`nfer` – A Notation and System for Inferring Event Stream Abstractions*

Sean Kauffman[1], Klaus Havelund[2], and Rajeev Joshi[2]

[1] University of Waterloo, Canada
[2] Jet Propulsion Laboratory, California Inst. of Technology, USA

Abstract. We propose a notation for specifying event stream abstractions for use in spacecraft telemetry processing. Our work is motivated by the need to quickly process streams with millions of events generated by the Curiosity rover on Mars. The approach builds a hierarchy of event abstractions for telemetry visualization and querying to aid human comprehension. Such abstractions can also be used as input to other runtime verification tools. Our notation is inspired by Allen's Temporal Logic, and provides a rule-based declarative way to express event abstractions. The system is written in Scala, with the specification language implemented as an internal DSL. It is based on parallel executing actors communicating via a publish-subscribe model. We illustrate the solution with several examples, including a real telemetry analysis scenario.

1 Introduction

A key challenge in operating remote spacecraft is that human operators must rely on telemetry to assess the status of the spacecraft. Telemetry can be thought of as an execution trace, a stream consisting of millions of discrete events. These event streams are difficult to interpret and validate because of their size and complexity. The current approach to analyzing spacecraft telemetry relies on ad-hoc scripts that are difficult to write and maintain. We propose a notation for computing abstractions of event streams, resulting in a hierarchy of interval abstractions, which is useful for telemetry visualization and querying to aid human comprehension. Our notation is inspired by interval logics, specifically Allen's Temporal Logic [2], commonly used in the planning and artificial intelligence (AI) domains. We extend a variation of this logic with a rule-based declarative way to express event abstractions. We also present a system named `nfer` (i**nfer**ence), written in Scala, which implements the notation as an internal Scala domain-specific language (DSL). The `nfer` system is based on concurrently executing actors communicating via a publish-subscribe model. We show the application of `nfer` to telemetry received from the Curiosity Mars rover.

Our system differs from traditional runtime verification (RV) systems, in which a program execution trace is checked against a user-provided specification.

* The research performed by the last two authors was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

RV usually results in a binary decision (true/false) as to whether the execution trace satisfies the specification, although variations on this theme have been developed. These include 3-valued logics (true, false, don't know) [8] and 4-valued logics (true, false, true-so-far, false-so-far) [6].

The remaining content of the paper is as follows. Section 2 introduces preliminary notation. Section 3 provides the problem statement and motivation for this work. Section 4 defines the `nfer` notation. Section 5 describes the implementation of the system in Scala, including the DSL. Section 6 illustrates the application of `nfer` to a scenario from the Mars Science Laboratory. Section 7 discusses related work. Finally, Section 8 concludes the paper.

2 Preliminary Notation

By $\mathbb{B}$ we denote the set of Boolean values $\{true, false\}$. By $\mathbb{N}$ we denote the set of natural numbers $\{0, 1, 2, \ldots\}$ and by $\mathbb{R}$ we denote the set of real numbers. For readability, we use the type $\mathbb{C} = \mathbb{R}$ to represent clock time stamps. By $A \times B$ we denote the cross product of types A and B. By $A \to B$ we denote the set of total functions from A to B. Functions in $A \to B$ can be denoted by lambda terms: $\lambda x.e$. A function of type $A \to \mathbb{B}$ is referred to as a predicate. Predicates with the same domain type can be composed with Boolean operators. For example, given $f : A \to \mathbb{B}$ and $g : A \to \mathbb{B}$, then $(f \wedge g)(x) = f(x) \wedge g(x)$. Given a set S, 2^S denotes the power set of S containing as elements all subsets of S. S^* denotes the set of finite ordered sequences over S where each sequence element is of type S. A sequence σ of length N is a function of type: $\{n \in \mathbb{N} | n < N\} \to S$. The i'th element of a sequence is denoted $\sigma(i)$. We say that a value v is in σ, denoted by $v \in \sigma$ iff $\exists\ i\ \in\ \mathbb{N}$ such that $\sigma(i) = v$. Given a set S, by S^n for a given $n \in \mathbb{N}$ $(n \geq 2)$ we denote the tuple type: $S\ \times\ S\ \times\ \ldots\ \times\ S$ (n times).

Let $\mathcal{I}$ be a set of identifiers, and let $\mathcal{V}$ be a set of values, including strings, integers, and floating point numbers[3]. A *map* is a partial function from identifiers to values with a finite domain, that is, a function of type $\mathcal{I} \xrightarrow{m} \mathcal{V}$. We use $\mathbb{M}$ to denote the type of all maps. The empty map is denoted by $[\,]$. We denote by $\mathbb{M}_\perp$ the extension of $\mathbb{M}$ with a bottom element: $\mathbb{M}_\perp = \mathbb{M} \cup \{\perp\}$. Here $\perp$ represents a "no map" value.

An event is a timestamped named tuple of the type $\mathbb{E} = \mathcal{I}\ \times\ \mathbb{C}\ \times\ \mathbb{M}$. An element (id, t, M) of type $\mathbb{E}$ is written as $id(t, M)$. A trace is a sequence of events. The type of traces is denoted by $\mathbb{T}$ and is defined by $\mathbb{T} = \mathbb{E}^*$. In our context a trace corresponds to a telemetry stream.

