Comprehension of spacecraft telemetry using hierarchical specifications of behavior*

Klaus Havelund and Rajeev Joshi

Jet Propulsion Laboratory
California Institute of Technology
California, USA

Abstract. 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. We describe a tool for processing spacecraft telemetry that allows ground operators to impose structure on received telemetry in order to achieve a better comprehension of system state. A key element of our approach is the design of a domain-specific language that allows operators to express models of expected system behavior using partial specifications. The language allows behavior specifications with data fields, similar to other recent runtime verification systems. What is notable about our approach is the ability to develop hierarchical specifications of behavior. The language is implemented as an internal DSL in the Scala programming language that synthesizes rules from patterns of specification behavior. The rules are automatically applied to received telemetry and the inferred behaviors are available to ground operators using a visualization interface that makes it easier to understand and track spacecraft state. We describe initial results from applying our tool to telemetry received from the Curiosity rover currently roving the surface of Mars, where the visualizations are being used to trend subsystem behaviors, in order to identify potential problems before they happen. However, the technology is completely general and can be applied to any system that generates telemetry such as event logs.

1 Introduction

One of the key challenges in operating remote spacecraft is that ground operators must rely on limited telemetry visible on the ground in order to assess the health and operational status of the spacecraft. Such telemetry typically consists of a log of system events and sensor measurements (such as battery voltage or probe temperature) which, for the purposes of this paper, may be viewed as a sequence of timestamped records with named fields. Because this telemetry comprises essentially *all* the knowledge that ground operators have about a given

* 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.

spacecraft, processing this telemetry in a timely manner is of utmost importance to any mission. However, as spacecraft have become more autonomous and capable, and improvements in radio performance have resulted in greater downlink bandwidth, the resulting volume and complexity of the telemetry requires more automated processing tools so that any potential problems are diagnosed quickly and accurately. Unfortunately, currently such tools are developed by ground operators in an ad-hoc manner, typically using libraries developed by various subsystem teams that mine the telemetry to infer summaries that are of interest to that subsystem. These summaries are typically presented to ground operators using various visualization interfaces. While these tools have been overall quite effective, and the domain knowledge encoded in these libraries has led to many problems being identified early, the current approach of relying on ad-hoc scripts also makes the resulting tools fragile, hard for new team members to understand, and difficult to maintain. The maintainability issue is especially important for long-running missions that are expected to last many years.

To address this problem, this paper presents a declarative notation for expressing domain-specific knowledge about telemetry structure. In our formalism, the behavior of spacecraft subsystems may be expressed in terms of *behaviors*, in a language that resembles regular expressions, but with support for conjunction and data arguments to nonterminals. A key feature of our notation is that behaviors may be nested, since in our experience, most subsystems are usually viewed as a set of (possibly interrelated) hierarchical behaviors, and often viewed using visualization interfaces that allow behaviors of interest to be explored interactively. We demonstrate our approach in practice by showing how it is being applied to telemetry received from the Curiosity rover [3] currently on Mars. However, although applied to spacecraft operation, the techniques are fully general, and can be used for analysis of any form of event logs produced by a software system.

There has been much previous work in processing telemetry event logs in the field of *runtime verification* (RV), typically checking logs against user provided specifications, often expressed in some form of temporal logic. Such analysis may take place pre-deployment, as the system is being developed, or post-deployment, during operation. Orthogonally, the monitoring may be done online, processing telemetry on-board during execution, or offline by analyzing logs. We are concerned with post-deployment offline trace analysis. Most previous work, however, focuses on checking if a given event log satisfies a given specification or not (sometimes extending the Boolean domain with extra values to 3 or 4-valued logics, indicating grade of satisfaction). In our experience, coming up with formally verifiable properties is difficult in practice, especially for complex missions where design requirements were not in a formal notation to begin with. Thus our focus is more on providing a framework for performing *log comprehension*. This form of log comprehension can often be useful in identifying problems not easily formalized, but can also serve as a stepping stone to eventually writing (traditional) formal properties that may be checked for satisfaction.

