives (although it can be argued that any cycles should be avoided), edges in the lock graph should be labelled with task ids, and a cycle is only reported in case all the task ids on the edges of the cycle are different. Secondly, the monitor will report the same deadlock potential multiple times due to the fact that different cycles (starting in different locks) represent the same problem. Thirdly, lock(t,l) and unlock(t,l) events are assumed to have exactly two arguments. Events in general may have many arguments,

```
object ApplyMonitor {
  def main(args: Array[String]) {
    val m = new NoLockCycles

    m.addEvent('lock)(1, "l1")
    m.addEvent('lock)(1, "l2")
    m.addEvent('unlock)(1, "l2")
    m.addEvent('unlock)(1, "l1")

    m.addEvent('lock)(2, "l2")
    m.addEvent('lock)(2, "l3")
    m.addEvent('unlock)(2, "l3")
    m.addEvent('unlock)(2, "l2")

    m.addEvent('lock)(3, "l3")
    m.addEvent('lock)(3, "l1")
    m.addEvent('unlock)(3, "l1")
    m.addEvent('unlock)(3, "l3")
  }
}
```

Fig. 3. Applying the lock pattern monitor to a trace corresponding to executing the three tasks in Figure 1 in sequential order

and instead of referring to them in a positional style as shown, we may want to pick out those arguments we are interested in by name, as for example with the notation unlock('task $\rightarrow$ 't, 'lock $\rightarrow$ 'l). The alternative specification shown in Figure 5 is an attempt to make these improvements and to illustrate additional features of LogFire.

As can be seen, event arguments are referred to by name, as in unlock('task $\rightarrow$ 't,'lock $\rightarrow$ 'l). Each edge now also includes a set of task ids, namely those involved in forming the edge. A check in rule `close` is now performed that two edges can only be composed (transitive closure) if their task ids differ. We see here the use of set operations and sets as arguments to facts. Finally, in order to avoid a deadlock between a set of tasks to be reported multiple times, a variable is declared in the monitor, storing the sets of tasks that have so far been reported being involved in a deadlock potential. The rule `cycle` avoids to report a deadlock potential between a set of tasks in case a such has already been reported for those tasks. This illustrates how rules can be mixed with Scala code, including declaration of variables and methods.

```
[1]  'lock(1,"l1") ⟹ 'Locked(1,"l1")
     rule: "lock" −− 'lock('t,'l) ⟼ {...}

[2]  'lock(1,"l2") ⟹ 'Edge("l1","l2")
     rule: "edge" −− 'Locked('t,'l1) & 'lock('t,'l2) ⟼ {...}

[5]  'lock(2,"l2") ⟹ 'Locked(2,"l2")
     rule: "lock" −− 'lock('t,'l) ⟼ {...}

[6]  'lock(2,"l3") ⟹ 'Edge("l1","l3")
     rule: "close" −− 'Edge('l1,'l2) & 'Edge('l2,'l3) & not('Edge('l1,'l3)) ⟼
       { ... }

[6]  'lock(2,"l3") ⟹ 'Edge("l2","l3")
     rule: "edge" −− 'Locked('t,'l1) & 'lock('t,'l2) ⟼ {...}

[9]  'lock(3,"l3") ⟹ 'Locked(3,"l3")
     rule: "lock" −− 'lock('t,'l) ⟼ {...}

[10]  'lock(3,"l1") ⟹ 'Edge("l3","l1")
      rule: "edge" −− 'Locked('t,'l1) & 'lock('t,'l2) ⟼ {...}

[10]  'lock(3,"l1") ⟹ 'Edge("l1","l1")
      rule: "close" −− 'Edge('l1,'l2) & 'Edge('l2,'l3) & not('Edge('l1,'l3)) ⟼
        { ... }

[10]  'lock(3,"l1") ⟹ 'Fail("ERROR cycle detected on l1")
      rule: "cycle" −− 'Edge('l1,'l2) ⟼ {...}
```