3 Problem Statement

In this section, we briefly outline the requirements to our specification language. We first illustrate a concrete problem with an example. Subsequently, we outline the specific needs.

[3] $\mathcal{V}$ can be any set of values that are part of monitored events.

3.1 Illustrating Example

Consider the trace shown on the left part of Figure 1, that we assume has been generated by a spacecraft[4]. The trace consists of a sequence of events, or EVent Reports (EVRs) as they are named in space mission operations, each with a name, a time stamp, and a list of parameters. The events in this particular trace represent such activities as a boot process starting, a boot process ending, downlink of data to ground, and operating the antenna and radio.

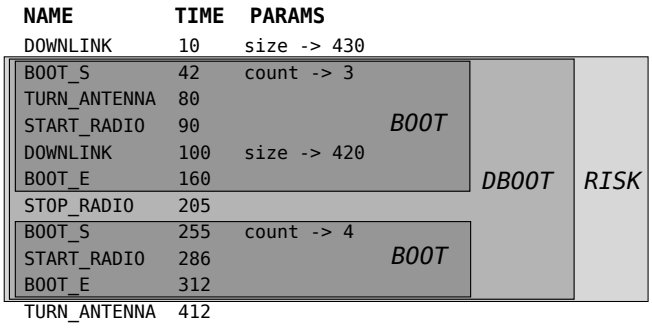

Fig. 1. An event trace and its abstractions

Our concern, in this case, is whether there is a downlink operation during a 5-minute time interval where the flight computer reboots twice. This scenario could cause a potential loss of downlink information. Notice the use of the term *interval*. We need a form of interval notation. We suggest imposing a structure on the trace, where these intervals are named and highlighted, as shown on the right part of Figure 1. Specifically, we want to identify the following intervals: A `BOOT` represents an interval during which the spacecraft software is rebooting. A `DBOOT` (double boot) represents an interval during which the spacecraft reboots twice within a 5-minute timeframe. A `RISK` represents an interval during which the spacecraft reboots twice and at the same time also attempts to downlink information.

Our objective now is to formalize the definition of such intervals in a specification language. Specifically, in this case, we need a rule-based formalism for formally defining the following three intervals:

1. A `BOOT` interval starts with a `BOOT_S` (boot start) event and ends with a `BOOT_E` (boot end) event.
2. A `DBOOT` (double boot) interval consists of two consecutive `BOOT` intervals, with no more than 5-minutes from the start of the first `BOOT` interval to the end of the second `BOOT` interval.
3. A `RISK` interval is a `DBOOT` interval during which a `DOWNLINK` occurs.

[4] The trace is artificially constructed to have no resemblance to real artifacts.

3.2 Desired Features

The specification language should allow a user to:

1. **define intervals** as a composition of other intervals/events. For example to define the label `BOOT` as an interval delimited by the events `BOOT_S` and `BOOT_E`, or to define a `DBOOT` to be composed sequentially of two `BOOT` intervals.
2. **refer to time stamps** associated with events, as well as generate and later read start and end times of generated intervals. It should be possible to define complex time constraints.
3. **refer to data** associated with events, as well as generate and later read data of generated intervals using a rich expression language. For example, a generated interval may have a datum value defined as the sum of two lower-level interval data.

We believe that Allen's Temporal Logic (ATL) [2], specifically its operators for expressing temporal constraints on time intervals, is a good starting point. In ATL, a time interval represents an action or a system state taking place over a period. A time interval has a name, a start time, and an end time. ATL offers 13 mutually exclusive binary relations. Examples are: $\mathit{Before}(i, j)$ which holds iff interval i ends before interval j starts, and $\mathit{During}(i, j)$ which holds iff i starts strictly after j starts and ends before or when j ends, or i starts when or after j starts and ends strictly before j ends. An ATL formula is a conjunction[5] of such relationships, for example, $\mathit{Before}(A, B) \wedge \mathit{Contains}(B, C)$. A model is a set of intervals satisfying such a conjunction of constraints. ATL is typically used in planning for generating a plan (effectively a model) from a formula, but ATL can also be used for checking a model against a formula, as described in [20].

Our objective is different from planning and verification. Given a trace, we want to generate a model (a set of intervals), guided by a specification that we provide, that represents a layered view of the trace, and is used for system comprehension.

4 The `nfer` Notation

4.1 Intervals

Before we more formally introduce the `nfer` notation, we shall introduce some further basic semantic concepts. As already mentioned in Section 2, a telemetry stream (for example received from a spacecraft) is a sequence of events, also referred to as a trace. In contrast to most runtime verification systems, however, the `nfer` notation does not directly operate on such traces. Instead, it operates on a set of *intervals* (defined below). We will provide the definition and intuition behind intervals, and how a trace is converted into an initial set of intervals, on which `nfer` operates.

[5] A limited form of disjunction is also allowed but not described here.

An interval represents a named section of a trace, spanning a certain time period. An interval can carry data as well, using a map. Concretely, an *interval* is a 4-tuple of the form (η, t_1, t_2, M), where $\eta \in \mathcal{I}$ is an interval name, $t_1, t_2 \in \mathbb{C}$ are time stamps[6] representing the start and end time of the interval, satisfying the condition $t_1 \leq t_2$, and M is a map in $\mathbb{M}$, the data that the interval carries. The type of all intervals is denoted by $\mathbb{I}$.

