Formal Methods in System Design manuscript No.
(will be inserted by the editor)

Inferring Event Stream Abstractions

Sean Kauffman · Klaus Havelund · Rajeev Joshi · Sebastian Fischmeister

Received: date / Accepted: date

Abstract We propose a formalism 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 e.g. 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. We present an algorithm for applying specifications to an event stream and explore modifications to improve the algorithm's asymptotic complexity. The system is implemented in both Scala and C, with the specification language implemented as internal as well as external DSLs. We illustrate the solution with several examples, a performance evaluation, and a real telemetry analysis scenario.

Keywords telemetry comprehension · event stream processing · Allen logic · temporal logic · runtime verification

The research performed by K. Havelund and R. Joshi and part of the research performed by S. Kauffman was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The research performed by K. Havelund was furthermore supported by AFOSR Grant FA9550-14-1-0261.

S. Kauffman
University of Waterloo, Waterloo, Canada
E-mail: skauffma@uwaterloo.ca

K. Havelund
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, USA
E-mail: klaus.havelund@jpl.nasa.gov

R. Joshi
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, USA
E-mail: rajeev.joshi@jpl.nasa.gov

S. Fischmeister
University of Waterloo, Waterloo, Canada
E-mail: sfischme@uwaterloo.ca

1 Introduction

A key challenge in operating remote spacecraft is that human operators must rely on received telemetry to assess the status of the spacecraft. Telemetry can be thought of as an execution trace: a stream consisting of discrete events, each having a name, a time stamp, and carrying data. Telemetry streams can contain millions of events and can therefore be difficult to comprehend by humans, as well as interpret against the higher-level execution plans submitted to the spacecraft. The current approach to analyzing spacecraft telemetry for missions like the Mars Science Laboratory (MSL), and specifically from its Curiosity rover, relies on ad-hoc scripts that are labor intensive to write and maintain. We propose a formalism for specifying *interval abstractions* of event streams, with a semantics that produces a set of intervals from a trace. Such abstractions can be useful for telemetry visualization[1] and querying to aid human comprehension. Our formalism 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 formalism, named `nfer`, for expressing event abstractions.

The `nfer` formalism and implementation has commonalities with classical rule-based systems known from AI. Our early work on the trace abstraction problem was effectively done using a rule-based system, as documented in [37, 36]. However, we learned that a rule system does not represent meta-level constraints well, i.e., properties that hold on the collection of all facts produced by the rule system. An example of a meta-level constraint is the notion of minimality: *do not create an interval if an interval has already been generated with the same name in the same time period*. Such meta-constraints and, in particular, the minimality constraint, turn out to be crucial to reduce the complexity of trace analysis. A similar observation can be made about Prolog. As an experiment, we formulated the article's guiding example in LogFire [35], our homegrown rule-based system used in [37, 36]; and in Prolog, and compared them to the Scala and C versions of `nfer`. The four systems were applied to an event trace of 10,000 randomly generated events. LogFire had to be aborted after running more than a week. Prolog took 64 hours to finish. The Scala version and C version, both of which implement minimality, finished in 1.8 and 0.05 seconds, respectively.

Our system differs from traditional runtime verification (RV) systems, in which a program execution trace is checked against a user-provided specification. 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, including 3-valued logics [15] and 4-valued logics [12]. In contrast, the result of running `nfer` on an event stream is a set of named and timed intervals carrying data collected from the trace, representing abstractions of the trace. Such abstractions can be visualized to support trace comprehension, or can be considered as input to further analysis.

This article extends a previously published conference version of the work [40]. The earlier version focused on the semantics and derived forms of inclusive rules (rules that produce intervals based on the existence of other intervals), the internal Scala DSL, and a case study on MSL. The work presented here expands on the semantics of inclusive rules, introduces exclusive rules (rules that produce intervals based on the non-existence as well as existence of other intervals), and provides a precise definition and implementation of minimality. In contrast to the prior version, this article introduces an external DSL with implementations in both Scala and C. We also propose an algorithm for applying `nfer` specifications to a trace,

[1] Visualization of information is e.g. at JPL considered an important approach to aid humans in daily spacecraft operations.

study its complexity, evaluate its performance, and suggest modifications that can be used to better scale the approach. The case study on MSL is included in this work, along with performance results from it and two other datasets.

The remaining contents of the article are as follows. Section 2 introduces preliminary notation. Section 3 provides the problem statement and motivation for this work. Section 4 defines the `nfer` formalism. Section 5 presents an algorithm for applying an `nfer` specification to an event stream. Section 6 describes the implementation of the system, including the external DSL. Section 7 illustrates the application of `nfer` to a scenario from the Mars Science Laboratory. Section 8 introduces modifications to the `nfer` algorithm to improve its execution time, including an experimental evaluation of their performance. Section 9 discusses related work. Finally, Section 10 concludes the article.

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 measured in continuous (or dense) time. 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 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 $\mathscr{N}$ be a set of names (identifiers), and let $\mathscr{V}$ be a set of values, including strings, integers, and floating point numbers[2]. A *map* is a partial function from names to values with a finite domain, that is, a function of type $\mathscr{N} \xrightarrow{m} \mathscr{V}$. 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 $\mathbb{E} = \mathscr{N} \times \mathbb{C} \times \mathbb{M}$. 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 for our specification formalism. We first illustrate a concrete problem with an example. Subsequently, we outline the requirements. Consider the trace (telemetry stream) shown on the left part of Figure 1, that we assume has been generated by a spacecraft[3]. 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 activities such

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

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

as a boot process starting, a boot process ending, downlink of data to ground, and operating the antenna and radio. Our goal is to produce higher level views of this trace, which will make it easier to understand the meaning of its contents. One particular 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 suggest imposing a structure on the trace, where such 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 where the flight computer is rebooting. A `DBOOT` (double boot) represents an interval where the flight computer reboots twice within a 5-minute timeframe. A `RISK` represents an interval where the flight computer reboots twice while the downlink software is also attempting to send data to Earth. Our objective now is to formalize the definition of these intervals in a specification. In this case, we need a formalism for defining the following three intervals:

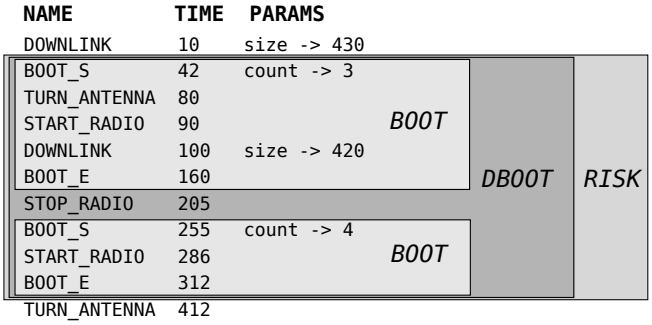

Fig. 1 An event trace and its abstractions

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.

The specification formalism 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 specify start and end times of generated intervals. It should be possible to define complex time constraints; and
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 have found that Allen's Temporal Logic (ATL) [2], specifically its operators for expressing temporal constraints on time intervals, is a useful 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 include: $Before(i, j)$ which holds iff interval i ends before interval j starts, and $During(i, j)$ 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[4] of such relationships, for example, $Before(i, j) \wedge During(j,k)$. 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 [55]. Our objective is different from planning and verification. Given a trace, we want to generate a set of intervals, guided by a specification that we provide. Each interval represents an abstraction of the original trace, either of low-level events, or of other lower-level intervals. As such, the set of resulting intervals represents a hierarchical abstraction of the original trace, useful for human comprehension and further automated processing.

4 The `nfer` Formalism

This section describes the semantic foundations of `nfer`. The syntax given in this section forms part of the theory of `nfer`, in contrast to the domain-specific language (DSL) introduced in Section 6.1 that is intended for practical use. We first introduce the notion of intervals, the fundamental data structure processed by `nfer` specifications. Subsequently the core formalism is introduced including syntax and semantics, followed by derived forms which map to the core form. Finally, we present an example.

4.1 Intervals

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` formalism does not directly operate on such traces *from a semantics point of view*. 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.

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 $\eta \in \mathcal{N}$ is an interval name, $t_1, t_2 \in \mathbb{C}$ are time stamps[5] 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 $\mathbb{T} \rightarrow \mathbb{P}$:

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

The `nfer` system subsequently transforms this initial pool of intervals to a pool also containing the abstractions defined by the specification. We say that we are annotating the original trace with *labels* (interval names). In the following section, we illustrate how such specifications are written.

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

[5] Time stamps have no specified units and their interpretation depends on the specification.

4.2 Syntax of the `nfer` Formalism

An `nfer` specification consists of a list of declarative *labeling* rules taking two forms: inclusive and exclusive. The application of a rule results in a set of intervals, which is the set of all possible intervals filtered to include only those that match the constraints specified by the rule.

