Mining Temporal Intervals from Real-Time System Traces

Sean Kauffman
Electrical & Computer Engineering
University of Waterloo, Canada
Email: skauffma@uwaterloo.ca

Sebastian Fischmeister
Electrical & Computer Engineering
University of Waterloo, Canada
Email: sfischme@uwaterloo.ca

***Abstract*—We introduce a novel algorithm for mining temporal intervals from real-time system traces with linear complexity using passive, black-box learning. Our interest is in mining `nfer` specifications from spacecraft telemetry to improve human and machine comprehension. `Nfer` is a recently proposed formalism for inferring event stream abstractions with a rule notation based on Allen Logic. The problem of mining Allen's relations from a multivariate interval series is well studied, but little attention has been paid to generating such a series from symbolic time sequences such as system traces. We propose a method to automatically generate an interval series from real-time system traces so that they may be used as inputs to existing algorithms to mine nfer rules. Our algorithm has linear runtime and constant space complexity in the length of the trace and can mine infrequent intervals of arbitrary length from incomplete traces. The paper includes results from case studies using logs from the Curiosity rover on Mars and two other realistic datasets.**

I. Introduction

Earth-based personnel must construct their understanding of a spacecraft's operation from event sequences called *telemetry*. `Nfer` is a recently introduced formalism for inferring abstractions of such event sequences for easier human and machine comprehension [1]. The `nfer` notation is based on Allen's Temporal Logic (ATL), which has been traditionally employed in the planning domain. `Nfer` resembles classical rule-based systems from AI, in that the result of applying a specification to a trace is a collection of fact-like items.

To use the results from `nfer` to improve telemetry comprehension, rules must specify the interval abstractions to infer from the trace. Typically, such a specification would be written by the engineers that designed the software. Writing specifications is time-consuming and error-prone, however, and only captures facets of the software that were understood at design time. Dynamic specification mining seeks to solve these problems by using machine learning techniques to discover rules that define how the system should behave.

We seek to mine `nfer` specifications from historical telemetry, but prior research in pattern mining using Allen-like relations has assumed the existence of a multivariate interval series. To be able to use existing research in interval pattern mining, we must first convert our telemetry (what Masseglia called a symbolic time sequence [2]) to a multivariate interval series.

Our requirements lead to an approach that combines aspects of sequential pattern mining and automata-based specification mining. We are interested in finding intervals that resemble what Dwyer called *response* patterns [3] and what Mannila called *serial episodes* [4]. We mine patterns in event sequences that define the beginning and the end of a *process* in a real-time embedded system such as a spacecraft. A process may be any task, routine, or function that is part of the behavior of the system. Since our technique outputs `nfer` rules, the mined response patterns are expressed as ATL **before** relations.

Our contributions are the following. We introduce a black box, passive, learning algorithm for mining interesting temporal intervals from a symbolic time sequence generated by the execution of a real-time embedded system. Our algorithm is simple and easy to implement and has linear runtime complexity and constant space complexity in the size of the trace. Unlike previous efforts, we are able to mine episodes of arbitrary length and frequency while supporting incomplete traces. We justify the heuristics used in our technique both through careful analysis and empirical research using realistic case studies.

The rest of the paper is organized as follows. In Section II we provide necessary background information. Section III lists existing works related to our research. Section IV defines the problem more rigorously. Section V discusses the how to determine that an interval is *interesting*, Section VI describes our algorithm, and Section VII gives an example of applying it to a simple trace. Section VIII describes the case studies we performed to evaluate the effectiveness of our technique. Section IX concludes the paper.

II. Background

$\mathbb{B}$ denotes the set of Boolean values $\{true, false\}$, $\mathbb{N}$ denotes the set of all natural numbers and $\mathbb{R}$ denotes the set of all real numbers. The notation $\mathbb{C} = \mathbb{R}$ represents clock time stamps. $\mathcal{I}$ is the set of all event names. Given a set S, the set of finite sequences of S is S^*.