A *pool* is a set of intervals, that is, an element of type $\mathbb{P} = 2^{\mathbb{I}}$. A trace τ is converted into an initial pool by a function *init* of type $\mathbb{T} \rightarrow \mathbb{P}$:

$$init(\tau) = \{\ (\eta, t, t, M) \mid \eta(t, M)\ \in\ \tau\ \}$$

`nfer` subsequently transforms this initial pool of intervals to a pool containing as well all abstractions defined by the specification. In the following, we shall illustrate how such specifications are written.

4.2 Syntax of the `nfer` Notation

An `nfer` specification consists of a list of *labeling* rules of the form:

$$\eta \leftarrow \eta_1\ \oplus\ \eta_2\ \textbf{map}\ \Phi \tag{1}$$

where, $\eta, \eta_1, \eta_2\ \in\ \mathcal{I}$ are identifiers, $\oplus : \mathbb{C}^6 \rightarrow \mathbb{B}$ is a *clock predicate* on six time stamps, and $\Phi : \mathbb{M} \times \mathbb{M} \rightarrow \mathbb{M}_\perp$ is a *map function* taking two maps and returning a map or $\perp$. The syntax contains mathematical functions to simplify the presentation.

The informal interpretation of such a rule is as follows. Given a pool π, the rule generates a set of new intervals (a pool), each of the form (η, s, e, M), provided that in π there exist two intervals (η_1, s_1, e_1, M_1) and (η_2, s_2, e_2, M_2), such that the time constraint defined by $\oplus$ is satisfied: $\oplus(s_1, e_1, s_2, e_2, s, e)$, and such that the map function Φ produces a well-defined map as a function of the maps of the two input intervals: $M = \Phi(M_1, M_2) \neq \perp$. Note that the $\oplus$ time constraint constrains the start time s and end time e of the result interval as well. Hence, one can control the time values of the generated interval.

The time constraint can, for example, express that one interval ends before the other interval starts ($e_1 < s_2$), which is one of the Allen operators. Likewise, the map function can check whether the input maps M_1 and M_2 satisfy certain conditions: if not return $\perp$, but if so, return a new map to be part of the generated interval. The time constraint must evaluate to true and the resulting map not be $\perp$ for the rule to apply.

As an example, the following rule generates an abstraction interval named `BOOT` from a `BOOT_S` (boot start) event that occurs before a `BOOT_E` (boot end) event, and furthermore carries the boot count contained in the `BOOT_S` interval:

$$\texttt{BOOT} \leftarrow \texttt{BOOT\_S}\ \oplus\ \texttt{BOOT\_E}\ \textbf{map}\ \Phi$$

where the two functions $\oplus$ and Φ are defined as follows:

[6] Time stamps have no specified units.

$$\oplus(s_1, e_1, s_2, e_2, s, e) = e_1 < s_2 \wedge s = s_1 \wedge e = e_2$$
$$\Phi(m_1, m_2) = [count \mapsto m_1(count)]$$

Note how the resulting interval's start time s is constrained to be the start time of the `BOOT_S` event, and likewise the end time e is constrained to be the end time of the `BOOT_E` event. Below, we introduce a pre-defined set of candidate functions for $\oplus$ inspired by Allen logic to make specifications easier to write, allowing us instead to write this rule as follows (with the same Φ function and **before** denoting the $\oplus$ function above):

$$\texttt{BOOT} \leftarrow \texttt{BOOT\_S}\ \textbf{before}\ \texttt{BOOT\_E}\ \textbf{map}\ \Phi$$

4.3 Semantics of the `nfer` Notation

The semantics is provided in two steps. First the semantics for the core notation is provided, second a collection of derived symbols (called operators) are defined, which map to the core notation.

Semantics of core notation The semantics of the core notation is defined in three steps: the semantics R of individual rules on pools, the semantics S of a specification (a list of rules) on pools, and finally the semantics T of a specification on traces.

Let Δ be the type of rules. We define the semantics of labeling rules with the *interpretation* function R, with the type and definition below, and using the brackets $[\![\ \_\ ]\!]$ around syntax being given semantics:

$$\begin{array}{l}
R\,[\![\ _\ ]\!] \ :\ \Delta \rightarrow \mathbb{P} \rightarrow \mathbb{P} \\
R\,[\![\ \eta \leftarrow \eta_1 \oplus \eta_2\ \textbf{map}\ \Phi\ ]\!]\ \pi\ = \\
\quad \{\ (\eta,s,e,M) \in \mathbb{I}\ \mid \\
\qquad\qquad \exists\ s_1,e_1,s_2,e_2 \in \mathbb{C}\ \bullet\ \exists\ J,K \in \mathbb{M}\ \bullet \\
\qquad\qquad\quad (\eta_1,s_1,e_1,J) \in \pi\ \wedge \\
\qquad\qquad\quad (\eta_2,s_2,e_2,K) \in \pi\ \wedge \\
\qquad\qquad\quad \oplus(s_1,e_1,s_2,e_2,s,e)\ \wedge \\
\qquad\qquad\quad M = \Phi(J,K) \neq \bot \\
\quad \}
\end{array}$$

That is, given a rule δ and a pool π, a new pool is returned by: $R[\![\delta]\!]\pi$, containing (only) the new intervals generated. The definition reads as follows. A pool is returned containing intervals (η, s, e, M), where there exist two intervals in π, with names η_1 and η_2, and where the time constraint is satisfied, and the map resulting from applying Φ to the respective sub-maps is not $\bot$.