Fig. 4. An error trace representing a lock cycle

3 Analyzing telemetry from the Curiosity rover

In this section, we describe how we have used LogFire to process telemetry received from the Curiosity rover on Mars. Our focus here is on building tools based on LogFire for processing telemetry in order to generate summaries that can be used for creating effective visualizations for use by the daily operations team. Our tools are integrated into the mission ground data system, and receive and automatically process telemetry from the rover several times a day. As this telemetry is processed, the tools generate summary files, typically in comma separated values (CSV) format. These summary files are in turn used by visualizations built using the D3 library [12]; these visualizations are used as part of a "dashboard" that is regularly monitored by mission operators and

```
class NoLockCyclesImproved extends Monitor {
  val lock, unlock = event
  val Locked, Edge = fact

  def getset(s: Symbol) = get[Set[Int]](s)

  var cycles: Set[Set[Int]] = Set()

  "lock" -- lock('task → 't,'lock → 'l) ⟼ insert(Locked('t,'l))

  "unlock" -- Locked('t,'l) & unlock('task → 't,'lock → 'l) ⟼
    remove(Locked)

  "edge" -- Locked('t,'l1) & lock('task → 't,'lock → 'l2) ⟼
    insert(Edge(Set(get[Int]('t)),'l1,'l2))

  "close" -- Edge('s1,'l1,'l2) & Edge('s2,'l2,'l3) & not(Edge('_,'l1,'l3)) ⟼
  {
    if (getset('s1).intersect(getset('s2)).isEmpty)
      insert(Edge(getset('s1).union(getset('s2)),'l1,'l3))
  }

  "cycle" -- Edge('s,'l1,'l2) ⟼
  {
    if (get('l1) == get('l2) & !cycles.contains(getset('s)))
      fail("cycle detected between tasks" + get('s))
    cycles += getset('s)
  }
}
```

Fig. 5. Improved no-lock-cycles property in LogFire

science planners. In the following subsections, we give two examples of telemetry processing tools and show how they are used in building useful visualizations.[2]

3.1 Monitoring sequence execution status

The first example shows a tool that monitors execution of spacecraft *sequences*. A sequence is a list of commands that perform specific spacecraft actions such as taking an image, or deleting a file, or possibly even invoking another sequence.

[2] In the interests of readability, and to comply with NASA restrictions on publishing mission data, we have simplified the examples and modified the actual names and times from actual telemetry.

```
class SeqMonitor extends EvrMonitor {
  val SeqStart, SeqDone = fact
  def seq_name(s:String) = words(s)(2) // Helper function

  "start_seq" -- EVR('id → "EVR_SEQ_START", 'sclk → 'S, 'msg → 'M) ⟼ {
    val w = words('M.s)
    val seq_name = w(15).slice(1, w(15).length−2)
    insert (SeqStart(seq_name, 'S.d))
  }

  "end_seq_ok" -- EVR('id → "EVR_SEQ_SUCCESS", 'sclk → 'E, 'msg → 'M)
                & 'SeqStart('name, 'S) ⟼ {
      if (seq_name('M.s) == 'sname.s) {
        replace (SeqStart)(SeqDone('name.s, 'S.d, 'E.d, "OK"))
      }
    }

  "end_seq_fail" -- EVR('id → "EVR_SEQ_FAILURE", 'sclk → 'F, 'msg → 'M)
                  & 'SeqStart('name, 'S) ⟼ {
      if (seq_name('M.s) == 'name.s) {
        replace (SeqStart)(SeqDone('name.s, 'S.d, 'F.d, "FAIL"))
      }
    }

  "print" -- SeqDone('name, 'S, 'E, 'stat) ⟼ {
    updateCSV('name.s, 'S.d, 'E.d, 'stat.s)
    remove(SeqDone)
   }
}
```

Fig. 6. Rules for sequence execution

The operations team typically uplinks a list of sequences every other day containing the commands that the rover should perform over the next two days. This includes mobility requests (such as driving to a specific location), science requests (such as taking a panorama or firing a laser), as well as engineering requests (such as deleting old data files to free up space on flash memory).