An *event* is a pair $\mathbb{E} = \mathcal{I} \times \mathbb{C}$, written (η, t), where $\eta \in \mathcal{I}$ is the event name, and $t \in \mathbb{C}$ is the time when the event occurred. A trace is a partially ordered sequence of events $(\eta_1, t_1), \cdots, (\eta_k, t_k) | (\eta_1, t_1), (\eta_i, t_i) \in \mathbb{E},\ t_{i-1} \leq t_i\ \ \forall i \in \mathbb{N} | 1 < i \leq k$. The set of all traces is defined as $\mathbb{T} = \mathbb{E}^*$. In our context, a trace corresponds to a symbolic time sequence generated by the execution of a real-time embedded system. Figure 1 shows an example of a trace. In the figure, the event names are listed above the timeline and event times

are listed below. Note that neither event names nor times must be unique in a trace.

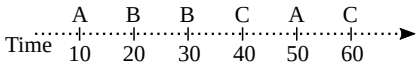

Fig. 1. Example trace

An *interval* is a triple $\mathbb{I} = \mathcal{I} \times \mathbb{C} \times \mathbb{C}$, written (η, b, e), where $\eta \in \mathcal{I}$ is the interval name, $b, e \in \mathbb{C}$ are time stamps which satisfy the condition $b \leq e$ where b represents the time when the interval began and e represents the time when it ended. The *duration* of an interval $(\eta, b, e) \in \mathbb{I}$ is equal to $e - b$. An *atomic* interval is any interval $(\eta, t, t) \in \mathbb{I}$. Given two intervals $(A, b_1, e_1), (B, b_2, e_2) \in \mathbb{I}$ the ATL relation "A **before** B" denotes that $e_1 < b_2$.

III. Related Work

The problem of mining patterns based on Allen's interval relations from a multivariate interval series has been thoroughly studied. Kam et al. looked for nested combinations of ATL patterns in an interval series [5]. Höppner used Allen's transitivity law in a sliding window to reduce the number of candidate patterns [6]. Likewise, De Amo et al. constrained candidate ATL patterns using regular expressions [7].

Our work is closely related to the field of specification mining using dynamic inference. Although we do not seek to mine general property specifications, we are interested in what Dwyer et al. called *response* and *precedence* patterns [3]. Yang et al. proposed solutions to the scaling problems of dynamic property inference in their Perracotta tool [8]. Ernst et al. introduced the popular Daikon tool that uses a library of patterns to efficiently check for program invariants [9]. Le Goues and Weimer found that they could reduce false positives in the mined properties by incorporating trustworthiness metrics [10]. Reger et al. introduced parametric specification mining using Quantified Event Automata (QEA) [11] and later proposed support for imperfect traces [12]. Lemieux et al. supported user-defined Linear Temporal Logic (LTL) patterns and imperfect traces in their Texada tool [13]. Cutulenco et al. efficiently mined Timed Regular Expressions (TREs) using a matrix of pattern automata [14].

A related and widely explored topic is *sequential pattern mining*, which seeks to find frequent subsequences of events in a database of sequences [2], [15]. Many sequential pattern mining works differ from ours because they seek to find arbitrary length patterns, because they expect the patterns to be frequent, and because the duration of the patterns is limited. Agraval and Srikant are often credited with introducing the idea, and proposed a set of Apriori-based algorithms for finding such consecutive subsequences [16]. Mannila et al. used sliding-window algorithms with pattern matching automata to mine frequent episodes, including serial episodes, from event traces [4]. Han et al. introduced a series of algorithms to reduce the search space of such algorithms by using tree-based data structures and by projecting subsequences into smaller databases [17], [18]. This work culminated in the well-known CloSpan algorithm by Yan et al. to find *closed* sequences, meaning sequences where no supersequence exists with the same support [19].

Some existing sequential pattern mining techniques have focused on performance. Zaki proposed Sequential PAttern Discovery using Equivalence classes (SPACE), which complexity improvements using combinatorial properties to decompose the problem using lattice search techniques [20]. Ayres et al. used a bitmap representation in their Sequential PAttern Mining (SPAM) tool to achieve improved performance over SPACE, but with a prohibitive increase in its memory requirements [21]. Wang and Han introduced their BI-Directional Extension (BIDE) tool to mine frequent closed sequences and achieved an order of magnitude improvement in speed over CloSpan [22]. They measured linear execution time and memory scalability in the size of the trace in their experiments. We are not, however, interested in mining frequent or closed sequences.