Next, we define the semantics of a list of rules, also referred to as a specification. For this we define the following one-step interpretation function S, which, given a set of rules and a pool, returns a new pool extending the input pool

with added abstraction intervals resulting from taking the union of the pools generated by each rule:

$$
\begin{array}{l}
S \; [\![\; \_ \;]\!] \; : \; \Delta^* \rightarrow \; \mathbb{P} \; \rightarrow \; \mathbb{P} \\
S \; [\![\; \delta_1 \; \ldots \delta_n \;]\!] \; \pi \; = \pi \; \cup \; R \; [\![\; \delta_1 \;]\!] \; \pi \; \cup \; \ldots \cup \; R \; [\![\; \delta_n \;]\!] \; \pi
\end{array}
$$

That is, given a specification $\delta_1 \ldots \delta_n$ and a pool π, a new pool is returned by: $S[\![\delta_1, \ldots, \delta_n]\!] \, \pi$. Finally, we define the semantics of a specification applied to a trace (a sequence of events). For this we define the interpretation function T, which given a list of rules and a trace returns a pool containing abstraction intervals:

$$
\begin{array}{l}
T \; [\![\; \_ \;]\!] \; : \; \Delta^* \rightarrow \; \mathbb{T} \; \rightarrow \; \mathbb{P} \\
T \; [\![\; \delta_1 \; \ldots \delta_n \;]\!] \; \tau \; = \\
\quad \textbf{least} \; \pi \; \in \; \mathbb{P} \; \textbf{such that} \\
\qquad \mathit{init}(\tau) \subseteq \; \pi \\
\qquad\quad \wedge \\
\qquad \pi \; = S \; [\![\; \delta_1 \; \ldots \delta_n \;]\!] \; (\; \pi \;)
\end{array}
$$

That is, given a specification $\delta_1 \ldots \delta_n$ and a trace τ, a pool of abstractions is returned by: $T[\![\delta_1, \ldots, \delta_n]\!]\tau$. The resulting pool is defined as the least fixed-point of $S[\![\delta_1 \ldots \delta_N]\!] : \mathbb{P} \rightarrow \mathbb{P}$ that includes $\mathit{init}(\tau)$, corresponding to repeatedly applying $S[\![\delta_1 \ldots \delta_N]\!]$, starting with $\mathit{init}(\tau)$, and until no new intervals are generated. Note that the least fixed-point exists since the semantic functions are monotonic. However, our simple iterative algorithm may not reach the least fixed-point if it is an infinite set. In practice, the `nfer` tool processes rules in a slightly different, but equivalent, order to improve performance.

4.4 Derived Forms

As hinted at the end of Section 4.2, a collection of $\oplus$ functions have been predefined, along with symbols (operators) denoting them. These symbols are shown in Table 1 together with their function definitions. Note that s_1 and e_1 are the start and end times for the left-hand interval, s_2 and e_2 are the start and end times for the right-hand interval, and s and e are the start and end times for the resulting interval. For all operators, except the **slice** operator, the start and end times of the resulting interval is the respectively left-most and right-most time stamps of the involved intervals. For the **slice** operator, the resulting time span denotes the overlapping section of two intervals. Note that the definitions of these operators differ from those of the Allen logic operators in [2], which are defined to be mutually exclusive, whereas `nfer`'s operators are not. This is due to our different practical needs.

The informal explanation of the operators is as follows: A **before** B: A ends before B starts; A **meet** B: A ends where B starts; A **during** B: all of A occurs during B; A **coincide** B: A and B occur at the exact same time; A **start** B:

Operator $\oplus$	$\oplus(s_1, e_1, s_2, e_2, s, e)$
before	$e_1 < s_2 \wedge s = s_1 \wedge e = e_2$
meet	$e_1 = s_2 \wedge s = s_1 \wedge e = e_2$
during	$s_1 \geqslant s_2 \wedge e_1 \leqslant e_2 \wedge s = s_2 \wedge e = e_2$
coincide	$s = s_1 = s_2 \wedge e = e_1 = e_2$
start	$s = s_1 = s_2 \wedge e = \max(e_1, e_2)$
finish	$s = \min(s_1, s_2) \wedge e = e_1 = e_2$
overlap	$s_1 < e_2 \wedge s_2 < e_1 \wedge s = \min(s_1, s_2) \wedge e = \max(e_1, e_2)$
slice	$s_1 < e_2 \wedge s_2 < e_1 \wedge s = \max(s_1, s_2) \wedge e = \min(e_1, e_2)$

Table 1. `nfer` operators

A starts at the same time as B; A **finish** B: A finishes at the same time as B; A **overlap** B: A and B overlap in time; A **slice** B: A and B overlap in time, and only the overlapping time span is returned. For the **before** operator, the `nfer` tool returns the shortest matching intervals, whereas the semantics specifies that all matching intervals are returned.