Figure 6 shows the rules for processing telemetry related to sequence execution. These rules rely on processing an *event log* which is generated on board and sent to the ground periodically. The event log consists of a list of EVRs (short for "event reports"); each EVR has an associated timestamp (indicating the sclk, or *spacecraft clock* time when the event occurred), a unique identifier, and a text *message* describing the event. The SeqMonitor class extends the trait EvrMonitor, which itself extends trait Monitor, and in addition defines various utilities,

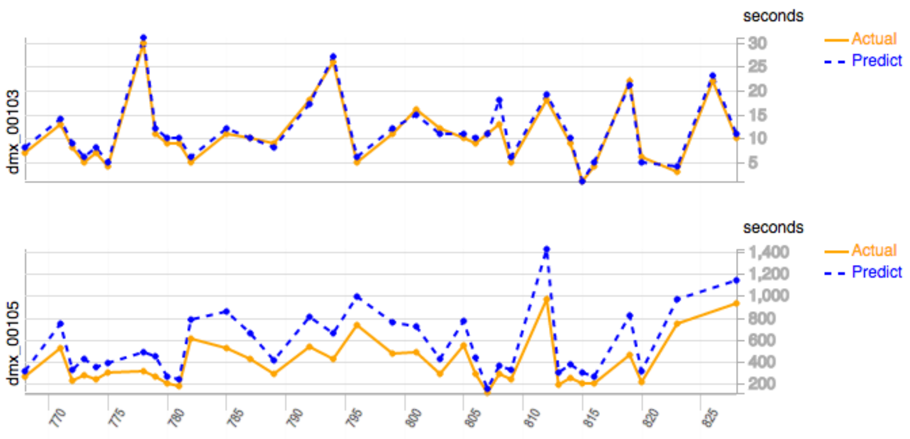

Fig. 7. Visualization showing actual vs predicted sequence run times

such as the EVR event. The rule `start_seq` is triggered by the log event `EVR_-SEQ_START` and adds a fact SeqStart to the fact memory, recording the name and start time of the sequence. A sequence may terminate either successfully or unsuccessfully. A successful termination is denoted by the event `EVR_SEQ_-SUCCESS`, which results in the SeqStart fact being replaced by a fact SeqDone, which records the name, start and end times of the sequence, along with the status `OK`, indicating that the sequence completed successfully. A sequence that terminates with failure results in the SeqStart fact being replaced by a fact SeqDone, which records the name, start and end times as before, along with the status `FAIL`. Finally, the `print` rule updates a CSV file containing a row for each sequence invocation, recording the start and end times and execution status.

This CSV file is useful for building various visualizations that track how ground commands are being performed by the rover. As an example, Figure 7 shows a visualization used by the data management operations team to compare the actual onboard execution times across multiple days (shown on the x-axis) for two sequences (`dmx_00103` and `dmx_00105`) against the times predicted by ground tools. As the figure shows, such a visualization makes it easy to see that the predictions for the `dmx_00103` sequence are much more accurate than the predictions for the `dmx_00105` sequence. This observation can then be used to further refine the models used by the ground tools to improve prediction times.