To provide flexibility in expressing varied subsystem models, we have implemented our notation as an internal DSL (Domain-Specific Language), essentially an API, in the Scala programming language. Scala offers language constructs that makes definition of such APIs have the appearance of DSLs. Specifically we use Scala's implicit functions to define concrete syntax, and case classes to define abstract syntax. The resulting DSL is largely a so-called *deep embedding*, in contrast to a *shallow* embedding. In a deep embedding a particular program in the DSL is completely defined by an abstract syntax tree, which can be processed as an internal data structure. In contrast, in a shallow embedding host language constructs are made part of the DSL. A deep embedding makes it easier to analyze DSL programs. As we shall see, however, we do allow the DSL to contain arbitrary Scala code in limited positions, hence our approach is a mix of deep and shallow embedding.

We implement our DSL using the rule-based LogFire system [17], which is itself an internal Scala DSL. Rule-based systems, which have been extensively studied within the artificial intelligence (AI) community, allow formulation of rules of 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 case 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. LogFire is an implementation of the Rete algorithm [13] used in many AI rule systems.

The rule formalism, although very natural and expressive, turns out to be slightly verbose for writing log properties. The core problem can be illustrated by an example. Assume that one wants to monitor that the events E_1 and E_2 occur in that order. A rule system would have to explicitly create an intermediate fact E_1Seen representing the fact that E_1 has occurred. The issue is similar to that of state machines where all states must be explicitly defined. Regular expressions and temporal logics provide a solution to this problem. We here show a regular expression-like formalism which (i) makes this more convenient, and (ii) which allows for abstraction as discussed above. The DSL we present is defined as patterns that are translated to rules, in a similar manner as discussed in [17].

The paper is organized as follows. Section 2 outlines related work. Section 3 introduces briefly the rule-based system LogFire, and outlines the inconveniences in using this solution for this problem. Section 4 introduces the new DSL and its translation to rules. Section 5 presents the application to a spacecraft scenario, illustrating visualization of event abstractions. Finally, Section 6 concludes the paper.

2 Related Work

Several systems have been developed over the last decade for supporting monitoring of parameterized events. These systems support various formalisms, such as state machines [14, 19, 11, 7, 5], regular expressions [4, 19], variations over the μ-calculus [6], temporal logics [6, 19, 7, 15, 9, 10, 12], grammars [19], and rule-based systems [8, 17]. Some of these systems focus on being efficient. However, this efficiency is typically achieved at the price of some lack of expressiveness, as discussed in [5]. Our previous research has focused on more expressive formalisms, including rule-based systems, such as RULER [8] and more recently LOGFIRE [17]. Rule-based systems in general provide a rich formalism, which can be used to encode the kind of abstraction needed for our behavior definitions. LOGFIRE is based on the RETE algorithm [13], which is the basis for many rule-based systems developed over time, including for example DROOLS [2]. Standard rule systems usually enable processing of facts, which have a life time. In contrast, LOGFIRE in addition implements events, which are instantaneous, and which are needed for the kind of application presented in this paper. DROOLS supports a notion of events, which are facts with a limited life time. These events, however, are not as short-lived as possibly desirable in runtime verification. The event concept in DROOLS is inspired by the concept of *Complex Event Processing* (CEP), described by David Luckham in [18]. This concept is related to our approach using hierarchical behaviors. CEP is concerned with processing streams of events in (near) real time, where the main focus is on the correlation and composition of atomic events into complex (compound) events. TRACECONTRACT [7] and DAUT [16] are internal SCALA DSLs for trace analysis based on state machines. They allow for multi-transitions without explicitly naming the intermediate states, which corresponds to sequential composition of events. MOPBOX [11], and its more efficient successor PRM4J, are JAVA APIs for a set of algorithms implementing MOP's [19] functionality.

3 The LOGFIRE Rule Engine

As already mentioned, LOGFIRE is a SCALA API for writing rule-based programs in a manner that has the appearance of a DSL. It was originally created as a study of how the RETE algorithm could be used for runtime verification purposes, where the main goal is to check event traces against formalized specifications, and emit verdicts in a Boolean domain, stating whether the event stream satisfies the specification or not. In the following, we shall first illustrate the originally intended application, and in the subsequent sub-section we shall illustrate its use for abstraction, which is the topic of this paper. We then suggest that a more convenient solution is desirable for this objective.