4.2.1 Inclusive rules

The first form of labeling rule, called an *inclusive* rule, defines a new interval by the presence of two existing intervals:

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

where, $\eta, \eta_1, \eta_2 \in \mathscr{N}$ are identifiers, $\oplus : \mathbb{C}^6 \to \mathbb{B}$ is a *clock predicate* on six time stamps (two for each of $\eta, \eta_1,$ and η_2), and $\Phi : \mathbb{M} \times \mathbb{M} \to \mathbb{M}_\perp$ is a *map function* taking two maps and returning a map or returning $\perp$, which represents non-satisfaction of a constraint on the maps. The syntax presented here contains mathematical functions to simplify the presentation.

The informal interpretation of an *inclusive* 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 defines 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 corresponds to one of the Allen operators. Likewise, the map function can check whether the input maps M_1 and M_2 satisfy certain conditions: if they do not, the map function returns $\perp$, but if they do, it returns a new map that is part of the generated interval. The time constraint must evaluate to true and the result of the map function must 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) interval that occurs before a `BOOT_E` (boot end) interval, 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:

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

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. In Section 4.4, 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):

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

4.2.2 Exclusive rules

The second form of labeling rule, called an *exclusive* rule, defines an interval by the presence of one interval and the absence of a second:

$$\eta \leftarrow \eta_1 \textbf{ unless } \ominus\ \eta_2 \textbf{ map } \Phi \tag{2}$$

where, $\eta, \eta_1, \eta_2 \in \mathscr{N}$ are identifiers, $\ominus : \mathbb{C}^4 \rightarrow \mathbb{B}$ is a *clock predicate* on four time stamps (two for each of η_1 and η_2, while η is constrained implicitly), 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 informal interpretation of an exclusive rule is as follows: Given a pool π, the rule generates a set of new intervals (a pool), each of the form (η, s_1, e_1, M_1), provided that in π there exists an interval (η_1, s_1, e_1, M_1), and there does not exist an interval (η_2, s_2, e_2, M_2), such that (i) the time constraint defined by $\ominus$ is satisfied: $\ominus(s_1, e_1, s_2, e_2)$, (ii) the map function Φ produces a well-defined map as a function of the maps of the two input intervals: $M = \Phi(M_1, M_2) \neq \perp$, and (iii) the second interval ends before the first ends ($e_2 < e_1$).

For example, the time constraint can express that the first interval begins at the same time the second interval ends ($s_1 = e_2$). Likewise, the map function can check whether the input maps M_1 and M_2 satisfy conditions, and return $\perp$ if not. If an η_1 interval exists, and no η_2 interval exists for which both the time constraint is true and the map function is not $\perp$, then a new η interval is generated. Unlike the first form, the start and end times and the map of the new interval cannot be controlled by the labeling rule, but are copied from the existing η_1 interval. Also unlike the first rule, the second interval must end before the first interval ends. This constraint ensures that exclusive rules are monotone – produced facts will not later be retracted.

As an example, the following rule generates an abstraction interval named `BOOT_OK` from a `BOOT` interval, if no interval named `FAILURE` with the same value of the map key *bootId* exists, that starts after the `BOOT` begins and ends before the `BOOT` ends:

$$\texttt{BOOT\_OK} \leftarrow \texttt{BOOT} \textbf{ unless } \ominus\ \texttt{FAILURE} \textbf{ map } \Phi$$

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

$$\ominus(s_1, e_1, s_2, e_2) = s_1 \leq s_2 \wedge e_2 \leq e_1$$
$$\Phi(m_1, m_2) = \textbf{if } m_1(bootId) = m_2(bootId) \textbf{ then } [\,] \textbf{ else } \perp$$

Like in the first form, we introduce a pre-defined set of candidate functions for $\ominus$ which make specifications easier to write. The above rule could be rewritten with the pre-defined function **contain** denoting the $\ominus$ function above, and the same Φ function:

$$\texttt{BOOT\_OK} \leftarrow \texttt{BOOT} \textbf{ unless contain } \texttt{FAILURE} \textbf{ map } \Phi$$

4.3 Semantics of the `nfer` Formalism

The semantics of the core form 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. We first define the semantics of labeling rules with the *interpretation* function $R_{\wp}$, with the following type and definition. This function is parameterized with a *selection function* $\wp : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$, which will be explained below. Briefly stated: the Function $\wp$ is used, for example, to express the minimality constraint mentioned

earlier, which helps to reduce the complexity of the algorithm introduced in Section 5. Let Δ be the type of rules. Semantic functions are defined using the brackets $[\![\ \_\ ]\!]$ around syntax being given semantics.

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

The above definition, which defines the semantics of inclusive rules, reads as follows: Given an inclusive rule $\delta \in \Delta$ and a pool π, $R_{\wp}[\![\delta]\!]\,\pi$ first produces a pool π' containing intervals (η,s,e,M), where there exist two intervals in π, with names η_1 and η_2, where the time constraint is satisfied, and the map resulting from applying Φ to the respective sub-maps is not $\bot$. Subsequently the *selection function* $\wp$ selects from (potentially modifies) π', informed by π as well. The selection function is said to be *idempotent* iff $\wp(\pi',\pi) = \pi'$, and a *refinement* iff $\wp(\pi',\pi) \subseteq \pi'$. The following definition gives semantics to exclusive rules:

$$
\begin{array}{l}
R_{\wp}[\![\eta \leftarrow \eta_1 \ \textbf{unless} \ \ominus \ \eta_2 \ \textbf{map} \ \Phi \ ]\!] \ \pi \ = \\
\quad \textbf{let} \ \ \pi' \ = \\
\quad\quad \{ \ (\eta,s_1,e_1,J) \ \in \ \mathbb{I} \ \mid \\
\quad\quad\quad\quad (\eta_1,s_1,e_1,J) \ \in \ \pi \ \wedge \\
\quad\quad\quad\quad \neg \ (\exists \ s_2,e_2 \ \in \ \mathbb{C} \ \bullet \ \exists \ K \in \ \mathbb{M} \ \bullet \\
\quad\quad\quad\quad\quad\quad e_2 \ < e_1 \ \wedge \ (\eta_2,s_2,e_2,K) \in \ \pi \ \wedge \\
\quad\quad\quad\quad\quad\quad \ominus(s_1,e_1,s_2,e_2) \ \wedge \ \Phi(J,K) \neq \ \bot) \ \} \\
\quad \textbf{in} \ \ \wp(\pi',\pi)
\end{array}
$$

The above definition, which defines the semantics of exclusive rules, reads as follows: Given an exclusive rule $\delta \in \Delta$ and a pool π, $R_{\wp}[\![\delta]\!]\,\pi$ first produces a pool π' containing intervals (η,s_1,e_1,J), where there exists an interval with the name η_1 in π with the same time stamps and same data, and there does not exist a second "older" ($e_2 < e_1$) interval with the name η_2 in π, where the time constraint is satisfied, and the map resulting from applying Φ to the respective sub-maps is not $\bot$. As with inclusive rules, the selection function $\wp$ is then applied and may modify π'.

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:

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

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:

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

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 $S[\![\delta_1 \ldots \delta_n]\!] : \mathbb{P} \to \mathbb{P}$ 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.

4.4 Derived Forms

As hinted at the end of Section 4.2, a collection of $\oplus$ functions and $\ominus$ functions have been pre-defined, along with symbols (operators) denoting them. The symbols denoting $\oplus$ functions are shown in Table 1 together with their 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 are the earliest and latest time stamps of the involved intervals, respectively. 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.

Table 1 `nfer` $\oplus$ operators

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	$s_1 < e_2 \wedge s_2 < e_1 \wedge s = \max(s_1, s_2) \wedge e = \min(e_1, e_2)$

The informal explanation of the $\oplus$ 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: 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.

The symbols denoting $\ominus$ functions are shown in Table 2 together with their definitions. Note that s_1 and e_1 are the start and end times for the left-hand interval, and s_2 and e_2 are the start and end times for the right-hand interval. Unlike the $\oplus$ operators, the $\ominus$ operators do not affect the start and end times of the resulting interval. The informal explanation of

the $\ominus$ operators is as follows: A **unless after** B: A starts after B ends; A **unless follow** B: A starts where B ends; A **unless contain** B: all of B occurs during A. These operators are the *dual* of **before**, **meet**, and **during**, respectively.

Table 2 `nfer` $\ominus$ operators

Operator $\ominus$	$\ominus(s_1, e_1, s_2, e_2)$
after	$s_1 > e_2$
follow	$s_1 = e_2$
contain	$s_1 \leqslant s_2 \wedge e_2 \leqslant e_1$

The next abbreviation concerns further time constraints a user may want to impose. The core rule notation (see Section 4.2) 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 derived rule form:

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

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

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

We shall allow the time constraint (**within**) and/or map transformation (**map**) to be left out, in which case they assume the default function values respectively $\lambda s_1, e_1, s_2, e_2, s, e.\ \mathit{true}$ and $\lambda m_1, m_2.\ [\,]$.

So far rules can only be defined that refer to one operator and one additional clock predicate as shown above. This format presents a simple notation with a clean semantics. However, further convenient syntax allows rules containing more than one operator, for example: $A \ \leftarrow \ (B \ \textbf{before} \ C) \ \textbf{overlap} \ D$. Such rules are mapped into multiple rules in the core form (in this case two). The external DSL described in Section 6.1 allows such enriched rules.