The next abbreviation concerns further time constraints a user may want to impose. The core rule notation, see (1) on page 5, allows for any time constraints to be expressed. Possible constraints include the just introduced relational operators, but also time spans, such as stating that an event B should follow an event A within 10 time units. We present the following shorthand for allowing the specification of additional time constraints in addition to the just introduced operators. Let $\odot \in \{\textbf{before}, \textbf{meet}, \textbf{during}, \textbf{coincide}, \textbf{start}, \textbf{finish}, \textbf{overlap}, \textbf{slice}\}$, and let $\odot_p$ denote the corresponding clock predicate. The following abbreviation is introduced:

$$\eta \leftarrow \eta_1 \ \odot \ \eta_2 \ \textbf{within} \ \Theta \ \textbf{map} \ \Phi$$

where $\Theta : \mathbb{C}^6 \rightarrow \mathbb{B}$ is a predicate on six time stamps. This is synonymous with:

$$\eta \leftarrow \eta_1 \ (\odot_p \wedge \Theta) \ \eta_2 \ \textbf{map} \ \Phi$$

The one operator (clock predicate) rule format (1) on page 5 presents a simple notation with a clean semantics. However, further convenient syntax allows rules containing more than one operator on the right-hand side, for example: $A \leftarrow (B \ \textbf{before} \ C) \ \textbf{overlap} \ D$. Such rules are mapped into the core form resulting in additional auxiliary rules. The internal Scala DSL described in Section 5 allows such enriched rules. Note that we shall allow time constraints (**within**) and map transformations (**map**) to be left out in rules, in which case they assume the default function values respectively $\lambda s_1, e_1, s_2, e_2, s, e.\ \mathit{true}$ and $\lambda m_1, m_2.\ [\,]$.

4.5 Example

As an example, we will formalize the three rules that were informally stated in Section 3.1. The specification similarly consists of three rules:

`BOOT` $\leftarrow$ `BOOT_S` **before** `BOOT_E` **map** (λ m_1,m_2 . [*count* $\mapsto$ m_1(*count*)])

`DBOOT` $\leftarrow$ `BOOT` **before** `BOOT` **within** (λ s_1,e_1,s_2,e_2,s,e . $e-s \leqslant 300$)
map *snd*

`RISK` $\leftarrow$ `DOWNLINK` **during** `DBOOT` **map** *snd*

The rules should be mostly self-explanatory (time is assumed measured in seconds). The first rule creates from the two sub-maps m_1 and m_2 a new map, mapping count to the same value as in m_1. The function *snd* selects m_2 from a binary tuple (m_1, m_2).

Let us illustrate how this specification is evaluated on the trace in Figure 1. This trace is first converted into an initial pool. The semantic S function on (page 7) will go through three iterations when applied to this initial pool before a fixed-point is reached. The added intervals in each iteration are as follows:

1 : { (`BOOT`, 42, 160, [*count* $\mapsto$ 3]), (`BOOT`, 255, 312, [*count* $\mapsto$ 4]) }
2 : { (`DBOOT`, 42, 312, [*count* $\mapsto$ 4]) }
3 : { (`RISK`, 42, 312, [*count* $\mapsto$ 4]) }

5 Implementation

In this section, we outline the `nfer` infrastructure and internal DSL, implemented in the Scala programming language.

5.1 The nfer Infrastructure

The `nfer` implementation is based on Scala actors communicating via asynchronous message passing through a publish/subscribe model built with Apache Kafka [16]. Figure 2 shows the `nfer` implementation's internal configuration corresponding to the double boot example from Section 4.5. The Kafka publish/subscribe framework is represented in the center by the Shared Telemetry Bus. Each actor is represented by a circle, with arrows showing the messages that are passed to the actor (those it subscribes to), as well as the messages the actor publishes back.

Specifically, each rule in an `nfer` specification results in an actor, which subscribes to events/intervals occurring on the right-hand side of the rule, and publishes the interval mentioned on the left-hand side of the rule to the shared bus. This means that rule actors are only passed events and intervals which are pertinent to their execution. For example, the RISK actor subscribes to both `DBOOT` intervals and `DOWNLINK` events, and publishes back `RISK` intervals. A special actor receives messages from the spacecraft and publishes them to the bus. When a rule actor publishes an interval, any subscribers will be notified and can build on this interval to create yet new intervals. The `nfer` notation is

declarative and the order in which rules are declared is unimportant. Likewise, the order in which actors execute is also unimportant, since the results of one actor cannot inhibit the behavior of any other actor. If the DSL offered a negation operator that would not be the case.

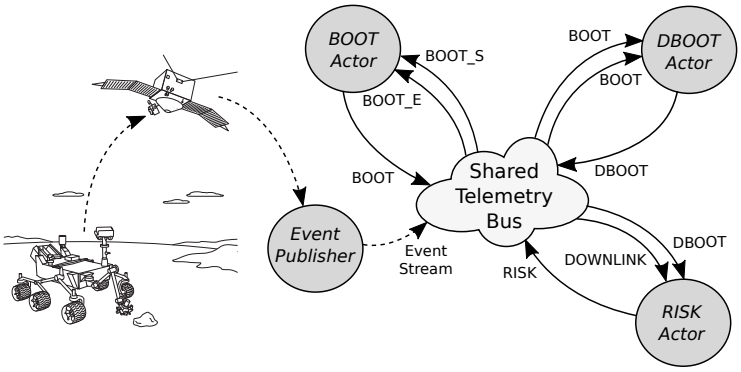

Fig. 2. Implementation of the example from Section 4.5

The implementation can process events *online*, as they come down to ground from the spacecraft, as well as events produced at an earlier point in time, and stored in a database. The full telemetry stream in principle includes all events from the start of the mission. Normally ground operators are only interested in recent events. However, there can be a need to analyze the telemetry stream from the start of the mission. It is not expedient to process all events in the full telemetry stream from the start of the mission whenever the `nfer` system is activated. Instead, `nfer` can be used to incrementally create intervals, which can then be stored for later use as an abstraction of the entire telemetry stream.