3.1 LOGFIRE used for Verification

Consider a system that emits two kinds of events: $E_1(clk \rightarrow t_1)$ and $E_2(clk \rightarrow t_2)$, each being a named record (names are E_1 and E_2) with a field clk that is

```
class Verifier extends Monitor {
  "v1" -- 'E1('clk → 't1) ⟼ insert('E1Seen('t1))

  "v2" -- 'E2('clk → 't2) & not('E1Seen('t1)) ⟼ fail()

  "v3" -- 'E2('clk → 't2) & 'E1Seen('t1) ⟼ {
    if ('t2−'t1 > 5000) fail ()
  }
}
```

Fig. 1. A LogFire verifier

mapped to a time stamp t_i indicating the time when these events were generated. Suppose we want to enforce that E_2 can only occur after E_1, and furthermore, if E_2 occurs, it has to occur within 5 seconds of the occurrence of E_1. This property is shown in Figure 1. 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* E_1 and E_2 are short-lived instantaneous observations about the system being monitored, those submitted to the monitor. In contrast, *facts*, in this case E_1Seen, are long-lived pieces of information stored in the fact memory of the rule system, generated and deleted explicitly by the rules. In the monitor above the fact $E_1Seen(t_1)$ is used to represent the fact that the event $E_1(clk \to t_1)$ has been seen. The monitor contains three rules, named v_1, v_2 and v_3. Each rule has the form:

$$name \texttt{ -- } condition_1 \ \& \ldots \& \ condition_n \ \longmapsto \ action$$

Event and fact names, as well as parameter names are values of the Scala type *Symbol*, which contains quoted identifiers. The need for representing user-defined names as symbols is a consequence of the fact that LogFire is a deep embedding (we don't use Scala's names). Events and facts can have arguments specified in one of two ways: using *positional notation* or *using map notation*. Positional notation means just listing arguments as a list of patterns (identifiers or literals). In our example facts are represented using positional notation. The positional notation is convenient if events/facts carry few arguments. Map notation means considering the events/facts as being maps from field names to values. In our example event patterns are shown using map notation, assuming each event has a time stamp named *'clk*. When using map notation only fields relevant for the rule need be mentioned. An action is any Scala statement, that specifically for example can add or delete facts, or call failure methods.

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

The rules are to be read as follows. Rule v_1 states that when an E_1 event is observed, a fact, E_1Seen is created to record this. Rule v_2 states that an error is generated if an E_2 event is observed, but no E_1 event has been observed before that. Finally, rule v_3 states that in the case an E_1 event and subsequently an E_2 event is observed, the time difference must be within 5 seconds. A monitor can be applied as shown in Figure 2, which also shows an example of an error trace produced. 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. In this case the 5 second requirement is violated.

```
object ApplyMonitor {
  def main(args: Array[String]) {
    val m = new Verifier
    m.addMapEvent('E1)('clk → 1023)
    m.addMapEvent('E3)('clk → 3239)
    m.addMapEvent('E2)('clk → 7008)
  }
}
...

*** error:

[1] 'E1('clk→ 1023) ⟹ 'E1Seen(1023)
        rule: "v1" −− 'E1('clk→ 't1) ⟼ {...}

[3] 'E2('clk→ 7008) ⟹ 'Fail("ERROR")
        rule: "v3" −− 'E2('clk→ 't2) & 'E1Seen('t1) ⟼ {...}
```

Fig. 2. Applying a LogFire verifier

3.2 LogFire used for Abstraction

In this sub-section we shall illustrate how LogFire may be used to model the hierarchical behaviors of interest in our application. We consider a scenario with a top-level behavior (denoted *alpha*) that consists of an inner behavior (denoted *beta*) in parallel with a single event E_3. The behavior *beta* in turn consists of two events E_1 and E_2 that must occur in that order. Denoting the three atomic events as $E_1(clk \rightarrow t_1)$, $E_2(clk \rightarrow t_2)$, and $E_3(clk \rightarrow t_3)$, we want to record two facts: that E_2 occurs after E_1 is to be recorded as an occurrence of $beta(t_1, t_2)$, and that E_3 occurs either before or after (that is: in parallel with) $beta(t_1, t_2)$ is to be recorded as an occurrence of $alpha(t_1, t_2, t_3)$. The resulting monitor is shown in Figure 3.

```
class Abstracter extends Monitor {
  "a1" -- 'E1('clk → 't1) ⟼ insert('E1Seen('t1))

  "a2" -- 'E1Seen('t1) & 'E2('clk → 't2) ⟼ {
    remove('E1Seen);
    insert ('beta('t1, 't2))
  }

  "a3" -- 'E3('clk → 't3) ⟼ insert('E3Seen('t3))

  "a4" -- 'beta('t1, 't2) & 'E3Seen('t3) ⟼ {
    remove('E3Seen);
    insert ('alpha('t1, 't2, 't3))
  }
}
```