4.5 Example

As an example, we will formalize the three rules that were informally stated in Section 3. The specification similarly consists of three rules:

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

`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 8) 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, assuming the selection function is idempotent:

1 : { (BOOT, 42, 160, [count ↦ 3]),
(BOOT, 255, 312, [count ↦ 4]),
(BOOT, 42, 312, [count ↦ 3]) }
2 : { (DBOOT, 42, 312, [count ↦ 4]) }
3 : { (RISK, 42, 312, [count ↦ 4]) }

Note that the third interval in step one, $(\mathsf{BOOT}, 42, 312, [\mathsf{count} \mapsto 3])$, is irrelevant, since it spans other BOOT intervals. In the next section, we will define the concept of *minimality* to restrict the generated intervals to only those in which we are interested.

5 Algorithm

The semantics given in Section 4 is expressed using an interpretation function $R_{\wp}[\![\delta]\!]\,\pi$ that operates on a finite pool, built from a finite, known trace. However, in online telemetry stream analysis, the stream of incoming events is, in theory, infinite, since we do not know when it ends. Therefore, constructing a pool from such a trace is not practical. To interpret an `nfer` specification with respect to a live telemetry stream (online), events in the stream must be converted to intervals and processed one at a time as they arrive. Algorithm 1 expresses a simple procedure for interpreting one interval at a time, either coming from the trace, or produced by the algorithm. The algorithm is defined as a function, calling itself recursively with newly produced intervals. An informal explanation of Algorithm 1 follows with a detailed example in Section 5.1.

Rules are assumed to be in a simplified binary form only referring to one temporal operator. Any specification can be rewritten into a semantically equivalent one consisting only of such binary rules, and it simplifies the algorithm. In a rule of the form $\eta \leftarrow \eta_1 \oplus \eta_2$ **map** Φ we refer to η as the *rule head*, and to $\eta_1 \oplus \eta_2$ **map** Φ as the *rule body*, or the *rule expression*. In such a rule body, we refer η_1 as the left-hand label, and η_2 as the right-hand label. We refer to the head, body, and left and right labels of exclusive rules in the same way. For each rule, the algorithm keeps track of three sets of already produced intervals relevant for that rule: *rule.LeftCache* holds the intervals which have been produced and matched the left-hand label of the rule, *rule.RightCache* holds the intervals which have been produced and matched the right-hand label of the rule, and finally *rule.Produced* holds the intervals which have been produced by the rule itself. In addition, the algorithm uses a variable *Subscribers*, which maps interval names to those rules that subscribe to intervals with those names.

Each rule also has methods that behave according to the operators $\oplus$ and $\ominus$, and map function Φ. Method *rule.testInclusion* checks that the time operator $\oplus$ is true and that the map function Φ does not return $\perp$. Method *rule.testExclusion* checks that the time operator $\ominus$ is true and that the map function Φ does not return $\perp$. Finally, method *rule.createInterval* generates a new interval using the time operator and map function.

Algorithm 1 Basic `nfer` Processing Algorithm

```
 1: procedure PROCESS(interval)
 2:     for rule ∈ Subscribers[interval.name] do
 3:         New ← ∅
 4:         if interval.name = rule.leftLabel then
 5:             if rule is exclusive then
 6:                 exclude ← false
 7:                 for rightIntv ∈ rule.RightCache do
 8:                     exclude ← exclude ∨ rule.testExclusion(interval, rightIntv)
 9:                 if ¬ exclude then
10:                     New ← New ∪ {rule.createInterval(interval)}
11:             else
12:                 for rightIntv ∈ rule.RightCache do
13:                     if rule.testInclusion(interval, rightIntv) then
14:                         New ← New ∪ {rule.createInterval(interval, rightIntv)}
15:         if interval.name = rule.rightLabel ∧ rule is inclusive then
16:             for leftIntv ∈ rule.LeftCache do
17:                 if rule.testInclusion(leftIntv, interval) then
18:                     New ← New ∪ {rule.createInterval(leftIntv, interval)}
19:         if interval.name = rule.leftLabel then
20:             rule.LeftCache ← rule.LeftCache ∪ {interval}
21:         if interval.name = rule.rightLabel then
22:             rule.RightCache ← rule.RightCache ∪ {interval}
23:         for new ∈ select(New, rule.Produced) do
24:             rule.Produced ← rule.Produced ∪ {new}
25:             process(new)
```

An informal explanation of Algorithm 1 follows. On Line 2, the procedure accesses the map *Subscribers* that associates interval names with rules. The rules that subscribe to the submitted interval's name are then iterated over.

Between lines 4 and 14, the algorithm handles the case where the passed interval name matches the left-hand label of the rule. If the rule is an exclusive rule (see Section 4.2.2), then *rule.RightCache* is iterated over looking for any intervals for which *rule.testExclusion* is true. If no such interval is found, then a new interval is generated and added to the set *New*. If the rule is an inclusive rule (see Section 4.2.1), then *rule.RightCache* is iterated over looking for any intervals for which *rule.testInclusion* is true. If such an interval is found, then a new interval is generated and added to the set *New*.

Between lines 15 and 18, the algorithm handles the case where the submitted interval name matches the right-hand label of the rule, and the rule is inclusive. In such a case, *rule.LeftCache* is iterated over looking for any intervals for which *rule.testInclusion* is true. If such an interval is found, then a new interval is generated and added to the set *New*. No work is required if the rule is exclusive, other than adding the interval to *rule.RightCache*.

Between lines 19 and 22, the algorithm adds the submitted interval to either or both caches, depending on which labels the interval name matches. It is necessary to wait to add the interval to the caches until after all of the condition tests are performed so that the interval cannot be matched against itself.

On Line 23 the selection function ($\wp$ in the semantics in Section 4.3) is applied and its results are iterated over. Each interval in the selected set is added to the *rule.Produced* set, and the PROCESS function is called on the interval recursively.

5.1 Example

This section presents an example illustrating an execution of Algorithm 1. Assume the following rule: `BOOT` ← `BOOT_S` **before** `BOOT_E`. We will trace the processing of two intervals: (`BOOT_S`, 10, 10, [ ]) and (`BOOT_E`, 20, 20, [ ]).

First, PROCESS((`BOOT_S`, 10, 10, [ ])) is called. The above rule is found to be a subscriber to this interval on Line 2 because its label (BOOT_S) is referenced in the rule's expression. The condition on Line 4 is true because BOOT_S is used on the left side of the **before** operator in the rule expression. The rule is inclusive, so the condition on line 5 is false. Since the condition was false, execution continues on Line 12 by iterating over the rule's *RightCache*, which is empty. The condition on Line 15 is false, since the interval's name (BOOT_S) does not appear on the right side of the **before** operator. Execution continues on Line 19, where the condition is met and so the interval is added to the rule's *LeftCache*. The condition on Line 21 is not met, so execution continues on Line 23. The *select* function is called on $(\varnothing, \varnothing)$, and the results are iterated over. If the select function is a *refinement* (see Section 4.3), then the returned set will also be empty, and the procedure returns.

Next, PROCESS((`BOOT_E`, 20, 20, [ ])) is called. The same rule is found to be a subscriber because BOOT_E is referenced in the rule's expression. Since BOOT_E appears on the right side of the **before** operator, the condition on Line 4 is false, but the condition on Line 15 is true and execution continues on Line 16. The rule's *LeftCache* contains the BOOT_S interval from above, so *rule.testInclusion* is called on the two intervals. The *testInclusion* method returns true, since the conditions of the **before** operator are met and there is no map function, so a new interval is created and added to the *New* set. The condition on Line 19 is false, but the condition on Line 21 is true, so the interval is added to the rule's *RightCache*. Next, the *select* function is called on two arguments: *New*, which contains the created interval; and the set of the rule's previously produced intervals, which is empty. If the *select* function returns a set containing the created interval, then that interval is added to the *Produced* set and then PROCESS is called on it recursively.

5.2 Complexity

In this section, we analyze the asymptotic complexity of Algorithm 1 and find that it is $O(n^3)$ in the length of the trace. We assume that the methods associated with a rule (*testInclusion*, *testExclusion*, *createInterval*) are constant time expressions. Given the number r of rules in a specification, and the number n of intervals in a trace, we can find the complexity as follows, with n_x referring to n's value ($n_x = n$, where x differentiates its use in the algorithm).

$$n_{trace} \times r \times \mathbf{max}\left(\underbrace{n_{\text{right}} + s + (1 \times n_{\text{recurse}})}_{\text{exclusive}}, \underbrace{n_{\text{right}} + n_{\text{left}} + s + ((n_{\text{left}} + n_{\text{right}}) \times 2n_{\text{recurse}})}_{\text{inclusive}}\right)$$

For each interval in the trace (n_{trace}) (Line 1), for each rule (r) (Line 2), we calculate the maximum of the two cases of: an exclusive rule versus an inclusive rule. Assume first that the rule is *exclusive*. If the submitted interval name matches the left-hand label of the rule, check all the intervals in the right cache (n_{right}), and generate at most one interval (Line 7). Call the selection function (s) (Line 23), and for each interval returned by the selection function, call PROCESS recursively (Line 25). If the selection function is a refinement, then

the number of iterations in the loop on Line 23 is bounded by the cardinality of the set *New*, which in this case is 1 (if the selection function is not a refinement, then the complexity of Algorithm 1 is unbounded). On the other hand, if the submitted interval name matches the right-hand label of the rule the interval is just stored (not included in the complexity calculation).

