Towards a Logic for Inferring Properties of Event Streams*

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

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

Abstract. We outline the background, motivation, and requirements of an approach to create abstractions of event streams, which are time-tagged sequences of events generated by an executing software system. Our work is motivated by the need to process event streams with millions of events that are generated by a spacecraft, that must be processed quickly after they are received on the ground. Our approach involves building a tool that adds hierarchical labels to a received event stream. The labels add contextual information to the event stream, and thus make it easier to build tools for visualizing and analyzing telemetry. We describe a notation for writing hierarchical labeling rules; the notation is based on a modification of Allen Temporal Logic, augmented with rule-definitions and features for referring to data in data parameterized events. We illustrate our notation and its use with an example.

1 Introduction

The most broadly applied approach to ensure functional correctness of software systems is testing. That is, executing the software in a finite number of scenarios and verifying the correct behavior. Various techniques have been developed to improve the testing experience, including Runtime Verification (RV). RV is a method for verifying that a program execution satisfies a user-provided formal specification. Such specifications are typically expressed in some form of temporal logic, regular expressions, or state machines. Occasionally, but more rarely, they are expressed as rule systems and grammars. 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. Logics have, furthermore, been developed which aggregate data as part of the verification [4, 3, 2].

In this paper, we outline an approach to *software comprehension*. A user provides a specification that is used to annotate a given event stream with contextual information that makes it easier to build tools for visualizing and analyzing the trace. The proposed specification logic is a modification of Allen's Temporal

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

Logic (ATL) [1], well known from AI, which turns out to be suitable for expressing hierarchical specifications of spacecraft behavior. We have implemented our ideas in a system named `nfer` (a tool for "telemetry i**nfer**ence"). The design of this logic is driven by the challenges faced in operating spacecraft, where the only knowledge ground personnel have of the remote behavior is from telemetry sent down to Earth. The `nfer` system provides both a declarative notation that allows engineers to write hierarchical specifications of spacecraft behavior, and a tool that uses these specifications to automatically label a received telemetry stream. The labels are used both in visualizing telemetry in real-time as it is received, as well as for building tools that make it easier to query past telemetry. The tool is being applied for analyzing telemetry received from the Curiosity rover currently on Mars [6].

The work is a continuation and refinement of previous work described in [5]. Roşu and Bensalem [7] define a translation of a modified ATL to Linear Temporal Logic (LTL) for monitoring, realizing, however, that a specialized monitoring algorithm is more efficient. Our work differs in a number of respects: (i) instead of monitoring ATL relationships for verification, we generate a relationship hierarchy for program understanding, (ii) we handle data parameterized intervals, (iii) we allow any constraints on time and parameter values, not just the 13 ATL constraints, (iv) in their system, an interval is unique, while in `nfer` it can occur multiple times. Our work has strong similarities to data-flow (data streaming) languages. A very recent example is QRE [2], 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, in order 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.

The remainder of the paper is organized as follows. Section 2 outlines the background as well as requirements for this effort, including an example. Section 3 suggests a solution and uses it to formalize the provided example. Finally, Section 4 concludes the paper.

2 Requirements

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

2.1 Illustrating Example

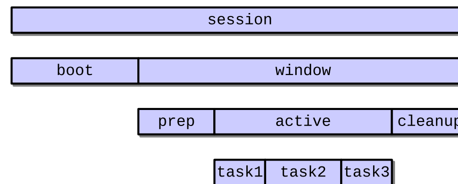
Consider the trace shown in Figure 1(a), that we assume has been generated by a spacecraft[3]. The trace consists of a sequence of events, or Event Verification Records (EVRs), each with a name, and list of arguments, including a time stamp. This sequence of 15 events is already too long for human comprehension, even if we provide the following informal description of how to read the trace:

- A session interval consists of a *boot* interval followed by a *window* interval.
- A *boot* interval starts with a `VERSION` event, ends with a `DEACTIVATE` event, and must contain a `BOOT_COUNT` event.
- A *window* interval starts with a *prep* interval, followed by an *active* interval, followed by a *cleanup* interval, and must contain an `ACTIVATE_SEQ` event.
- A *prep* interval starts with a `WINDOW_PREP` event and ends with a `DUR1` event.
- An *active* interval starts with a *task1* interval, followed by a *task2* interval, followed by a *task3* interval.
- A *task1* interval starts with a `DUR1` event and ends with a `DUR2` event.
- A *task2* interval starts with a `DUR2` event and ends with a `DUR3` event.
- A *task3* interval starts with a `DUR3` event and ends with a `FINISHED` event.
- A *cleanup* interval starts with a `FINISHED` event and ends with a `CLEANUP` event.