Fig. 3. A LogFire abstracter

The monitor contains four rules. The first rule, a_1, records when an $E_1(clk \rightarrow t_1)$ event is seen. Rule r_2 records a $beta(t_1, t_2)$ fact when an $E_2(clk \rightarrow t_2)$ event is seen after an $E_1(clk \rightarrow t_1)$ event. It also removes the intermediate event recording that E_1 was seen, in order to not clutter the set of facts generated. Rule a_3 records when an $E_3(clk \rightarrow t_3)$ event is seen, and finally rule a_4 creates the $alpha(t_1, t_2, t_3)$ fact. When applying the abstracter to the same event sequence as shown in Figure 2, instead of an error trace, we obtain a set of generated facts, as shown in Figure 4.

The main observation to be made, about this specification, as well as the verifier in Figure 1, is that it is inconvenient that we have to add (and delete) intermediate facts such as E_1Seen and E_3Seen explicitly, which makes these rules cumbersome to write and maintain. To avoid this problem, in the next section, we introduce notation that allows hierarchical events to be described more directly, in a form similar to the way one writes regular expressions, but with support for conjunctive composition and event parameters.

4 A DSL for Log Abstraction

We start by presenting our notation first in an idealized form, showing how the LogFire abstracter presented in the previous section can be written, as well as an idealized grammar. Subsequently we show how our notation is embedded as an internal Scala DSL, and we briefly sketch how our DSL implementation automatically generates LogFire rules from such descriptions.

```
object ApplyMonitor {
  def main(args: Array[String]) {
    val m = new Abstracter
    m.addMapEvent('E1)('clk → 1023)
    m.addMapEvent('E3)('clk → 3239)
    m.addMapEvent('E2)('clk → 7008)
  }
}
...

−−− facts: −−−−−−−−
'beta(1023,7008)
'alpha(1023,7008,3239)
−−−−−−−−−−−−−−−
```

Fig. 4. Applying a LogFire abstracter

4.1 A More Convenient Notation for Abstraction

Our proposed idealized syntax for the example shown in Figure 3 is shown in Figure 5. The model contains two so-called *behaviors*, one generating $beta(t_1, t_2)$ facts, and one generating $alpha(t_1, t_2, t_3)$ facts. The first rule shows an example of sequential composition, and reads as follows: when an $E_1(clk \rightarrow t_1)$ event is observed *followed by* an $E_2(clk \rightarrow t_2)$ event, a $beta(t_1, t_2)$ fact is generated. The second behavior shows an example of *parallel composition*, and reads: when a $beta(t_1, t_2)$ facts has been generated at some point, and a $E_3(clk \rightarrow t_3)$ event has been observed at some point, an $alpha(t_1, t_2, t_3)$ is generated, the ordering is unimportant. The fact generated, occurring to the left of the symbol `|==`, is referred to as the *behavior head*. The expression occurring on the right of the symbol `|==` is referred to as the *behavior expression*. Such behavior definitions have some resemblance to Prolog, but differ by being focused on events, and by supporting sequential composition as well as choice.

```
beta(t1, t2) |==  E1(clk -> t1)  >>  E2(clk -> t2)

alpha(t1, t2, t3) |==  beta(t1, t2)  &&  E3(clk -> t3)
```