Assume next that the rule is *inclusive*. If the submitted interval name matches the left-hand label of the rule, for all the intervals in the right cache (n_{right}), generate an interval (Line 12). Symmetrically, if the submitted interval name matches the right-hand label of the rule, for all the intervals in the left cache (n_{left}), generate an interval (Line 16). Subsequently the selection function is called (s). The cardinality of *New* is at most $n+n$ since an interval may be created for each pairing of the newly submitted interval with all the intervals in both the left and right cache. For each interval returned by the selection function, call PROCESS recursively (Line 25).

The complexity of the algorithm is in part determined by the complexity of the selection function and the cardinality of the set it returns. In the case of an idempotent selection function, its complexity should be constant and the cardinality of the returned set should be at most $2n$. As long as the complexity of the selection function is not super-linear, the complexity of the processing function is $O(n^3)$ in the length of the trace, which appears impractical.

5.3 Minimality

This complexity is largely related to the number of new intervals returned by the selection function. Limiting the size of this set is therefore desirable as long as it is consistent with pragmatic needs. In practice, it is typically not desirable to use an idempotent selection function and return every matching interval. This is similar to how practical implementations of regular expressions use greedy matching instead of complete matching [41]. In the example in Section 4.5, three `BOOT` intervals are generated, but only two of them are relevant. The interval that begins at time 42 and ends at time 312 does not represent an intended abstraction, i.e. a period when the spacecraft is continuously booting. Instead, it contains two smaller `BOOT` intervals with a gap between where there is no boot taking place.

We observe that the property that differentiates relevant intervals from others is their *minimality*. An interval is defined as minimal if no other interval with the same label occurs **during** it. That is, given a pool $\pi \in \mathbb{P}$ and an interval (η, s, e, M),

$$\text{minimal}(\eta,s,e,M)(\pi) \leftarrow \nexists\ (\eta',s',e',M') \in \pi\ \bullet\ \eta = \eta' \wedge\ s \leq s' \wedge\ e' \leq e$$

The following selection function implements the minimality constraint. The function is a refinement and has a complexity determined by the cardinality of the *New* and *Prior* sets. In Section 8 we explore methods to limit the size of these sets.

Minimal Selection Function

1: **procedure** SELECT(*New, Prior*)
2: $\{i \in New \mid \nexists\ j \in Prior \cup New - \{i\} \mid i.name = j.name \wedge i.start \leq j.start \wedge j.end \leq i.end\}$

6 Implementation

The `nfer` logic has been implemented as a shallow, internal DSL (iDSL - essentially a library where functions on data are coded as functions in the implementation language), as well as an external DSL (eDSL - a stand-alone domain-specific language) in both Scala and C. The Scala iDSL was described in detail in the conference version [40] of this article. This section provides an overview of the eDSL as well as its implementation in both Scala and C. The C implementation is available to the public under the terms of the Gnu Public License version 3 (GPLv3) at `http://nfer.io`. The C iDSL is, furthermore, used to support an R language [52] interface.

Each kind of DSL (internal and external) has advantages and disadvantages. The advantages of the iDSL are ease of implementation and modification, ease of use in already existing programming environments (like IDEs), as well as a maximally expressive formalism for writing arbitrary data processing functions to be called in specifications. The disadvantages include the requirement that the user must be a programmer in the host language (Scala or C), generally a somewhat poorer syntax compared to an external DSL, and difficulty in analyzing specifications (in the case of a *shallow* internal DSL, where the host language forms a fundamental part of the DSL). Due to these negative iDSL properties, we chose to implement an eDSL. The choice of iDSL versus eDSL in a practical situation depends on the value given to the advantages and disadvantages mentioned just above.

In addition to these textual languages, we have experimented with visual entering of rules. A prototype GUI has been designed and implemented[6] for visual entering of rules based on an initial visualization of a trace. That is, the user is in this system presented a linear visualization of a trace, and can interact with this by selecting events of importance (point-and-click), thereby informing the system of the event patterns forming the body of a rule. The rationale behind this approach is the acknowledgement that it can be difficult for users to write rules without some guidance based on the format of actual traces. This prototype forms a basis for future work, and is not elaborated on further in this paper.

This section first introduces the eDSL syntax, then gives an overview of the actor-based Scala implementation, and finally describes the C implementation.

6.1 The External DSL

This section introduces the external DSL (eDSL) 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 external DSL:

```
BOOT :- BOOT_S before BOOT_E map {count → BOOT_S.count}

DBOOT :- b1:BOOT before b2:BOOT
  where b2.end − b1.begin ≤ 300 map {count → b1.count}

RISK :- DOWNLINK during DBOOT map {count → DBOOT.count}
```

The syntax should be self explanatory except for a few details. The first rule creates a BOOT interval containing a data map, which maps count to the count value of the BOOT_S

[6] The GUI was designed and implemented by Nathaniel Guy (JPL).

interval. This illustrates how data of particular intervals can be referenced. In cases where a rule body contains more than one occurrence of the same interval, as BOOT in the second rule, these can be labelled (here b1 and b2) to enable reference to their data and begin and end time points. The second rule illustrates a **where**-clause expressing a constraint on time values[7]. Such constraints can also refer to data. An expression language covering Boolean expressions involving arithmetic operations and comparisons (integers and real numbers) as well as string comparisons is built in.

⟨*spec*⟩ ::= ⟨*rule*⟩* | ⟨*module*⟩*

⟨*module*⟩ ::= **module** ⟨*id*⟩ '{' [**import** ⟨*id*⟩*, ';'] ⟨*rule*⟩* '}'

⟨*rule*⟩ ::= ⟨*id*⟩ ':-' ⟨*interval*⟩ [⟨*whereExp*⟩] [⟨*mapExp*⟩] [⟨*endPoints*⟩]

⟨*interval*⟩ ::= ⟨*intervalPrim*⟩ (⟨*op*⟩ ⟨*intervalPrim*⟩)*

⟨*intervalPrim*⟩ ::= [⟨*id*⟩ ':'] ⟨*id*⟩ | '(' ⟨*interval*⟩ ')'

⟨*whereExp*⟩ ::= **where** ⟨*exp*⟩

⟨*mapExp*⟩ ::= **map** '{' (⟨*id*⟩ '→' ⟨*exp*⟩)*, '}'

⟨*endPoints*⟩ ::= **begin** ⟨*exp*⟩ **end** ⟨*exp*⟩

⟨*op*⟩ ::= **also** | **before** | **meet** | **during** | **coincide** | **start** | **finish** | **overlap** | **slice** | ⟨*exclude*⟩

⟨*exclude*⟩ ::= **unless** (**after** | **follow** | **contain**)

⟨*exp*⟩ ::= ... | ⟨*id*⟩ '.' (⟨*id*⟩ | **begin** | **end**) | ⟨*id*⟩ '(' ⟨*exp*⟩+, ')' | ...

Fig. 2 Grammar for external `nfer` DSL

A grammar for the eDSL is shown in Figure 2. A specification is either a list of rules or a list of modules. Modules are useful for conceptually grouping a large number of rules. A module can import other modules and contains rules. The last occurring is the main module. A rule body is defined by an interval expression (interval), and three optional items: a where-constraint, a map, and a definition of the end points *begin* and *end*, in case the default generated time points are not desired. An interval (expression) is a composition of primary intervals separated by the temporal operators. A primary interval consist of an optional label, and an interval name. Alternatively a primary interval can be an interval in parentheses. Value expressions are standard and only specified partially here, focusing on the syntax for referring to data fields, and begin and end times of intervals; as well as function calls. Function calls are calls to user-defined functions in a programming language, registered to the monitor. This allows a user to call a function in, for example, Python or Scala, achieving the full expressiveness of a real programming language. Among the temporal operators is one that has not mentioned before: **also**, representing the lack of any constraints at all. This operator allows for time constraints defined purely using the **where** clause.

[7] Note that the eDSL uses **begin** to denote the start time of an interval, in contrast to the notation in Section 4 where it was referred to as *start* (time).

6.2 Scala Implementation

The Scala implementation is based on Akka actors communicating via asynchronous message passing through a publish/subscribe model built with Apache Kafka [42]. Each rule in an `nfer` specification results in an actor, which subscribes to intervals referenced by name in the body of the rule, and publishes the interval mentioned in the rule head (head, in head :− body) to the shared bus. This means that rule actors are only passed intervals which are pertinent to their execution. Figure 3 shows the `nfer` implementation's internal configuration corresponding to the double boot example in Section 6.1. 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.

For example, the RISK actor subscribes to both `DBOOT` and `DOWNLINK` intervals, 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` formalism 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.