Ding et al. introduced a variation called *repetitive pattern mining* which more closely resembles our work [23]. Repetitive pattern mining considers infrequent patterns of arbitrary length that may repeat in a trace. Our work achieves improved time and space complexity over theirs but our application is more specialized to finding patterns of length two in real-time system traces.

IV. Problem Statement

Given a trace $\tau \in \mathbb{T}$, find pairs of event names $(\eta_1, \eta_2) \in \mathcal{I} \times \mathcal{I}$ such that $\exists(\eta_1, t_1), (\eta_2, t_2) \in \tau$ and $t_1 < t_2$, and such that $\forall(\eta_1, t_j), (\eta_2, t_k) \in \tau$ the intervals $(\ \cdot\ , t_j, t_k) \in \mathbb{I}$ are *interesting*.

It is simple to convert an event trace into a sequence of atomic intervals, but this is insufficient for our purposes. Any event may be converted to an atomic interval by applying a simple transformation function *e2i* : $\mathbb{E} \to \mathbb{I}$, with the definition $e2i((\eta, t)) = (\eta, t, t)$. By applying this function to each event in a trace, we can create an interval sequence called an *atomic interval sequence*. However, such a sequence is not more useful than the original symbolic time sequence for deriving meaning from the trace or as the input to an algorithm meant to mine ATL relations.

We observe that an algorithm to mine Allen's or other temporal relations on an interval series will not work on an atomic interval sequence. The only Allen relation that can be applied to an atomic interval sequence to define intervals is **before**. We designed our algorithm therefore to mine **before** relations from a symbolic time sequence. The intervals that result from the mined relations may then be used for human or machine comprehension, or combined with traditional algorithms meant to mine temporal relations from an interval series.

V. Measuring Interestingness

We must also address the problem of whether or not intervals are *interesting*. Although the concept is subjective, we can observe properties that make some intervals more interesting than others. It is often assumed that an interesting

interval is one that is derived from a rule that also appears in a handwritten specification. This definition is only helpful for judging the quality of a mining algorithm, however. We must define metrics that we can use to mine intervals when no specification exists. Below, we argue the value of the heuristics we use to define the interestingness of intervals.

A. Minimality

A *minimal* interval is one which does not **contain** another interval with the same name. Given a set of intervals $\pi \in \mathbb{I}$, an interval $(\eta, b_1, e_1) \in \pi$ is defined as *minimal* in π iff $\nexists(\eta, b_2, e_2) \in \pi \mid b_1 \leq b_2 \wedge e_2 \leq e_1$. Our algorithm mines **before** relations that match minimal intervals in the given trace.

The example in Figure 2 illustrates why minimality is a property of interesting intervals. The intervals 1, 2, and 3 all match the relation A **before** B, but only 1 and 2 are minimal. If A is the beginning of a process and B is its end, then interval 3 does not capture the intent of the relation.

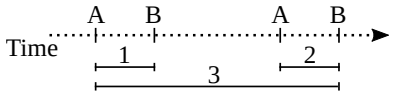

Fig. 2. Minimality example

B. Support and Confidence

We use the two most common metrics used to judge the acceptance of rules in specification or pattern mining: *support* and *confidence*. Support is usually defined as the number of times a rule is matched in the training data, and confidence is the probability that the post-condition of the rule follows the pre-condition of the rule.

Some research has suggested that these common metrics are less effective than other statistical measurements for judging the correctness of a rule. Le and Lo published a study that compared the effectiveness of different metrics in mining response and precedence patterns from the Java software development kit (SDK) source code [24]. In their work, they found that the "odds ratio" and "leverage" metrics outperformed support and confidence on average. Their definitions for computing the leverage and odds ratio metrics for a relation A **before** B include the notion of tracking the probability that A occurs independently and tracking the probability that neither A nor B occur.