Fig. 5. Abstracter using idealized syntax

The idealized grammar for our language is shown in Figure 6, using a form of extended BNF, where $\langle N \rangle$ denotes a non-terminal, $\langle N \rangle ::= \ldots$ defines the non-terminal $\langle N \rangle$, S^* denotes zero or more occurrences of S, $S^{*,*}$ denotes zero or more occurrences of S separated by commas (','), $S \mid T$ denotes the choice between S and T, and finally an expression in single quotes (such as '$\gg$') denotes a terminal symbol. A $\langle$*behaviorModel*$\rangle$ is a sequence of definitions, each being either a $\langle$*variableDef*$\rangle$ or a $\langle$*behaviorDef*$\rangle$. We already saw examples of behavior definitions in Figure 5. Variable definitions allow us to define convenient abbreviations for expressions which simplify the definition of a behavior expression and make it more readable. A $\langle$*behaviorExp*$\rangle$ can have one of six forms. We have already seen examples of sequential ($\gg$) and parallel (`&&`) composition. In addition behavior expressions can be composed with choice (`++`), meaning: one of the two sub-behaviors are observed. Behavior expressions can be grouped with parentheses. At the atomic level we distinguish between events observed and facts generated. They differ in two ways: event names are in all capital, and the arguments are given using map notation (see page 5), mapping field identifiers to identifiers representing their value. Fact names cannot be all capital, and arguments are provided in positional style.

$\langle behaviorModel \rangle ::= (\ \langle variableDef \rangle \mid \langle behaviorDef \rangle\ )^*$

$\langle variableDef \rangle ::= \langle id \rangle$ '`:=`' $\langle expr \rangle$

$\langle behaviorDef \rangle ::= \langle name \rangle$ '`(`' $\langle id \rangle^{*,*}$ '`)`' '$\models$' $\langle behaviorExp \rangle$

$\langle behaviorExp \rangle ::= \langle behaviorExp \rangle$ '$\gg$' $\langle behaviorExp \rangle$
$\mid \langle behaviorExp \rangle$ '`&&`' $\langle behaviorExp \rangle$
$\mid \langle behaviorExp \rangle$ '`++`' $\langle behaviorExp \rangle$
$\mid$ '`(`' $\langle behaviorExp \rangle$ '`)`'
$\mid \langle event \rangle$
$\mid \langle fact \rangle$

$\langle fact \rangle ::= \langle identifier \rangle$ '`(`' $\langle id \rangle^{*,*}$ '`)`'

$\langle event \rangle ::= \langle identifier \rangle$ '`(`' $\langle binding \rangle^{*,*}$ '`)`'

$\langle binding \rangle ::= \langle id \rangle$ '$\rightarrow$' $\langle id \rangle$

Fig. 6. Idealized grammar for abstracter DSL

4.2 Embedding as an Internal DSL in Scala

A variant of the idealized example shown in Figure 5, formalized in our internal Scala DSL, is shown in Figure 7. We have augmented the example with two

variable definitions, one defining the variable $'min$ as the minimal value to the two time stamps t_1 and t_3, and one defining the variable $'max$ as the maximal value to the two time stamps t_2 and t_3. These variables will be computed for each *alpha* fact generated, representing the time interval within which all important events occurred.

```
trait Example extends Abstracter {
  'tmin := { Math.min('t1.toDouble, 't3.toDouble) }
  'tmax := { Math.max('t2.toDouble, 't3.toDouble) }

  'beta('t1, 't2) |= 'E1('clk → 't1) >> 'E2('clk → 't2)

  'alpha('tmin, 'tmax) |= 'beta('t1, 't2) && 'E3('clk → 't3)
}
```

Fig. 7. Abstracter in SCALA DSL

As can be observed, the syntax has the same look and feel as the idealized syntax presented earlier. This is achieved by using some of SCALA's features for defining domain-specific languages, including implicit functions, possibility to define methods using non-alphanumeric symbols, and the possibility of leaving out dots and parentheses in calls of methods on objects. Generally, implicit functions automatically convert values of the argument type into values of the result type as follows. Whenever a SCALA expression fails to type check, the SCALA compiler will consult the implicit functions in scope and determine whether the application of a such will make the expression type check, and in this case the compiler will insert an application of the function (there can be no more than one such implicit conversion function, otherwise the SCALA compiler will complain).