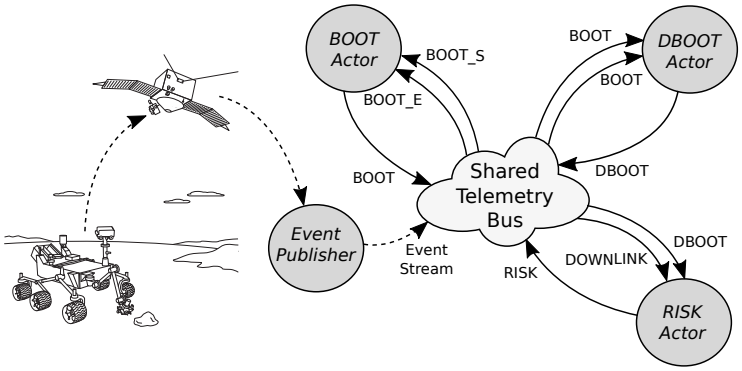

Fig. 3 Actor network corresponding to double boot example in Section 6.1

The implementation can process events *online*, as they come down to the ground from the spacecraft, or it can process a log of events stored on a file system. When processing logs, ground operators are usually only interested in recent events. However, there can be a need to analyze the telemetry stream from earlier points in time stored as multiple logs, e.g. from the start of the mission. In this case, 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 from older logs, which can then be stored for later use as an abstraction of those logs. In other words: stored intervals produced by `nfer` represent abstractions of the past.

6.2.1 Internal Representation of Rules in Scala

As already mentioned, each rule is implemented as an actor receiving and publishing events to the event bus. The rule inside an actor is represented by a tree structure closely corresponding to the abstract syntax tree obtained by parsing the body of the rule. Each node in the tree corresponds to a temporal operator, or a leaf node representing intervals to which the actor subscribes. During execution, a node contains the intervals that have thus far matched the corresponding sub-expression. To illustrate this tree structure, consider a slightly different formulation of the last two rules DBOOT and RISK above, merging them into one:

$$\begin{aligned}&\mathsf{RISK} :- \mathsf{DOWNLINK}\ \textbf{during}\ (\mathsf{b1{:}BOOT}\ \textbf{before}\ \mathsf{b2{:}BOOT})\\ &\quad \textbf{where}\ \mathsf{b2.end} - \mathsf{b1.begin} \leq 300\ \textbf{map}\ \{\mathsf{count} \rightarrow \mathsf{b1.count}\}\end{aligned}$$

The body of this rule contains two temporal operators. This rule is represented as the tree shown in Figure 4. Each node lists (in the top part of the box) a node number, a (possibly auto-generated) name, and an indication of which temporal operator it represents, including "Atomic" for the leaf nodes. The tree is shown after the following three intervals have been submitted: $(\mathsf{BOOT\_S}, 100, 100, [\mathsf{count} \mapsto 1])$, $(\mathsf{BOOT\_E}, 200, 200, [\,])$, and $(\mathsf{DOWNLINK}, 300, 300, [\,])$. As can be seen, some of the nodes contain intervals, and others do not yet. This reflects the step-wise evaluation of the rule as intervals arrive. An interval is submitted to the appropriate atomic leaf nodes of the tree, and then ascends the tree. It merges with other intervals according to the temporal operators, until the top node is reached and an interval is generated and published on the bus if the constraint is satisfied.

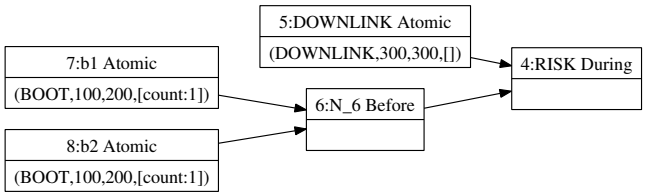

Fig. 4 Tree structure representing rule RISK

An interesting observation is that this data structure resembles the Rete data structure [30] used in rule-based systems, and explained in detail in [24]. In [35] we implemented the Rete algorithm following [24], while augmenting it for runtime verification. For a mention of other rule-based systems, the reader is referred to Section 9. In rule systems, rules typically have the form $c_1, \ldots, c_n \rightarrow a$, representing an action a to be executed when conditions $c_1, \ldots, c_n$ are true. Conditions and actions refer to facts stored in the Rete network, similar to how intervals are stored in the rule trees. The Rete algorithm maintains a single directed acyclic graph of nodes containing produced facts. The single graph represents all the rules in the rule program, and allows rules to share parts of the graph, thus reducing the amount of evaluations needed. In `nfer`, each rule is represented by its own tree, not shared with other rules. Similar to the `nfer` algorithm, the Rete algorithm for each node handles data coming up from a child node (left or right) by traversing the "other" child-node for matches. Whereas the `nfer` tree structure consists of nodes all of the same kind, the Rete data structure involves four kinds of different nodes.

6.3 C Implementation

The C language implementation is single threaded and conforms closely to the algorithm in Section 5. It encodes all rules as relations between two intervals[8]. Each rule subscribes to a left and a right label. For example, the following rule subscribes to BOOT_S as its left label and BOOT_E as its right label because of their position relative to the **before** operator:

```
BOOT :− BOOT_S before BOOT_E map {count → BOOT_S.count}
```

The implementation keeps two linked lists of subscribers to every label, one for left labels and one for right labels. When an interval is received, both lists of subscribers are iterated over for that label. Separate left and right lists are necessary because each rule may subscribe to two different labels, so it may appear in two different linked lists.

Nested rules are handled by adding anonymous, internal intervals to which other rules subscribe. All rules must have a label for the intervals they create, so nested rules create anonymous intervals with generated labels. Uniqueness is guaranteed for these labels by using an augmented naming alphabet and they are omitted from the final output of the algorithm. The parent rule that relies on the results of a nested rule then subscribes to this generated label. For example, the nested rule in Section 6.2.1 becomes the following two rules in the C implementation:

```
$BOOTBOOT1 :− b1:BOOT before b2:BOOT
              where ( b2.end − b1.begin ≤ 300 )
              map { $count1 → ( b1.count ) }

RISK :− DOWNLINK during $BOOTBOOT1
              map { count → ( $BOOTBOOT1.$count1 ) }
```

To ensure the correct behavior in **where**, **map**, **begin**, and **end** expressions, data items must be renamed and passed between nested rules. The external DSL only allows such expressions to be specified at the highest level, but they may refer the intervals in a nested rule. In the example above, the RISK interval sets the map item "count" to be equal to the value of the "count" data item of the left BOOT interval in the nested rule. The value is copied to an intermediate map key, "$count1", of the generated interval $BOOTBOOT1 so it can be accessed in the RISK rule.

Similarly, nested rules must inherit parts of **where** expressions that apply to them. This is important due to the influence of selection functions on the result. In the example above, the **where** expression is applied to the nested rule (b1:BOOT **before** b2:BOOT) because it applies only to the intervals in that rule. If the nested rule does not contain the restriction (b2.**end** − b1.**begin** ≤ 300), then the wrong intervals may be selected. If a subexpression of a **where** restriction, with type $\mathbb{B}$, concerns only a nested rule, the subexpression will be applied to the nested rule and replaced in the original expression with a generated map value.

The C implementation includes some performance optimizations. Strings are interned in dictionaries and expressions are stored and processed using Reverse Polish Notation (RPN). It is meant as a reference implementation, but its execution time and memory requirements are low enough to be used in an embedded setting.

[8] The eDSL supports unary rules but we will avoid describing them here as they represent a special case.

7 Example Application to Warning Analysis

As noted earlier, the `nfer` tool has been applied to processing of 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 *anticipated* due to known idiosyncrasies of the system, and therefore can be ignored. Events (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 (are false positives). 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 when a thread servicing the radio is starved and generates the warning `TLM_TR_ERROR` which indicates missing telemetry. This happens because the thread is preempted by higher-priority threads that are processing one of two commands (either `MOB_PRM` or `ARM_PRM`) that generate reports of current mobility and robotic arm parameter values. 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
  where CMD_DISPATCH.cmd = CMD_COMPLETE.cmd
  map {cmd → CMD_DISPATCH.cmd}

okRace :− TLM_TR_ERROR during cmdExec
  where cmdExec.cmd = "MOB_PRM" | cmdExec.cmd = "ARM_PRM"
```

The next rule involves a timing consideration. In this case, an instrument 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 being expected. The **this** keyword serves as a label for the interval that is generated.