Our objective is to formalize the above information in a specification, match the specification against the trace, and convey the actual matches in a visually appealing manner. We are not interested in whether the trace satisfies the above information exactly, but rather to what extent it matches. The result could for example be the visualization shown in Figure 1(b). As can be seen, the visualization clearly shows how a *session* consists of a *boot* and a *window*, which itself consist of a *prep*, *active* and *cleanup*, and where an *active* consists of the three tasks executed in sequence.

```
SLEEP(07:12:02)
VERSION(09:23:10,10.2.1)
BOOT_COUNT(09:23:16,12)
REPORT(09:23:18)
DEACTIVATE(09:23:30)
WINDOW_PREP(09:29:59,782,25,2)
ACTIVATE_SEQ(09:59:12,2)
OK(10:04:59)
DUR1(10:05:05)
RESET(10:05:06)
DUR2(10:05:21)
DUR3(10:07:03)
STORING(10:16:48)
FINISHED(10:17:04)
CLEANUP(10:20:05)
```

(a) A trace of events



(b) Visualization of the trace

Fig. 1. An event trace and its visualization

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

2.2 Desired Features

The specification language should allow a user to:

1. label event relations in the trace, for example to define the label *task1* to represent an interval delimited by the events `DUR1` and `DUR2`.
2. define higher-level labels as a composition of lower-level labels. For example, a *session* is composed of a *boot* and a *window* in sequence.
3. refer to time stamps associated to events in the trace, as well as generate and read start and end times of generated labels.
4. refer to other data associated with events, as well as generate and read data of generated labels using arbitrary expressions. For example, a label can have a datum value defined as the sum of two lower-level event data.
5. specify other relationships than one event/labeling occurs before another. For example it should be possible to specify that one label contains another, that two labels overlap, etc.

3 Outline of a Logic

Our logic is inspired by ATL [1], specifically its operators for expressing temporal constraints on time intervals. In ATL, a time interval represents an action taking place over a time period (e.g. "Drive"), or a system state over a time period (e.g. "Overheated"). A time interval has a name, a start time, and an end time.

ATL offers 13 mutually exclusive binary relations. Examples are: $\mathit{Before}(i,j)$ which holds iff interval i ends before interval j starts, and $\mathit{During}(i,j)$ which holds iff i starts strictly after j starts and ends before or when j ends (or vice versa). An ATL formula is a conjunction[4] of such relationships, for example, $\mathit{Before}(A,B) \ \wedge \ \mathit{Contains}(B,C)$. A model is a set of intervals satisfying such a conjunction of constraints. ATL is typically used for generating a model (plan) from a formula (planning), but can also be used for checking a model against a formula, as described in [7].

Our objective is different from planning and verification. Given a trace, we want to generate a model (a set of intervals), guided by a specification that we provide, that represents a layered view of the trace. Let an interval be defined as a 4-tuple (η, t_1, t_2, m) consisting of a name η, and a start time t_1, an end time t_2, and a map $m : Id \rightarrow V$ from identifiers to values, the arguments of the interval. The input to our system is a trace σ: a sequence of named events of the form $\eta(t,m)$ consisting of a name η, a unique time stamp t, and a map m (the arguments to the event). The trace is converted into an initial model, which is the set $\{(\eta,t,t,m) \mid N(t,m) \in \sigma\}$. The specification defining the transformation of this initial model is a set of rules of the form:

$$\eta \doteq \eta_1(m_1) \ \oplus \ \eta_2(m_2) \ \textbf{if} \ C \ \textbf{map} \ M$$

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

Operator $\oplus$	Name	Explanation
$A ; B$	*A before B*	A ends before B starts
$A : B$	*A meet B*	A ends where B starts
$A \sqsubseteq B$	*A during B*	all of A occurs during B
$A = B$	*A coincide B*	A and B occur at the exact same time
$A \vdash B$	*A start B*	A starts at the same time as B
$A \dashv B$	*A finish B*	A finishes at the same time as B
$A + B$	*A join B*	an A and a B with no constraint
$A \mid B$	*A overlap B*	A and B overlap in time
$A \sqcap B$	*A slice B*	A and B overlap in time and only overlap is returned