3.2 Monitoring communication windows

Figure 8 shows the rules used for monitoring Curiosity's *communication windows* [30]. A communication window defines the periods when the spacecraft communicates either directly with Earth, or with one of several relay orbiting spacecraft. Due to the importance of communication, monitoring rover performance during a window is of great interest to the operations team. To aid this

```
class CommWindowMonitor extends EvrMonitor {
  def wid(s:String,k:Int=5) = { val w = words(s)(k) ; w.slice(1,w.length).toInt }

  "prep" -- EVR('id → "EVR_BEGIN_PREP", 'sclk → 'P, 'msg → 'M) ⟼ {
    insert('Prep(wid('M.s,4), 'P.d))
  }

  "active" -- EVR('id → "EVR_BEGIN_ACTIVE", 'sclk → 'A, 'msg → 'M)
            & 'Prep('W, 'P) ⟼ {
    if (wid('M.s,2) == 'W.i) {
      insert('Active('W.i, 'A.d))
    }
  }

  "cleanup" -- EVR('id → "EVR_CLEANUP", 'sclk → 'C, 'msg → 'M)
             & 'Active('W, 'A) ⟼ {
    if (wid('M.s,1) == 'W.i) {
      insert('Cleanup('W.i, 'C.d)
    }
  }

  "print" -- 'Prep('W, 'P) & 'Active('W, 'A) & 'Cleanup('W, 'C) ⟼ {
    updateCSV('W.i, 'P.d, 'A.d, 'C.d)
    remove('Prep)
    remove('Active)
    remove('Cleanup)
  }
}
```

Fig. 8. Rules for communication windows

monitoring, we developed a set of rules that are used to generate summaries from rover telemetry; these summaries are in turn used to build useful visualizations that help the operational team monitor window performance.

A communication window consists of 3 phases – a *prep* phase, when onboard software configures the rover for the communication window (for instance, by turning on appropriate radios and retrieving from various cameras the images that will be sent to Earth); an *active* phase, during which the communication takes place; and a *cleanup* phase, for performing any cleanup actions (for instance, turning the radios off). Figure 8 shows four rules for processing telemetry for a communication window. The `prep` rule is triggered by the event `EVR_BEGINS_PREP` that indicates the start of a communication window; it adds the fact $\mathsf{Prep}(w, p)$ to the fact memory. Here w is the (unique) integer identifier associated with the window (this identifier is reported in the event message, and

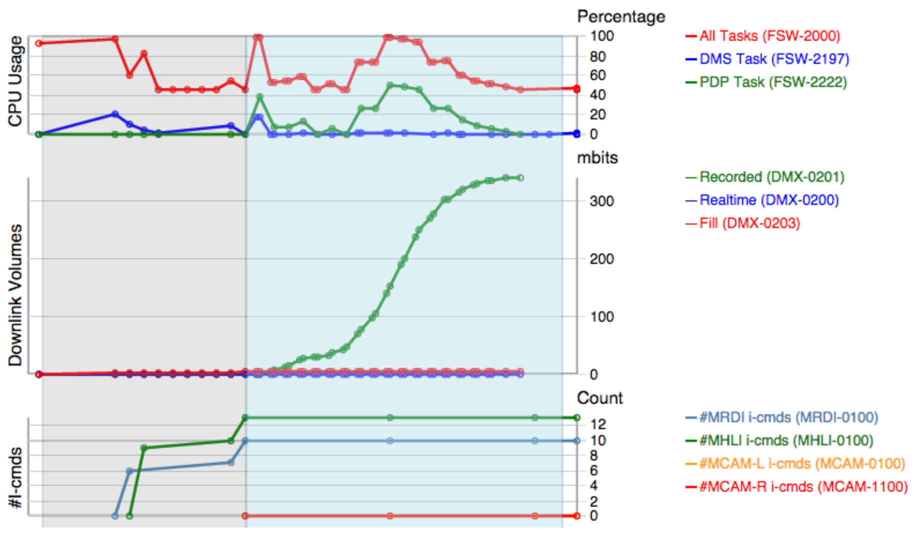

Fig. 9. Visualization showing communication window performance

is extracted by the helper *wid* function shown in the example), and p is the event timestamp (which indicates the time when prep started). Next, the `active` rule is used to detect when the active window begins; it is triggered by the `EVR_-BEGINS_ACTIVE` event, and adds the fact Active(w, a) to the fact memory, where w is the window identifier and a is the event timestamp. In a similar fashion, the `cleanup` rule is triggered by the `EVR_CLEANUP` event, and adds the Cleanup fact to the memory. Finally, the `print` rule updates a CSV file that defines all windows that have been performed on the rover; each row of this CSV file contains the window identifier and times when the prep, active and cleanup phases started.