```
okCmdFail :− INST_PWR_ON before
             INST_CMD_FAIL before
             INST_RECOVER
  where this.end − this.begin ≤ 15
```

The last set of rules label a situation in which a warning about task starvation is expected whenever an activity (labeled vdp) which fetches data products from the cameras overlaps with an Earth communication activity (labeled comm, and identified by an id field in its

map). In this case, we use the slice operator to identify the interval of overlap between the `vdp` and `comm` intervals:

```
comm :− COMM_BEGIN before COMM_END
  map {id → COMM_BEGIN.id}

vdp :− VDP_START before VDP_STOP

okStarvation :− TASK_STARVATION during (vdp slice comm)
  map {id → comm.id}
```

8 Performance Considerations

We saw in Section 5 that the complexity of the basic `nfer` processing algorithm with an idempotent selection function is $O(n^3)$ with respect to the length of the trace. For many cases, this is too high to be practical. The LogFire and Prolog experiments referred to in Section 1 illustrate this. Introducing a selection function to only keep intervals which are *minimal* reduces the complexity considerably, according to experiments, pushing the algorithm into the realm of being practical. However, it can still be improved. In this section we look for modifications which can improve the performance of the basic algorithm, anticipating, however, that such improvements may suffer from lack of soundness and completeness compared to the basic algorithm. These modifications are shown as alterations to Algorithm 1. Three such modifications are given: one-use, most-recent, and rolling-window. Note that, although they are given as modifications to the algorithm, their relationship to the semantics of `nfer` may be understood as changes to the selection function. Each of these modifications can be understood as a method to reduce the worst-case cardinality of the data structures over which Algorithm 1 iterates, specifically limiting the size of the left and right caches.

8.1 Proposed Modifications

The **one-use** modification, shown as Algorithm 2 (changes are underlined), deletes input intervals when they are used to create new intervals. In this way, any interval may only be used to create one new interval per rule. The *New* variable is modified to hold triples $(i_{\text{left}}, i_{\text{right}}, i_{\text{new}})$ for each new candidate interval i_{new} included (before selection), where i_{left} and i_{right} are the intervals in the left and right caches, respectively, that i_{new} is derived from. This is needed to remove those intervals from the caches later if i_{new} is selected (lines 25 and 26). The one-use modification reduces the maximum sizes of the left and right caches. This also reduces the *amortized* worst-case cardinality of the *New* set to one. Note that, because the worst-case cardinality of the *Produced* set is still linear in the size of the trace, the complexity of the minimality selection function is also linear, so the worst-case complexity of the `nfer` processing function is not changed.

The **most-recent** modification, shown as Algorithm 3, only stores the most recent intervals instead of keeping a cache of all of the previously seen ones. This change reduces the maximal cardinality of all caches to 1, except for the *Produced* cache. A cache's single element can be selected with the *head* function. The variable *New* holds at most one interval in this solution. Because the *Produced* cache still has a worst-case cardinality of $n-1$

Algorithm 2 One-use Modification

```
 1: ···
 4: if interval.name = rule.leftLabel then
 5:     ···
 9:     if ¬ exclude then
10:         New ← New ∪ {(interval, ε, rule.createInterval(interval))}
11: else
12:     for rightIntv ∈ rule.RightCache do
13:         if rule.testInclusion(interval, rightIntv) then
14:             New ← New ∪ {(interval, rightIntv, rule.createInterval(interval, rightIntv))}
15: if interval.name = rule.rightLabel ∧ rule is inclusive then
16:     for leftIntv ∈ rule.LeftCache do
17:         if rule.testInclusion(leftIntv, interval) then
18:             New ← New ∪ {(leftIntv, interval, rule.createInterval(leftIntv, interval))}
19: ···
23: for (left, right, new) ∈ select(New, rule.Produced) do
24:     rule.Produced ← rule.Produced ∪ {new}
25:     rule.LeftCache ← rule.LeftCache \ {left}
26:     rule.RightCache ← rule.RightCache \ {right}
27:     process(new)
```

(suppose a rule A :− B **before** B, then an interval is created for each subsequent B after the first), the worst-case complexity of the `nfer` processing algorithm is not reduced.

Algorithm 3 Most-recent Modification

```
18: ···
19: if interval.name = rule.leftLabel then
20:     rule.LeftCache ← {interval}
21: if interval.name = rule.rightLabel then
22:     rule.RightCache ← {interval}
```

The **rolling-window** modification, shown as Algorithm 4, only considers intervals in a cache that occur within a time window. Intervals falling outside the window are deleted from the caches. The time window is calculated as a fixed offset from the end of the last submitted interval. The rolling-window modification does not change the maximum cardinality of the caches (if all events occur within the window, then nothing is deleted), so it does not change the complexity of the algorithm according to the formula. However, if the window size is carefully chosen, the heuristic can have a drastic effect in the execution time of the processing algorithm.

8.2 Experimental Setup

We conducted a series of screening experiments to explore possibilities for improving the execution time of the `nfer` processing algorithm. We do not intend for this to be a comprehensive evaluation for choosing one algorithm over another. We implemented each algorithm and used it to apply `nfer` specifications on three different test datasets. All algorithms were tested using the *minimality* selection function discussed in Section 5.3. Both the C and Scala

Algorithm 4 Rolling-window Modification