This is illustrated with the *Abstracter* **trait** in Figure 8, shown in part, that behavior models extend. Consider the rule for generating $'beta('t_1,'t_2)$ in Figure 7. The SCALA compiler fails to make meaning out of this definition for a number of reasons. First of all, symbols like $'beta$ are being applied as if they were functions, and methods $\models$ and $\gg$ are being applied to objects on which they are not defined. The compiler searches the implicit functions, and finds that S will lift a symbol to an object that defines an *apply* method, which when applied generates a *Fact* object. Furthermore, the compiler finds that the implicit function F lifts such a *Fact* object to an object that defines a $\models$ method, which as argument takes a behavior expression. The behavior expression itself likewise is composed by calling the method $\gg$ on the firstly created behavior expression, without dot notation. When all implicit function calls and dots and parentheses have been inserted, the definition is equivalent to the following.

```
trait Abstracter {
  ...
  implicit def S(s: Symbol) = new {
    def apply(args: Any*): Fact = Fact(s, args.toList)
  }

  implicit def F(lhs: Fact) = new {
    def |=(rhs: BehExp) = ruleGen.generate(rhs, lhs)
  }
  ...
  trait BehExp {
    def ≫(n: BehExp): BehExp = SeqBehExp(this, n)
    def &&(n: BehExp): BehExp = ParBehExp(this, n)
    def ++(n: BehExp): BehExp = ChoBehExp(this, n)
     ...
  }
  ...
}
```

Fig. 8. Definition of a DSL

```
F(S('beta).apply('t1, 't2)).|=(
  (S('E1).apply('clk → 't1)).≫(S('E2).apply('clk → 't2)))
```

4.3 Rule Generation with Scala

The synthesized method call creates an abstract syntax tree, upon which a method is finally called, which generates LogFire rules. We shall not illustrate this in detail, but only outline the general idea. Figure 9 illustrates the method, *mkParRule*, that generates rules from a parallel composition of behavior expressions. The method takes four parameters: *pre*, which is a pre-condition, a fact that has to occur before the sub-behaviors of the parallel composition will be observed. The two sub-behaviors a and b, being arguments to the $\gg$ operator, and finally a post condition: a fact that is generated when the two sub-behaviors have been observed. Two intermediate facts P_1 and P_2 are first generated. Note how the parameters coming from the pre-condition are carried over such that generated facts accumulate all parameters seen so far. Rules for the two subexpressions a and b are subsequently generated, inheriting the pre-condition, and with respectively P_1 and P_2 as post-conditions: these facts are generated once the sub-behaviors have been observed. Finally, the main rule for parallel composition is generated, using the LogFire DSL. It gets an internal name generated by *newRuleId*(), and triggers once P_1 and P_2 have occurred, with the proper parameters. As a result the post-condition fact of the parallel composition is

generated and the intermediate facts are removed. The rules generated are very similar to the rules shown in Figure 3.

```
trait BehaviorMonitor extends Monitor {
  ...
  def mkParRule(pre: Fact, a: BehExp, b: BehExp, post: Fact) = {
    val P1 = new Fact(mkSym("par"), params(pre) ⊕ params(a))
    val P2 = new Fact(mkSym("par"), params(pre) ⊕ params(b))

    generate(pre, a, P1)
    generate(pre, b, P2)

    newRuleId() −− P1.s(params(P1): _∗) & P2.s(params(P2): _∗) ⟼ {
      insert (post.s(params(post): _∗)
      remove(P1.s)
      remove(P2.s)
    }
  }
  ...
}
```

Fig. 9. Synthesis of LogFire rules from a parallel behavior expression

5 Application: Mars-Earth Communication Sessions

In this section, we briefly describe how our notation is applied to analyze telemetry received from the Curiosity rover on Mars. In particular, we describe how we process telemetry related to the rover's direct communication sessions with Earth. A communication session with Earth consists of two behaviors that happen in parallel: a *tracking* behavior that moves the high-gain antenna to point towards the Earth and starts tracking to compensate for Mars's rotation, and a *configuration* behavior that turns on and configures the radios to communicate with the deep space network back on Earth. Since the Mars-Earth distance varies over time, this requires compensating for variable one-way light time, to ensure that the rover antenna is pointed and the radio ready when the signal from Earth arrives at Mars. Because communication is a critical behavior for the spacecraft, the operations team carefully monitors telemetry received from the rover to ensure adequate margins are being maintained for the signal arrival at Mars.

```
09:23:10  WINDOW_BEGINS("W25211", "HGA")
09:23:16  HGA_START_TRACK
09:23:18  XBAND_CONFIG("RECEIVE_ONLY")
09:23:30  HGA_EARTH_ACQUIRE
09:29:59  START_COMM
09:59:11  STOP_COMM
09:59:12  HGA_STOP_TRACK
09:59:27  WINDOW_CLEANUP

10:04:59  WINDOW_BEGINS("W60002", "HGA")
10:05:05  HGA_START_TRACK
10:05:06  XBAND_CONFIG("CARRIER_ONLY_2")
10:05:21  HGA_EARTH_ACQUIRE
10:07:03  START_COMM
10:16:46  STOP_COMM
10:16:48  HGA_STOP_TRACK
10:17:04  WINDOW_CLEANUP
```