5.2 The Internal Scala DSL

This section introduces the internal Scala DSL for writing `nfer` specifications. Consider the double boot example written in the `nfer` notation in Section 4.5. This example can be written as follows in the internal DSL that we shall describe:

```
class DoubleBoot extends Nfer {
  "BOOT" :- ("BOOT_S" before "BOOT_E" map {
    case (m1,m2) ⇒ Map("count" → m1("count"))
  })

  "DBOOT" :- ("BOOT" before "BOOT" within 300 map (_._2))

  "RISK" :- ("DOWNLINK" during "DBOOT" map (_._2))
}
```

The complete specification is a class, named DoubleBoot, that extends the Nfer class, which provides all the operators needed for writing rules. The specification in the DSL has largely the same format as the specification in our notation. Some differences include the use of the symbol `:-` instead of ←, and the map function defined using Scala's partial function **case**-notation. Also, note that the constant 300 is automatically lifted by an *implicit* function (defined in the Nfer class) into a predicate on six time stamps with the expected semantics. Each rule in turn is essentially a function call having the side-effect of creating an actor that subscribes and publishes on the shared telemetry bus. For example, the first rule corresponds to a function call (series of function calls really) that will create an actor, which consumes `BOOT_S` and `BOOT_E` events (represented as intervals) from the telemetry bus and returns `BOOT` intervals back to the bus. Although it does not look like a normal function call, it is equivalent to the following call:

```
liftRuleName("BOOT").:−(
  (liftOperand("BOOT_S").before(liftOperand("BOOT_E"))).map {
    case (m1,m2) ⇒ Map("count" → m1("count"))
  )
```

This equivalence holds due to Scala's features for defining domain-specific languages. First of all, Scala allows method names to be non-alphanumeric, as for example :−. Second, Scala allows the omission of dots and parentheses in calls of methods on objects. For example, `"BOOT" :− (...)` is just another way of writing `"BOOT".:−(...)`. Finally, we notice that the method :− is called on the string object `"BOOT"`. However, no such method is defined on strings. Scala's implicit function concept can again be used here to lift the `"BOOT"` string to an object which defines a :− method. The following function (defined in the Nfer class) is applied automatically by the Scala compiler to resolve the typing conflict, as shown above:

```
implicit def liftRuleName(s: String) = new {
  def :−(op: Op) = makeRules(s, op)
}
```

The right-hand side of the rule contains the expression: `"BOOT_S"` before `"BOOT_E"`, which again is equivalent to: `"BOOT_S".before("BOOT_E")`, and again implicit lifting is needed. The following implicit function lifts `"BOOT_S"` to an object of type Op, on which methods like before are defined:

```
implicit def liftOperand(s: String) = new Op(s)
```

The Op class itself provides all the infix binary temporal operators, such as before, during, etc. as well as the functions within and map for defining time constraint and map functions (the latter two update variables holding these functions).

Note how these functions return new instances of the Op class such that further infix binary methods can be applied in a chain-like manner.

```
case class Op(s : String,  left :  Op = null, right:  Op = null, op: Fun = null)
{
  def before(e:  Op) = Op(..., this, e, BEFORE)
  def during(e:  Op) = Op(..., this, e, DURING)
   ...
}
```

Each instantiation of the Op class takes two arguments and an operator defining how they should be related. For example, BEFORE is defined as follows:

```
def BEFORE(...) {
  makeRule(...,
    { case (i1:  Interval ,  i2:  Interval ) ⇒ (i1 .end < i2.start )},
    { case (i1:  Interval ,  i2:  Interval ) ⇒ (i1 . start ,  i2 .end) })
}
```

The parameters to the makeRule function are (some dotted out): the name of the rule, two patterns (interval names essentially) that the generated actor subscribes to, the function evaluating the map, the function evaluating any added time constraints beyond the before, during, etc. constraints, and finally two functions (shown) defining respectively (1) the temporal operator, in this case an interval occurs before another if the end time of the first is less than the start time of the second, and (2) the boundary times of the generated interval as the start time of the first and the end time of the second.

6 Example Application to Warning Analysis

As noted earlier, we are currently applying the `nfer` tool to processing telemetry from the Curiosity rover. In this section, we briefly describe an application to a task, that is traditionally performed either manually or by ad-hoc scripts. We consider the problem of automatically labeling warning messages that are expected due to known idiosyncrasies of the system. EVRs produced by Curiosity are associated with a *severity* level, which is used to distinguish between expected and unexpected behavior. One of the severity levels is *WARNING*, which indicates potentially anomalous behavior. Unfortunately, due to various idiosyncrasies of hardware and software, there are several situations in which warning EVRs do not denote real anomalies. As a result, one of the roles of the ground operations team is to label those received warnings that are to be ignored; this work needs to be completed before the next plan can be uplinked to the spacecraft. To speed up analysis, we have implemented a set of rules that can label EVRs corresponding to known idiosyncrasies. As a result, ground operators can limit their attention to only unlabeled warning EVRs. We describe some of these rules below.