```
 6: ···
 7: for rightIntv ∈ rule.RightCache do
        if rightIntv.end < interval.end - WINDOW then
            rule.RightCache ← rule.RightCache \ {rightIntv}
        else
 8:          exclude ← exclude ∨ rule.testExclusion(interval, rightIntv)
 9: ···
12: for rightIntv ∈ rule.RightCache do
        if rightIntv.end < interval.end - WINDOW then
            rule.RightCache ← rule.RightCache \ {rightIntv}
        else
13:         if rule.testInclusion(interval, rightIntv) then
14:             New ← New ∪ {rule.createInterval(interval, rightIntv)}
15: ···
16: for leftIntv ∈ rule.LeftCache do
        if leftIntv.end < interval.end - WINDOW then
            rule.LeftCache ← rule.LeftCache \ {leftIntv}
        else
17:         if rule.testInclusion(leftIntv, interval) then
18:             New ← New ∪ {rule.createInterval(leftIntv, interval)}
19: ···
23: for new ∈ select(New, {p|p ∈ rule.Produced, p.end > interval.end - WINDOW} ) do
24:     rule.Produced ← rule.Produced ∪ {new}
25:     process(new)
```

implementations were used to run the experiments. This helped us to eliminate some implementation specific blocking factors that could affect the performance of an algorithm. The C implementation experiments were performed in the Linux 4.9.6 operating system on an Intel Core i5 running at 2.4 GHz with 16 GB of RAM. The Scala implementation experiments were performed in the Mac OS X 10.10.5 operating system on an Intel Core i7 running at 2.8 GHz with 16 GB of RAM. The datasets are described in the following paragraphs.

The Sequential Sense-Process-Send (SSPS) dataset was generated by a system mimicking an embedded data collection device. The device-under-test (DUT) was a first generation BeagleBone with a 720 MHz ARM Cortex-A8 running version 6.6.0 of the QNX real-time operating system [51]. Logs were collected using the QNX tracelogger utility. The tested dataset contained 12,766 relevant events covering a collection period of approximately 26 hours.

The System Call Logs with Natural Random Faults (LANL) dataset was generated by running a simulation of an automotive cruise-control application on a computer under high-energy neutron bombardment [49]. The DUT was a Xilinx ZC706 featuring a XC7Z045 [57] System-on-a-Chip (SoC) running version 6.6.0 of the QNX real-time operating system [51]. Faults in the dataset were generated by placing the SoC in the path of a high-energy neutron beam at the Los Alamos Neutron Science Center (LANSCE) facility at the Los Alamos National Laboratory (LANL) in New Mexico, USA. The tested dataset contained 50,000 relevant events covering a collection period of approximately 10 hours.

The Mars Science Laboratory (MSL) dataset was generated by checking the property okRace described in Section 7 on telemetry logs received from the Curiosity rover. We checked logs covering rover activities over around 60 days. For convenience, we first filtered the rover logs to include only relevant EVRs. These EVRs are generated on board the

rover when the software executes command sequences as part of daily activity plans that are uplinked to the rover from Earth. The test dataset contained 50,000 relevant EVRs.

8.3 Experimental Results

Table 3 shows the results from the experiments. The **Impl** column differentiates between the C and Scala implementation (note that comparisons in timing and memory use should not be made between the C and Scala implementations). The **Algorithm** column shows the version of the processing algorithm under consideration, where Basic means Algorithm 1, One-use means Algorithm 2, Most-recent means Algorithm 3, and Window means Algorithm 4. The Window algorithm was applied with different window sizes indicated in seconds. The **Data** column shows the dataset and specification used for the row. The **Ex Time** column shows the clock time in seconds used by the program to reach a fixed-point, and the **Memory** column shows the peak memory used during computation. The **Precision** and **Recall** columns show the precision and recall of the algorithm where the "true" output is the result of running the Original algorithm.

Precision is an indicator for soundness (wrt. the basic algorithm) and is defined as the fraction of created intervals that are correct, and recall is an indicator for completeness (wrt. the basic algorithm) and is defined as the fraction of all expected intervals that are generated. More precisely, given the set of intervals B created by running the basic Algorithm 1 on a dataset with a specification, and given the set of new intervals N created by running a different algorithm on the same dataset and specification, precision and recall are defined as:

$$\text{precision} = \frac{|B \cap N|}{|N|} \qquad \text{recall} = \frac{|B \cap N|}{|B|}$$

For example, Line 4 of Table 3 shows the results from running the C implementation of the rolling-window algorithm (with a window size of five seconds) on the SSPS dataset with its `nfer` specification. It took 0.01 seconds to complete and used 11 megabytes of memory at its peak. The number of intervals found by the basic algorithm for the dataset and specification was 19,360, the number of intervals found by the rolling-window modification (5 s) was 5,749, and the size of the intersection between the two results was 5,661. So, the precision was $5661/5749 = 0.985$ and the recall was $5661/19360 = 0.292$.

In case of the MSL dataset, the recall number is followed by a number in parentheses. This is the number of okRace intervals produced, which are the intervals in which we are ultimately interested for this dataset. Four such intervals should be produced as shown for the Basic algorithm. We observe that the Most-recent algorithm did not produce any of these, and that the Window algorithm with a window of size 30 seconds (the smallest shown) only produced two of these, whereas the remaining window sizes produced all four.

Table 3 shows evidence that the rolling-window modification has the most promise to improve performance while still finding most or all of the relevant intervals. For each dataset, a window size was found that enabled a precision and recall of exactly 1.0, while executing at least an order of magnitude faster than the original algorithm. This ideal examined window size was different for each dataset: it was 210 seconds for SSPS, 1,000 seconds for LANL, and 41,000 seconds for MSL. For each new application, the window size must be tuned to find this optimal number.

The one-use modification was interesting, in that it was able to return nearly the same results as the basic algorithm but it did not show as much of a performance improvement as the well-tuned rolling-window modification. For the LANL dataset, the results were *exactly*

Table 3 Results from experiments

Impl	Algorithm	Data	Ex Time	Memory	Precision	Recall
C	Basic	SSPS	1.58 s	16 MB	1.0	1.0
C	One-use	SSPS	0.37 s	13 MB	∼ 1.0	0.999
C	Most-recent	SSPS	0.14 s	9 MB	∼ 1.0	0.997
C	Window (5 s)	SSPS	0.01 s	11 MB	0.985	0.292
C	Window (15 s)	SSPS	0.01 s	13 MB	0.999	0.699
C	Window (30 s)	SSPS	0.02 s	15 MB	∼ 1.0	0.966
C	Window (60 s)	SSPS	0.02 s	15 MB	∼ 1.0	∼ 1.0
C	Window (210 s)	SSPS	0.03 s	15 MB	1.0	1.0
C	Window (1,000 s)	SSPS	0.06 s	15 MB	1.0	1.0
C	Basic	LANL	138.40 s	54 MB	1.0	1.0
C	One-use	LANL	26.93 s	42 MB	1.0	1.0
C	Most-recent	LANL	0.55 s	12 MB	1.0	0.613
C	Window (1 s)	LANL	0.18 s	49 MB	1.0	0.772
C	Window (5 s)	LANL	0.21 s	53 MB	1.0	0.986
C	Window (30 s)	LANL	0.30 s	52 MB	1.0	0.999
C	Window (500 s)	LANL	1.96 s	54 MB	1.0	∼ 1.0
C	Window (1,000 s)	LANL	3.63 s	53 MB	1.0	1.0
C	Window (10,000 s)	LANL	41.59 s	53 MB	1.0	1.0
Scala	Basic	MSL	251.1 s	80 MB	1.0	1.0 (4)
Scala	One-use	MSL	196.9 s	70 MB	∼ 1.0	0.916 (4)
Scala	Most-recent	MSL	0.4 s	1 MB	0.997	0.454 (0)
Scala	Window (30 s)	MSL	2.9 s	1 MB	∼ 1.0	0.550 (2)
Scala	Window (100 s)	MSL	27.3 s	10 MB	0.976	0.913 (4)
Scala	Window (2,000 s)	MSL	24.7 s	50 MB	0.992	0.972 (4)
Scala	Window (41,000 s)	MSL	28.5 s	40 MB	1.0	1.0 (4)
Scala	Window (500,000 s)	MSL	89.6 s	70 MB	1.0	1.0 (4)
Scala	Window (1,000,000 s)	MSL	166.0 s	80 MB	1.0	1.0 (4)

the same as the basic algorithm. For the SSPS dataset 25 out of 19,360 intervals were missing and three extra intervals were generated. For the MSL dataset around 9% of the expected results were missing and one extra interval was generated. Performance varied from about a 32% improvement for MSL to about a 80% improvement for LANL, but this is in contrast to the one or two order of magnitude improvements seen from the other methods.

The most-recent modification missed too many relevant intervals to be of much practical use, although it was found to execute quickly and had good precision. Its recall of only 0.613 for the LANL dataset and 0.454 for the MSL dataset show that it found too few of the expected results. The modification had nearly perfect results for the SSPS dataset, however, so it may be usable in some circumstances.

9 Related Work

An earlier effort to develop a telemetry comprehension tool is described in [36], which provided a Scala DSL for writing a subset of the specifications shown in this article. That work was based on a still earlier effort using the rule-based system LogFire [35] for analyzing telemetry streams, as described in [37]. Although rule systems are strongly related to `nfer`, they are not suited for expressing minimality constraints for optimization purposes, as discussed previously. Other rule systems include Drools [25], Clips [16], and Jess [39].

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 [46] and ANMLite [14]. 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 [56], 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, [55] 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. Other interval logics have been designed specifically for verification purposes, such as Interval Temporal Logic (ITL) [48], the Duration Calculus (DC) [34], and Graphical Interval Logic (GIL) [23].

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.

Runtime verification [44, 28] can generally be defined as the discipline of constructing monitors for analyzing systems executions. Assuming the type $\mathbb{E}$ of events, and some data domain D, a monitor can abstractly be considered as a function that takes a set of traces as an argument and returns a data value of type D. That is, M has the type $M : 2^{\mathbb{E}^*} \to D$. In most runtime verification systems a monitor processes a single trace. As such `nfer` can be seen as a runtime verification tool, processing a single trace and returning a set of intervals: $D = 2^{\mathbb{I}}$. Traditionally, however, runtime verification tools analyze traces in order to provide a Boolean verdict ($D = \mathbb{B}$), or some value in a simple extension of the Boolean domain [12], indicating whether a trace satisfies a specification or not (or the result can be unknown). Such systems include e.g. Eagle [7], MOP [47], Orchids [31], Ruler [8], TraceContract [6], LTL_3 [13], MarQ [54] (based on QEA - Quantified Event Automata [9]), LogFire [35], Larva [17], RiTHM [50], JUnitRV [21], MMT [22], MonPoly [11], and DejaVu [38]. Runtime verification systems have been developed which aggregate data as part of the verification [29, 10]. A system such as LOLA [20] computes general data streams from input data streams, and is in this sense more general than the Boolean verdict systems. Systems such as Ruler and LogFire produce sets of facts from traces, which is also more general than Boolean verdicts. Statistical model checking [43] is an approach collecting statistical information about the degree to which a specification is satisfied on multiple traces. In specification mining [27, 53] the user provides no specification. Instead it is learned from a collection of nominal executions.