Fig. 10. Sample event log from a communication session

Figure 10 shows a sample event log from a typical communication session[2]. Each window has an assigned unique identifier and a configuration parameter. As shown in the figure, our sample log consists of two back-to-back communication sessions performed on the rover. The first session (with identifier `W25211`) is configured as a `RECEIVE_ONLY` window and is used to uplink commands to the rover. One of the commands uplinked adds a second comm session (with identifier `W60002`) that is configured as `CARRIER_ONLY` and is used to send a 'beep' to Earth indicating successful receipt of commands from the first session. As shown in the figure, each session is bracketed by two events (named `WINDOW_BEGINS` and `WINDOW_CLEANUP`), and internally consists of two parallel behaviors: a *configuration* behavior that turns on the telecommunication hardware and configures it for communication, and a *tracking* behavior that points and tracks the high-gain antenna. The configuration behavior consists of three events: `XBAND_CONFIG`, indicating the start of radio configuration, `START_COMM`, indicating that the radio is ready to communicate, and `STOP_COMM`, indicating that the radio is being turned off. The tracking behavior also consists of three events: the `HGA_START_TRACK` event, indicating that pointing has commenced, the `HGA_EARTH_ACQUIRE` event, indicating that the antenna is pointed towards Earth, and the `HGA_STOP_TRACK` event, indicating that the antenna is terminating the tracking operation. As shown, each log event has an associated timestamp, along with optional arguments that provide additional information (such as the exact configuration used for the radio).

Figure 11 shows the behavior model for such a communication session in our notation. As shown in the figure, we define a SCALA **trait** called *CommSession*

[2] In the interests of readability, and to comply with guidelines about sharing telemetry details, we have omitted various technical details about radio configurations, and modified times and arguments from the original values.

```
trait CommSession extends Abstracter {
  'wid   := { getArg('wargs, 0) }
  'wtype := { getArg('wargs, 1) }
  'session ('wid, 'wtype, 'tws, 'twe, 'tts) |=
    (  'EVR('id → "WINDOW_BEGINS", 'lmst → 'tws, 'args → 'wargs)
    ≫ ('config ('ckind, 'tcs, 'tas, 'tce)   && 'track ('tts, 'tte))
    ≫ 'EVR('id → "WINDOW_CLEANUP", 'lmst → 'twe)
    )

  'kind := { getArg('cargs, 0) }
  'config ('kind, 'tcs, 'tas, 'tce) |=
    (  'EVR('id → "XBAND_CONFIG", 'lmst → 'tcs, 'args → 'cargs)
    ≫ 'EVR('id → "START_COMM", 'lmst → 'tas)
    ≫ 'EVR('id → "STOP_COMM", 'lmst → 'tce)
    )

  'track ('tts, 'tacq, 'tte) |=
    (  'EVR('id → "HGA_START_TRACK", 'lmst → 'tts)
    ≫ 'EVR('id → "HGA_EARTH_ACQUIRE", 'lmst → 'tacq)
    ≫ 'EVR('id → "HGA_STOP_TRACK", 'lmst → 'tte)
    )
}
```

Fig. 11. The model for a communication session in our notation

that extends the *Abstracter* **trait** defined in the previous Section 4, Figure 8. Nested within a *CommSession* is a parallel composition of the *config* and *track* behaviors, each of which is a sequential composition of the three log events described above. To extract event times, we rely on a SCALA library that processes spacecraft event logs and generates primitive LOGFIRE events (named *EVR*) that contain a map with a timestamp (denoted by key *lmst*) and optional event arguments (denoted by key *args*), which can be recovered using the *getArg* library function.