The CSV files are used to build the visualization shown in Figure 9. This visualization uses the window definitions in the CSV file to provide context for assessing window performance. The top graph in the figure shows the percentage of CPU time taken up by various tasks, including the DMS and PDP tasks which respectively read files from flash memory and generate data packets for downlink. The middle graph shows the volume of data sent through the radio to an overhead orbiter; as the figure shows, the downlink rate varies over time, reaching a maximum rate when the orbiter is directly overhead (approximately halfway into the active session). Finally, the bottom graph shows the number of images fetched from each of the four cameras during window prep; in the example shown, the software fetched 13 images from the MHLI camera and 10 images from the MRDI camera (and no images from the other two cameras). Such visualizations are useful to the operations team, which can use them to determine, for instance, that the PDP task needs 40% of the CPU for packet generation when the radio is communicating at its highest rates. This knowledge helps guide decisions on whether or not to schedule other processor-intensive activities during communication windows.

4 Conclusion and future work

We have described the use of a rule-based engine, the LogFire Scala library, in building applications for processing telemetry. The applications are not limited to checking specific logical or temporal properties (as is common in runtime verification), but in addition generate summaries that are used to build effective visualizations supporting systems comprehension. We have described how these telemetry analysis applications are being deployed to process telemetry and build visualizations illustrating various aspects of the behavior of the Curiosity rover. The rule-based notation is shown to be sufficiently expressive and convenient for the task. The combination of a monitoring logic with a high-level programming language, in this case Scala, has turned out to be a crucial advantage.

Future work includes studying alternatives for defining the internal LogFire DSL. LogFire is a *deep embedding*, meaning that we have defined the abstract syntax for rules in Scala, in contrast to a *shallow embedding* as in [23], where we would have used Scala's own language constructs for writing rules. This again means that as a default there is no type checking of rules beyond what we program it to be. Another consequence is that user-defined names must be either strings or symbols (of the Scala class Symbol), and to get to their values, in case they represent event/fact parameters, the user has to apply get functions. A more elegant solution could potentially be achieved by defining the DSL as a syntactic extension of Scala, for example using the SugarScala tool available at [35] (part of SugarJ). Finally, the intention is to deploy LogFire more broadly, within MSL, as well as within other missions, as a general approach to log analysis and comprehension at JPL.

References

1. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittamplan, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA'05*. ACM Press, 2005.
2. Howard Barringer, Ylies Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. Quantified Event Automata - towards expressive and efficient runtime monitors. In *18th International Symposium on Formal Methods (FM'12), Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *LNCS*. Springer, 2012.
3. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Program monitoring with LTL in Eagle. In *Parallel and Distributed Systems: Testing and Debugging (PADTAD'04), Santa Fee, New Mexico, USA*, volume 17 of *IEEE Computer Society*, April 2004.
4. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.