The `nfer` system can be regarded as a form of Complex Event Processing (CEP) [45, 26]. Like `nfer`, the objective in CEP is to use rewrite rules to abstract higher level events from lower level events for human comprehension and/or further processing. The initial ob-

jective of CEP, however, was analysis and abstraction of distributed systems, whereas `nfer` has been created for analyzing a single event stream generated by a single processor. In spite of this difference, `nfer` shares with CEP the objective of rule-based event abstraction. Examples of CEP systems include BeepBeep [33, 32], STREAM [19, 5], and Gigascope [18].

10 Conclusion

We have introduced the `nfer` rule-based formalism and system for inferring event stream abstractions. The problem has been inspired by actual planetary space mission operations, specifically the Mars Curiosity rover. The result of applying an `nfer` specification to an event stream is a set of intervals: named sections of the event stream, each including a start time, an end time, and a map holding data. Intervals are formed from events and other intervals, forming a hierarchy of abstractions. The result may be visualized or queried, and can generally help engineers to better comprehend the contents of an event stream. The `nfer` system is implemented in both Scala and C, with an external DSL for expressing rules, in addition to internal Scala and C DSLs (APIs). The system has been shown to scale well, aided by simple algorithmic enhancements without much loss of precision. Future work includes handling missing and out-of-order telemetry; support for visual entering of rules and visualization of results; and mining specifications from past event logs.

References

1. Albarghouthi A, Baier JA, McIlraith SA (2009) On the use of planning technology for verification. In: Proc. of the ICAPS Workshop on Verification & Validation of Planning & Scheduling Systems (VVPS), Citeseer
2. Allen JF (1983) Maintaining knowledge about temporal intervals. Communications of the ACM 26(11):832–843
3. Allen JF (1984) Towards a general theory of action and time. Artificial intelligence 23(2):123–154
4. Alur R, Fisman D, Raghothaman M (2016) Regular programming for quantitative properties of data streams. In: Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Eindhoven, The Netherlands, Springer, LNCS, vol 9632, pp 15–40
5. Arasu A, Babu S, Widom J (2006) The CQL continuous query language: semantic foundations and query execution. The International Journal on Very Large Data Bases 15(2):121–142
6. Barringer H, Havelund K (2011) TraceContract: A Scala DSL for trace analysis. In: Proc. of the 17th International Symposium on Formal Methods (FM'11), Springer, LNCS, vol 6664, pp 57–72
7. Barringer H, Goldberg A, Havelund K, Sen K (2004) Rule-based runtime verification. In: Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI), vol 2937, pp 44–57
8. Barringer H, Rydeheard D, Havelund K (2008) Rule systems for run-time monitoring: from Eagle to RuleR. Journal of Logic and Computation 20(3):675–706
9. Barringer H, Falcone Y, Havelund K, Reger G, Rydeheard D (2012) Quantified event automata: Towards expressive and efficient runtime monitors. In: Proc. of the 18th Int.

Symposium on Formal Methods (FM'12), Springer, pp 68–84, DOI 10.1007/978-3-642-32759-9_9
10. Basin D, Harvan M, Klaedtke F, Zălinescu E (2011) MONPOLY: Monitoring usage-control policies. In: 2nd Int. Conference on Runtime Verification (RV'11), Springer, LNCS, vol 7186, pp 360–364
11. Basin D, Klaedtke F, Marinovic S, Zălinescu E (2015) Monitoring of temporal first-order properties with aggregations. Formal Methods in System Design URL http://link.springer.com/article/10.1007/s10703-015-0222-7
12. Bauer A, Leucker M, Schallhart C (2007) The good, the bad, and the ugly, but how ugly is ugly? In: 7th Int. Workshop on Runtime Verification (RV'07), Springer, LNCS, vol 4839, pp 126–138
13. Bauer A, Leucker M, Schallhart C (2011) Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology (TOSEM) 20(4):14
14. Butler RW, Siminiceanu RI, Muno C (2007) The ANMLite language and logic for specifying planning problems. Report 215088:23,681–2199
15. Chen F, Roşu G (2007) MOP: an efficient and generic runtime verification framework. In: ACM SIGPLAN Notices, ACM, vol 42, pp 569–588
16. CLIPS (2017) Website. http://clipsrules.sourceforge.net, accessed: 2017-03-21
17. Colombo C, Pace GJ, Schneider G (2009) Larva — safer monitoring of real-time java programs (tool paper). In: Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods, IEEE Computer Society, Washington, DC, USA, SEFM '09, pp 33–37, DOI 10.1109/SEFM.2009.13
18. Cranor C, Johnson T, Spataschek O, Shkapenyuk V (2003) Gigascope: a stream database for network applications. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, ACM, pp 647–651
19. Cugola G, Margara A (2012) Processing flows of information: From data stream to complex event processing. ACM Computing Surveys (CSUR) 44(3):15
20. D'Angelo B, Sankaranarayanan S, Sánchez C, Robinson W, Finkbeiner B, Sipma HB, Mehrotra S, Manna Z (2005) LOLA: Runtime monitoring of synchronous systems. In: Proc. of the 12th Int. Symposium on Temporal Representation and Reasoning, IEEE Computer Society, pp 166–174
21. Decker N, Leucker M, Thoma D (2013) jUnit^{RV}—adding runtime verification to jUnit. In: Brat G, Rungta N, Venet A (eds) NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings, Springer, Lecture Notes in Computer Science, vol 7871, pp 459–464, DOI 10.1007/978-3-642-38088-4_34
22. Decker N, Leucker M, Thoma D (2016) Monitoring modulo theories. Int J Software Tools for Technology Transfer 18(2):205–225, DOI 10.1007/s10009-015-0380-3
23. Dillon LK, Kutty G, Moser LE, Melliar-Smith PM, Ramakrishna YS (1994) A graphical interval logic for specifying concurrent systems. ACM Trans Softw Eng Methodology 3:131–165
24. Doorenbos RB (1995) Production matching for large learning systems. PhD thesis, Carnegie Mellon University, Pittsburgh, PA
25. Drools (2017) Website. http://www.jboss.org/drools, accessed: 2017-03-21
26. Eckert M, Bry F (2009) Complex event processing (CEP). Informatik-Spektrum 32(2):163–167
27. Ernst MD, Perkins JH, Guo PJ, McCamant S, Pacheco C, Tschantz MS, Xiao C (2007) The Daikon system for dynamic detection of likely invariants. Science of Computer Programming 69(1):35–45

28. Falcone Y, Havelund K, Reger G (2013) A tutorial on runtime verification. In: Engineering Dependable Software Systems, pp 141–175, DOI 10.3233/978-1-61499-207-3-141
29. Finkbeiner B, Sankaranarayanan S, Sipma H (2005) Collecting statistics over runtime executions. Formal Methods in System Design 27(3):253–274
30. Forgy C (1982) Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19:17–37
31. Goubault-Larrecq J, Olivain J (2008) A smell of ORCHIDS. In: Proc. of the 8th Int. Workshop on Runtime Verification (RV'08), Springer, LNCS, vol 5289, pp 1–20
32. Hallé S (2016) When RV Meets CEP. In: Runtime Verification: 16th International Conference, RV 2016, Madrid, Spain, September 23–30, 2016, Proceedings, Springer, pp 68–91
33. Hallé S, Gaboury S, Bouchard B (2016) Activity Recognition Through Complex Event Processing: First Findings. In: AAAI Workshop: Artificial Intelligence Applied to Assistive Technologies and Smart Environments
34. Hansen MR, Van Hung D (2007) A theory of duration calculus with application. In: Domain modeling and the duration calculus, LNCS, vol 4710, Springer, pp 119–176
35. Havelund K (2015) Rule-based runtime verification revisited. Int J Software Tools for Technology Transfer 17(2):143–170
36. Havelund K, Joshi R (2014) Comprehension of spacecraft telemetry using hierarchical specifications of behavior. In: Merz S, Pang J (eds) Formal Methods and Software Engineering: 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, November 3-5, 2014. Proceedings, Springer International Publishing, LNCS, vol 8829, pp 187–202
37. Havelund K, Joshi R (2015) Experience with rule-based analysis of spacecraft logs. In: Artho C, Ölveczky CP (eds) Formal Techniques for Safety-Critical Systems: Third International Workshop (FTSCS 2014), November 2014, Luxembourg, Springer International Publishing, Communications in Computer and Information Science, vol 476, pp 1–16
38. Havelund K, Peled D, Ulus D (2017) First order temporal logic monitoring with BDDs. In: 17th Conference on Formal Methods in Computer-Aided Design (FMCAD 2017), 2-6 October, Vienna, Austria, IEEE Computer Society, pp 116–123
39. Jess (2017) Website. http://www.jessrules.com/jess, accessed: 2017-03-21
40. Kauffman S, Havelund K, Joshi R (2016) nfer – a notation and system for inferring event stream abstractions. In: Falcone Y, Sánchez C (eds) Runtime Verification: 16th International Conference, RV 2016, Madrid, Spain, September 23–30, 2016, Proceedings, Springer, LNCS, vol 10012, pp 235–250
41. Kearns SM (1991) Extending regular expressions with context operators and parse extraction. Software: Practice and Experience 21(8):787–804, DOI 10.1002/spe.4380210803
42. Kreps J, Narkhede N, Rao J (2011) Kafka: A distributed messaging system for log processing. In: Proc. of the 6th Int. Workshop on Networking Meets Databases (NetDB'11), ACM, pp 1–7
43. Legay A, Delahaye B, Bensalem S (2010) Statistical model checking: An overview. In: 1st Int. Conference on Runtime Verification (RV'10), Springer, LNCS, vol 6418
44. Leucker M, Schallhart C (2009) A brief account of runtime verification. The Journal of Logic and Algebraic Programming 78(5):293–303, DOI 10.1016/j.jlap.2008.08.004
45. Luckham D (2002) The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley

46. Mcdermott D, Ghallab M, Howe A, Knoblock C, Ram A, Veloso M, Weld D, Wilkins D (1998) PDDL - The Planning Domain Definition Language. Tech. rep., CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control
47. Meredith P, Jin D, Griffith D, Chen F, Roşu G (2011) An overview of the MOP runtime verification framework. Int J Software Tools for Technology Transfer pp 1–41, DOI 10.1007/s10009-011-0198-6
48. Moszkowski BC (1985) A temporal logic for multilevel reasoning about hardware. IEEE Computer 18:10–19
49. Narayan A, Kauffman S, Morgan J, Tchamgoue GM, Joshi Y, Hobbs C, Fischmeister S (2017) System call logs with natural random faults: Experimental design and application. In: SELSE-13: The 13th Workshop on Silicon Errors in Logic, System Effects, Boston, MA, USA
50. Navabpour S, Joshi Y, Wu W, Berkovich S, Medhat R, Bonakdarpour B, Fischmeister S (2013) RiTHM: a tool for enabling time-triggered runtime verification for C programs. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, pp 603–606
51. QNX (1997) QNX Operating System: system architecture. QNX Software Systems Ltd.
52. R Development Core Team (2008) R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria, URL http://www.R-project.org, ISBN 3-900051-07-0
53. Reger G (2014) Automata based monitoring and mining of execution traces. PhD thesis, University of Manchester
54. Reger G, Cruz HC, Rydeheard D (2015) MarQ: monitoring at runtime with QEA. In: Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)
55. Rosu G, Bensalem S (2006) Allen linear (interval) temporal logic - translation to LTL and monitor synthesis. In: 18th Int. Conference on Computer Aided Verification (CAV'06), Springer, LNCS, vol 4144, pp 263–277
56. Siminiceanu R, Butler RW, Muñoz CA (2009) 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, Springer, pp 132–146
57. Xilinx (2017) Zynq-7000 all programmable soc zc706 evaluation kit. https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html, accessed: 2017-03-13