However, Le's research makes assumptions that do not apply to this work. Most importantly, they assume the use of a sliding-window algorithm which is not sufficient to mine intervals of arbitrary duration. In our algorithm, which does not use a sliding window, the probability of A occurring independently will be either one or zero, and the same applies to the probability of neither A nor B occurring. Additionally, they used instrumented Java code in their research, where they added logging to the beginning of each method. Alternately, our traces represent telemetry or similar system traces logged from real-time systems, where the data is periodic, and events often occur at the *end* as well as the beginning of processes.

C. Relative Duration

We make the assumption that the average duration of an interval that represents the execution of a process will usually be shorter than the average duration of the time between executions. The real-time software that generates the event streams from which we mine intervals is made up of processes that are mostly *periodic*. Even when they can be interrupted, tasks are usually executed in a cycle where the system wakes up, performs work, and shuts down.

In developing our algorithm, the periodic nature of embedded software presented a challenge. In most cases, when the support and confidence thresholds are met for a relation A **before** B, they are also met for the relation B **before** A. In a periodic process, one of those intervals represents the time between when a process starts and ends (its duration) and the other represents the time between when it ends and starts again. The time between when a process ends and when it starts again is called its between-job inter-arrival time (BJI). If a process is interrupted its duration may be extended. The time during which it is interrupted is called its intra-job inter-arrival time (IJI).

Figure 3 shows a trace and the intervals that represent a periodic task execution. In the figure, A marks the beginning of the task and B marks the end, while I marks the beginning of an interrupt service routine (ISR) and J marks its end. Interval 1 shows an IJI of the task, interval 3 shows a BJI of the task, and intervals 2 and 4 show instances of the task's duration.

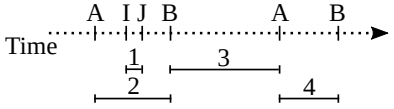

Fig. 3. Periodic processes

Without further heuristics, it is not possible to differentiate between the BJI and duration of a periodic process. When a process repeats, the result will be an alternating pattern of A and B events. Since we do not assume a complete trace, we cannot simply treat the first of the two to appear in the trace as the beginning of the process' execution.

A recent work by Iegorov et al. used the relationship between the IJI and the BJI of a process to mine strictly periodic tasks and their response times [25]. They used an assumption that the IJI of a process should be shorter than its BJI on average. This assumption comes from the predictable way that real-time software is scheduled and the fact that such systems are designed with a maximum expected CPU utilization. We use the same idea to differentiate between the duration of a process and its BJI.

We performed a simulation using the YAO-SIM tool [26] to see if a relationship exists between the duration of a task and its BJI that could be used to differentiate between them. Following a similar procedure to [25] and [26], we generated 1,000 periodic task sets for CPU utilizations between 10% and 70% at 5% increments and simulated execution of the system for 1×10^7 time units. We used two scheduling algorithms

common in real-time embedded systems: Rate Monotonic (RM) scheduling with fixed priorities, and Earliest Deadline First (EDF) scheduling with dynamic priorities. The number of tasks in each task set and their worst-case execution times (WCETs) were randomly chosen from a uniform distribution and the period of each task was computed from its WCET and the CPU utilization using the UUniFast algorithm [27]. The number of tasks was selected from the interval $[3, 10]$ and the WCET was taken from the interval $[1, 30]$.

Figure 4 shows the results of a scheduling simulation comparing the duration of a task to its BJI. The y-axis represents the ratio of the mean of duration for a task set over the mean of BJI, while the x-axis represents the CPU utilization of the system. At each CPU utilization level, the results using EDF scheduling are on the left and those using RM scheduling are on the right. Dots that appear below the horizontal line represent simulations where a task's average duration was shorter than its BJI, while dots that appear above the line represent simulations where the reverse was true.