The first pair of rules capture a known (benign) race condition in the software caused by a thread reading from a shared buffer before another thread has finished its write. While race conditions can be serious, in this case, the effect is that the reading thread generates a warning, and ignores the data. Because the error was discovered late in the mission, and the impact is benign, no code fix was deemed necessary. The rule below looks for this known scenario by checking for an occurrence of `TLM_TR_ERROR` during execution of either a `MOB_PRM` or an `ARM_PRM` command. A command execution interval itself is defined by a pair of `CMD_DISPATCH` and `CMD_COMPLETE` events whose maps agree on the `"cmd"` key, which denotes the command name.

```
"cmdExec" :− ("CMD_DISPATCH" before "CMD_COMPLETE" map {
  case (m1, m2) ⇒
    if (m1("cmd") == m2("cmd")) Some(Map("cmd" → m1("cmd"))) else None
})

"telecom0208" :− ("TLM_TR_ERROR" during "cmdExec" map {
  case (_, m2) ⇒ if ((m2("cmd") == "MOB_PRM") || (m2("cmd") == "ARM_PRM"))
    Some(Map()) else None
})
```

The second rule involves a timing consideration. In this case, a power-on command fails and then recovers within 15 seconds. Since the behavior is predictable, and benign, the two warnings about command failure and subsequent recovery are labeled as expected. Note that for readability we have simplified the signature of the `delay15` function.

```
def delay15(s1,e1,s2,e2,s,e : Double): Boolean = e − s <= 15

"instCmdFail" :− ("INST_PWR_ON" before
    ("INST_CMD_FAIL" before "INST_RECOVER") within delay15)
```

As we illustrated in Section 5.2, this can be written more simply by just providing 15 as argument to the within function, which would have the same effect.

```
"instCmdFail" :− ("INST_PWR_ON" before
    ("INST_CMD_FAIL" before "INST_RECOVER") within 15)
```

The third set of rules labels a situation in which a warning about task starvation is expected whenever the `vdp` activity overlaps with a communications activity (labeled `comm`). In this case, we use the slice operator to identify the interval of overlap between the `vdp` and `comm` intervals:

```
"comm" :− ("WIN_BEGIN" before "WIN_END" map {
   case (m1, _) ⇒ Map("wid" → m1("wid"))
 })
```

```
"vdp" :– ("VDP_START" before "VDP_STOP")

"starvationOk" :– "TASK_STARVATION" during ("vdp" slice "comm")
```

7 Related Work

An earlier effort to develop a telemetry comprehension tool is described in [14], which provided a Scala DSL for writing a subset of the specifications offered in this paper. That work was inspired by yet earlier efforts using the rule-based system LogFire [13] for analyzing telemetry streams, as described in [15]. LogFire, however, offers a more traditional rule notation, which becomes verbose for writing the desired specifications (similar to state machines being more verbose than regular expressions).

Interval logics are common in the planning domain. Allen formalized his algebra [2], which has come to be known as ATL, for modeling time intervals. He argued that it was necessary to model relative timing with significant imprecision, and proposed his algebra's use in planning systems [3]. Many other planning languages have been proposed which rely on these same concepts, including PDDL [18] and ANMLite [7].

The concepts introduced and formalized by these interval logics are useful for modeling telemetry data, but the languages themselves have been principally designed for planning, not verification. Some efforts have been made to adapt them to that role, however. An effort is described in [22], where the suitability of the ANMLite system for verification was evaluated, with some positive results, but it was ultimately concluded that the solver techniques were not yet mature enough to be useful. A translation from LTL to PDDL is described in [1] as a means to leverage PDDL's solver for verification.

Conversely, [20] defines a translation of a modified ATL to LTL for monitoring. It is concluded, however, that this approach is impractical since the generated monitoring automata become too large, even for small ATL formulas. Instead, they introduce a simple algorithm for that purpose using a state machine for each relationship. For example, a state machine is created for $\mathit{Before}(A, B)$, which is violated if a B is seen before an A. Our work differs in some respects: (i) Instead of monitoring ATL relationships for verification, we generate a relationship hierarchy for supporting system execution comprehension. (ii) We handle parameterized intervals. (iii) We allow any constraints on time and parameter values, not just the 13 ATL constraints. (iv) In their system, an interval is unique, while in `nfer` it can occur multiple times. Other interval logics have been designed specifically for verification purposes, such as Interval Temporal Logic (ITL) [19], the Duration Calculus (DC) [12], and Graphical Interval Logic (GIL) [9].

Our work has strong similarities to data-flow (data streaming) languages. A recent example is QRE [4], which is based on regular expressions, and offers

a solution for computing numeric results from traces. QRE allows the use of regular programming to break up the stream for modular processing, but is limited in that the resulting sub-streams may only be used for computing a single quantitative result, and only using a limited set of numeric operations, such as sum, difference, minimum, maximum, and average, to achieve linear time (in the length of the trace) performance. Our approach is based on Allen logic, and instead of a numeric result produces a set of named intervals, useful for visualization (and thereby systems comprehension). Furthermore, data arguments to intervals can be computed using arbitrary functions.