Table 1. `nfer` operators

The rule states that: if there are two intervals named η_1 respectively η_2 already generated, with maps specified by m_1 and m_2 respectively, that are related time-wise with the temporal operator $\oplus$, and if the condition C holds on the maps of the respective intervals (true if left out in abbreviated form)[5], then an interval named η is generated, with the map described by the map expression M (the empty map if left out in abbreviated form). The operators are those presented informally in Table 1, which are inspired by ATL, although not identical, since our needs are slightly different. Each operator on two intervals A and B returns an interval that time wise spans both intervals in their entirety (the maximal view), except for the last *slice* operator $A \sqcap B$, which returns only the interval (slice) which A and B have in common (the minimal view).

As convenient syntax we allow expressions containing several operators on the right hand side of a rule, but such derived rules map to the simple form above. The specification of our trace abstraction outlined in Section 2 is shown in Figure 2 (with a condition and map functions added for illustration). A term such as $BOOT_COUNT(2 : count)$ means matching a $BOOT_COUNT$ event where the second map argument is bound to the free variable $count$, and the expression $m \dagger \{seq : x\}$ is the map m overridden by seq being mapped to x.

4 Conclusion

We have introduced the problem of inferring a model from an event stream, guided by a formal specification, for the purpose of system comprehension. We have outlined a rule-based logic, `nfer`, influenced by Allen Temporal Logic (ATL), for writing specifications. ATL itself is an attractive logic due to its simplicity, as well as naturalness for visualization, and is normally used for planning purposes. `nfer` adds rule-definitions as well as data parameterization to a variant of this logical system. A prototype of `nfer` has been implemented in Scala as

[5] In the fully generic form the user can define his/her own operators as arbitrary predicates on time stamps.

```
session ← boot ; window .

boot ← VERSION ; BOOT_COUNT(2 : count) ; DEACTIVATE
  if count > 10 map {boot_count : count} .

window ← ACTIVATE_SEQ(2 : x) ⊑(prep(m) ; active ; cleanup)
  map m † {seq : x} .

prep ← WINDOW_PREP(3 :wi, 4 : ty) ; DUR1
  map {wid : wi, type : ty} .

active  ← task1; task2; task3.
task1← DUR1; DUR2.
task2← DUR2; DUR3.
task3← DUR3; FINISHED .
cleanup ← FINISHED ; CLEANUP .
```

Fig. 2. Example specification

an internal DSL (API), and is built on a publish and subscribe framework, for processing telemetry data from the Mars Curiosity rover. Future work includes refining the implementation, including optimizing time and space; improving the internal rule DSL; creating an external DSL; and allowing rules to be written in other languages, such as Python, commonly used by flight mission engineers.

References

1. Allen, J.F.: Maintaining knowledge about temporal intervals. Communications of the ACM 26(11), 832–843 (1983)
2. Alur, R., Fisman, D., Raghothaman, M.: Regular programming for quantitative properties of data streams. In: Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Eindhoven, The Netherlands. LNCS, vol. 9632, pp. 15–40. Springer (April 2016)
3. Basin, D., Harvan, M., Klaedtke, F., Zălinescu, E.: Monpoly: Monitoring usage-control policies. In: Runtime Verification. pp. 360–364. Springer (2011)
4. Finkbeiner, B., Manna, Z., Sipma, H.B.: Deductive verification of modular systems. In: Compositionality: The Significant Difference, pp. 239–275. Springer (1998)
5. Havelund, K., Joshi, R.: Comprehension of spacecraft telemetry using hierarchical specifications of behavior. In: 16th International Conference on Formal Engineering Methods (ICFEM), Luxembourg. LNCS, vol. 8829, pp. 187–202. Springer (Nov 2014)
6. Mars Science Laboratory (MSL) mission website: http://mars.jpl.nasa.gov/msl.
7. Rosu, G., Bensalem, S.: Allen linear (interval) temporal logic - translation to LTL and monitor synthesis. In: CAV (2006)