5. Howard Barringer and Klaus Havelund. TraceContract: A Scala DSL for trace analysis. In *17th International Symposium on Formal Methods (FM'11), Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *LNCS*, pages 57–72. Springer, 2011.
6. Howard Barringer, Klaus Havelund, Elif Kurklu, and Robert Morris. Checking flight rules with TraceContract: Application of a Scala DSL for trace analysis. In *Scala Days 2011, Stanford University, California*, 2011.
7. Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *J. Log. Comput.*, 20(3):675–706, 2010.
8. David A. Basin, Felix Klaedtke, and Samuel Müller. Policy monitoring in first-order temporal logic. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, Proceedings*, volume 6174 of *LNCS*, pages 1–18. Springer, 2010.
9. Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In *Runtime Verification - 4th Int. Conference, RV'13, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *LNCS*, pages 59–75. Springer, 2013.
10. Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In Shmuel Ur, Eyal Bin, and Yaron Wolfsthal, editors, *Haifa Verification Conference, Haifa, Israel, November 13-16, 2005*, volume 3875 of *LNCS*, pages 208–223. Springer, 2006.
11. Eric Bodden. MOPBox: A library approach to runtime verification. In *Runtime Verification - 2nd Int. Conference, RV'11, San Francisco, USA, September 27-30, 2011. Proceedings*, volume 7186 of *LNCS*, pages 365–369. Springer, 2011.
12. Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-driven documents. In *IEEE Transactions on Visualization and Computer Graphics*, volume 17, pages 2301–2309, 2011.
13. Clips website: http://clipsrules.sourceforge.net.
14. D3 website: http://d3js.org.
15. Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Grenoble, France, April 7-11, 2014. Proceedings*, volume 8413 of *LNCS*, pages 341–356. Springer, 2014.
16. Robert B. Doorenbos. *Production Matching for Large Learning Systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1995.
17. Drools website: http://www.jboss.org/drools.
18. Drools functional programming extensions website:
https://community.jboss.org/wiki/
FunctionalProgrammingInDrools.
19. Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
20. Mario Fusco. Hammurabi - a Scala rule engine. In *Scala Days 2011, Stanford University, California*, 2011.
21. Jean Goubault-Larrecq and Julien Olivain. A smell of ORCHIDS. In *Proc. of the 8th Int. Workshop on Runtime Verification (RV'08)*, volume 5289 of *LNCS*, pages 1–20, Budapest, Hungary, 2008. Springer.
22. Sylvain Hallé and Roger Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Transactions on Services Computing*, 5(2):192–206, 2012.

23. Klaus Havelund. Data automata in Scala. In Martin Leucker and Ji Wang, editors, *8th International Symposium on Theoretical Aspects of Software Engineering, TASE 2014, Changsha, China, September 1-3. Proceedings*. IEEE Computer Society Press, 2014.
24. Klaus Havelund. Monitoring with data automata. In T. Margaria and B. Steffen, editors, *6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. Track: Statistical Model Checking, Past Present and Future (organized by Kim Larsen and Axel Legay), Corfu, Greece, October 8-11. Proceedings*, volume 8803 of *LNCS*, pages 254–273. Springer, 2014.
25. Klaus Havelund. Rule-based runtime verification revisited. *Software Tools for Technology Transfer (STTT)*, April 2014. Published online.
26. Klaus Havelund and Rajeev Joshi. Comprehension of spacecraft telemetry using hierarchical specifications of behavior. In Stephan Merz and Jun Pang, editors, *16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxemborg, November 3-7. Proceedings*, volume 8829 of *LNCS*, pages 187–202. Springer, 2014.
27. Jess website: http://www.jessrules.com/jess.
28. David Luckham, editor. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.
29. Lunar Atmosphere Dust Environment Explorer (LADEE) mission website: http://www.nasa.gov/mission_pages/LADEE/main.
30. Andre Makovsky, Peter Ilott, and Jim Taylor. Mars Science Laboratory Telecommunications System Design. *Descanso Design and Performance Summary Series*, Article 14, 2009.
31. Mars Science Laboratory (MSL) mission website: http://mars.jpl.nasa.gov/msl.
32. Patrick Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *Software Tools for Technology Transfer (STTT)*, 14(3):249–289, 2012.
33. Rooscaloo website: http://code.google.com/p/rooscaloo.
34. Volker Stolz and Eric Bodden. Temporal assertions using AspectJ. In *Proc. of the 5th Int. Workshop on Runtime Verification (RV'05)*, volume 144(4) of *ENTCS*, pages 109–124. Elsevier, 2006.
35. SugarJ website:
http://www.student.informatik.tu-darmstadt.de/∼xx00seba/projects/sugarj.