RV systems have been developed which aggregate data as part of the verification [11, 5]. Statistical model checking [17] is an approach collecting statistical information about the degree to which a specification is satisfied on multiple traces. Pushing statistical analysis further, in specification mining [10, 21] the user provides no specification, and the system learns one by sampling nominal runs or by static analysis of the source code. This approach relieves the user of writing specifications and allows them to better understand the behavior of the software.

8 Conclusion

We have introduced the `nfer` rule-based notation and system for labeling event streams. The result of a labelling is a set of intervals: named sections of the event stream, each including a start time, and end time, and a map holding data selected from the events and sub-intervals making up the interval. Typically intervals are built on top of intervals, forming a hierarchy of abstractions. The result can for example be visualized, and can generally help engineers to better comprehend the structure of an event stream. The `nfer` system is implemented as an internal Scala DSL. Each interval-generating rule spawns an actor, that subscribes to events and/or sub-intervals, and publishes new intervals in the publish/subscribe architecture. Future work includes optimizing the implementation; handling missing telemetry; support for visual entering of rules and visualization of results; improving the internal Scala DSL; and allowing rules to be written in Python (commonly used by engineers) and encoded in JSON. The problem has been inspired by actual planetary space mission operations, specifically the Mars Curiosity rover, and the solution is being evaluated for use by the next Mars rover mission in 2020.

References

1. Albarghouthi, A., Baier, J.A., McIlraith, S.A.: On the use of planning technology for verification. In: Proc. of the ICAPS Workshop on Verification & Validation of Planning & Scheduling Systems (VVPS). Citeseer (2009)
2. Allen, J.F.: Maintaining knowledge about temporal intervals. Communications of the ACM 26(11), 832–843 (1983)
3. Allen, J.F.: Towards a general theory of action and time. Artificial intelligence 23(2), 123–154 (1984)
4. Alur, R., Fisman, D., Raghothaman, M.: Regular programming for quantitative properties of data streams. In: Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Eindhoven, The Netherlands. LNCS, vol. 9632, pp. 15–40. Springer (April 2016)
5. Basin, D., Harvan, M., Klaedtke, F., Zălinescu, E.: MONPOLY: Monitoring usage-control policies. In: 2nd Int. Conference on Runtime Verification (RV'11). LNCS, vol. 7186, pp. 360–364. Springer (2011)
6. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In: 7th Int. Workshop on Runtime Verification (RV'07). LNCS, vol. 4839, pp. 126–138. Springer (2007)
7. Butler, R.W., Siminiceanu, R.I., Muno, C.: The ANMLite language and logic for specifying planning problems. Report 215088, 23681–2199 (2007)
8. Chen, F., Roşu, G.: MOP: an efficient and generic runtime verification framework. In: ACM SIGPLAN Notices. vol. 42, pp. 569–588. ACM (2007)
9. Dillon, L.K., Kutty, G., Moser, L.E., Melliar-Smith, P.M., Ramakrishna, Y.S.: A graphical interval logic for specifying concurrent systems. ACM Trans. Softw. Eng. Methodol. 3, 131–165 (1994)
10. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming 69(1), 35–45 (2007)
11. Finkbeiner, B., Sankaranarayanan, S., Sipma, H.: Collecting statistics over runtime executions. Formal Methods in System Design 27(3), 253–274 (2005)
12. Hansen, M.R., Van Hung, D.: A theory of duration calculus with application. In: Domain modeling and the duration calculus, LNCS, vol. 4710, pp. 119–176. Springer (2007)
13. Havelund, K.: Rule-based runtime verification revisited. Software Tools for Technology Transfer (STTT) 17, 143–170 (2015)
14. Havelund, K., Joshi, R.: Comprehension of spacecraft telemetry using hierarchical specifications of behavior. In: 16th Intl. Conference on Formal Engineering Methods (ICFEM), Luxembourg. LNCS, vol. 8829. Springer (Nov 2014)
15. Havelund, K., Joshi, R.: Experience with rule-based analysis of spacecraft logs. In: Proc. of the 3rd Intl. Workshop on Formal Techniques for Safety-Critical Systems (FTSCS'14). Communications in Computer and Information Science, vol. 476, pp. 1–16. Springer (2014)
16. Kreps, J., Narkhede, N., Rao, J.: Kafka: A distributed messaging system for log processing. In: Proc. of the 6th Int. Workshop on Networking Meets Databases (NetDB'11). pp. 1–7. ACM (2011)
17. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: 1st Int. Conference on Runtime Verification (RV'10). LNCS, vol. 6418. Springer (2010)

18. McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D.: PDDL-the planning domain definition language (1998)
19. Moszkowski, B.C.: A temporal logic for multilevel reasoning about hardware. IEEE Computer 18, 10–19 (1985)
20. Rosu, G., Bensalem, S.: Allen linear (interval) temporal logic - translation to LTL and monitor synthesis. In: 18th Int. Conference on Computer Aided Verification (CAV'06). LNCS, vol. 4144, pp. 263–277. Springer (2006)
21. Shoham, S., Yahav, E., Fink, S.J., Pistoia, M.: Static specification mining using automata-based abstractions. IEEE Trans. Software Eng. 34(5), 651–666 (2008)
22. Siminiceanu, R., Butler, R.W., Muñoz, C.A.: Experimental evaluation of a planning language suitable for formal verification. In: 5th Int. Workshop on Model Checking and Artificial Intelligence (MoChArt'08), LNCS, vol. 5348, pp. 132–146. Springer (2009)