After the model has been interpreted, the resulting LOGFIRE rules generated may be used to process the event log shown above. The resulting nested structure is then saved in a web-readable format and processed by visualization tools, developed with D3 [1], resulting in the display shown in figure 12, which is integrated into an online *dashboard* used by the operations team. The figure shows the two *sessions* captured, composed sequentially, identified by $W25211$ and $W60002$. Each session in turn contains a *configuration* behavior and a *tracking* behavior, shown on top of each other, and each divided into two sections corresponding to the three events that define them. The visualization capability is crucial for presenting the hierarchical abstractions extracted by the tool

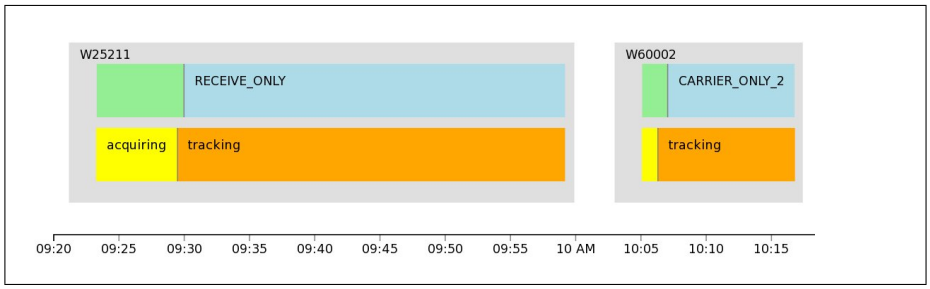

Fig. 12. Visualization of the two communication sessions from Figure 10

from the telemetry. In addition to communication sessions, we have also applied our notation for writing models for other rover subsystems, including behaviors describing the boot timeline, and certain behaviors involving on-board data management.

6 Conclusion and Future Work

We have described a notation for expressing domain-specific knowledge about subsystem behaviors that can be used for writing hierarchical models of telemetry streams (logs). These models are written using a Scala API that provides a great deal of flexibility. The formalism supports the following concepts: events, hierarchical abstraction, sequential, conjunctive and disjunctive composition, and allows users to write partial specifications that ignore events not of interest. The result of an analysis is a set of facts, rather than a boolean verdict. This allows existing models written in ad-hoc scripting languages to be easily expressed in our notation. The models are translated into a set of rules that can be used by the LogFire rule-based engine to automatically process telemetry received on the ground, allowing higher-level patterns to be matched and presented to ground operators. We have described how our method is applied to telemetry being received from the Curiosity rover, as part of an ongoing effort to build a system-wide dashboard for monitoring and analyzing spacecraft state. We are currently working on applying our methods to generate behavior models automatically from the hierarchical plans that are used to schedule rover activities every day. These models will then be applied to highlight discrepancies between predicted and actual rover activities. An interesting direction of research is to identify events that do not match any of the planned behaviors, since such events are often indicative of anomalous or unexpected behavior.

References

1. D3 website. http://d3js.org.

2. Drools website. http://www.jboss.org/drools.
3. Mars Science Laboratory (MSL) mission website. http://mars.jpl.nasa.gov/msl.
4. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA'05*. ACM Press, 2005.
5. H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. 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.
6. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
7. H. Barringer and K. 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.
8. H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *J. Log. Comput.*, 20(3):675–706, 2010.
9. D. A. Basin, F. Klaedtke, and S. Müller. Policy monitoring in first-order temporal logic. In T. Touili, B. Cook, and P. 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.
10. A. Bauer, J.-C. Küster, and G. 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.
11. E. 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. N. Decker, M. Leucker, and D. Thoma. Monitoring modulo theories. In E. Ábrahám and K. 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.
13. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
14. J. Goubault-Larrecq and J. 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.
15. S. Hallé and R. Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Transactions on Services Computing*, 5(2):192–206, 2012.
16. K. Havelund. Data automata in Scala. In M. Leucker and J. Wang, editors, *8th International Symposium on Theoretical Aspects of Software Engineering, TASE 2014, Changsha, China, September 1-3. Proceedings*. IEEE Computer Society Press, 2014.
17. K. Havelund. Rule-based runtime verification revisited. *Software Tools for Technology Transfer (STTT)*, April 2014. Published online.
18. D. Luckham, editor. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.
19. P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *Software Tools for Technology Transfer (STTT)*, 14(3):249–289, 2012.