The choice of 70% CPU utilization as the upper bound of the simulations was not arbitrary. The commonly accepted notions of safe CPU utilization in real-time systems are that 69% is the theoretical safe upper limit, 51-68% is considered "safe", 26-50% is considered "very safe", and below 26% is considered "unnecessarily safe" [28]. These regions are shown in Figure 4 by differently shaded backgrounds. For any system in the "very safe" or "unnecessarily safe" regions, our simulations show that any task's duration should always be shorter than its BJI. For systems in the "safe" region, this assumption still holds most of the time.

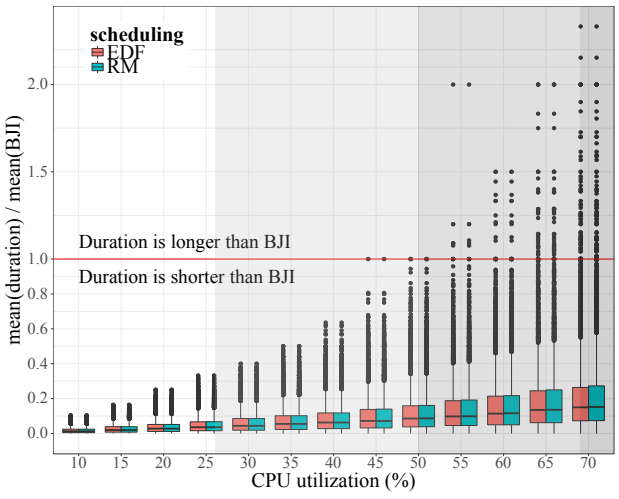

Fig. 4. Simulation results comparing duration with BJI

VI. Proposed Solution

This section describes our solution to mine pairs of event names which correspond to the beginnings and ends of minimal intervals. The details are shown in Algorithm 1. In practice, we mine `nfer` rules with the **before** relation, but the details of such rules are beyond the scope of this work. Our algorithm is loosely based on work by Cutulenco et al. to mine TREs [14] in that we achieve linear asymptotic complexity in the length of the trace using a matrix of pattern statistics.

The idea behind the algorithm is as follows. We process the trace in order. When we encounter an event with name A, we store it as the most recent A event. For every succeeding event B, we count the relation A **before** B as *matched* and record the duration of the interval. When the next A event is encountered, we increment a success counter for all A **before** $\cdot$ relations marked as matched exactly once and increment a failure counter otherwise. We also update the average duration of successful relations. When all events have been read, we check each pair of event names and output the associated rule if the user specified confidence and support thresholds are exceeded and if the average duration is shorter than that of the inverse relation.

We create a square matrix data structure $\mathcal{M}$ where each dimension maps to a name in the trace alphabet $\Sigma \subseteq \mathcal{I}$. The rows of $\mathcal{M}$ represent the left side of a **before** relation, and the columns represent the right side. Each entry $\mathcal{M}(i, j)$ in the matrix contains a 5-tuple (m, s, f, pd, ad) where $m \in \mathbb{N}$ represents the number of times the event name $\Sigma(j)$ has been seen since the last $\Sigma(i)$ (the *matched* count), $s, f \in \mathbb{N}$ are *success* and *failure* counts, and $pd, ad \in \mathbb{R}$ are the *previous duration* and the *average duration* of successfully matched pairs. We also keep an array $\mathcal{R}$, where indices map to names in Σ and contain the most recent copy of each event if one has been seen.

The user must specify a support threshold $\mathcal{S}_t$ and a *confidence* threshold $\mathcal{C}_t$. A count is kept of the *success* and *failure* of each event to imply every other event. The support for a pair is defined as the total number of successes for that pair. This varies somewhat from other notions of support because we do not use a sliding window in our algorithm and events may occur many times in the trace. The confidence for a pair is defined as the number of successes for that pair over the sum of the successes and failures.

After the trace is complete, the matrix is iterated over, and each cell is checked to see if it meets the acceptance conditions. A pair $(\Sigma(i), \Sigma(j))$ is accepted if the following conditions hold.

1) The event is not matching itself: $i \neq j$.
2) The confidence threshold is met: $\frac{\mathcal{M}(i,j).s}{\mathcal{M}(i,j).s + \mathcal{M}(i,j).f} \geqslant \mathcal{C}_t$.
3) The support threshold is met: $\mathcal{M}(i, j).s \geqslant \mathcal{S}_t$.
4) The average duration is less than the average duration of the inverse pair: $\mathcal{M}(i, j).ad < \mathcal{M}(j, i).ad$.

After a trace has been processed, the matrix $\mathcal{M}$ is traversed in a finalization step that facilitates handling multiple, non-contiguous traces. For each cell of the matrix, its *success* count is incremented if its *matched* count is exactly one and its *failure* count is incremented if its *matched* count is *greater than* one. Importantly, its *failure* count is not incremented if the *matched* count is zero. This causes the algorithm to assume that the behavior after the trace will not invalidate any of the relations. The matched count of each cell is also set to zero

Algorithm 1 Interval Mining Algorithm

```
1:  procedure ADDTOMODEL(event)
2:      define i : Σ(i) = event.name
3:      for left ∈ Σ.indices do
4:          if R(left) is set then
5:              // if the before relation holds
6:              if R(left).time < event.time then
7:                  increment M(left, i).matched
8:                  M(left, i).pd ← event.time − R(left).time
9:      end for
10:     if R(event.name) is set then
11:         for right ∈ Σ.indices do
12:             if M(i, right).matched = 1 then
13:                 increment M(i, right).success
14:                 M(i, right).ad ← M(i, right).ad + (M(i, right).pd − M(i, right).ad)/M(i, right).success
15:             else
16:                 increment M(i, right).failure
17:             M(i, right).matched ← 0
18:         end for
19:     R(event.name) ← event
```

and the array of the most recent events ($\mathcal{R}$) is cleared. This final step enables handling traces with few events such as the example in Section VII and the case study in Section VIII-A.

VII. Example

In this section, we present an illustrative example of the mining algorithm discussed in Section VI. Assume a confidence threshold $\mathcal{C}_t$ of 1.0, a support threshold $\mathcal{S}_t$ of 1, and the trace shown in Figure 1: (A, 10), (B, 20), (B, 30), (C, 40), (A, 50), (C, 60). The state of the matrix $\mathcal{M}$ and the array $\mathcal{R}$ are shown after each event is added to the model. Each field of $\mathcal{M}$ contains the 5-tuple (m, s, f, pd, ad), and each field of $\mathcal{R}$ contains the most recent event with that name.

- ADDTOMODEL(**(A,10)**) – The indices corresponding to A, B, and C in Σ are looped over on Line 3, but $\mathcal{R}$ is empty so the condition on Line 4 is false and the procedure continues on Line 10. The condition on Line 10 is also false because $\mathcal{R}$ is empty, so the event is simply inserted into $\mathcal{R}$ on Line 19.

$\mathcal{M}$:

	A	B	C
A	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0
B	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0
C	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0

$\mathcal{R}$:

A	(A,10)
B	
C	

- ADDTOMODEL(**(B,20)**) – There is now an A event in $\mathcal{R}$, so the conditional is taken for that event on Line 4 and (A,10) is compared to (B,20) on Line 6. Since $10 < 20$, *matched* is incremented for the pair (A,B) on Line 7 and its *previous duration* is set on Line 8. No B event is in $\mathcal{R}$, so the condition on Line 10 is false. The event is inserted into $\mathcal{R}$ on Line 19.

$\mathcal{M}$:

	A	B	C
A	0,0,0,0,0	1,0,0,10,0	0,0,0,0,0
B	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0
C	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0

$\mathcal{R}$:

A	(A,10)
B	(B,20)
C	

- ADDTOMODEL(**(B,30)**) – There are now A and B events in $\mathcal{R}$, so the conditional on Line 4 is taken for those. The matched count for (A,B) and (B,B) are incremented, and their *previous durations* are set. The pair (A,B) has been seen twice, so its matched count is now two. For the first time, the condition on Line 10 is true, since B has been seen before. The indices corresponding to A, B, and C in Σ are looped over on Line 11 and the pairs (B,A), (B,B), and (B,C) are checked to see if matched equals one on Line 12. This is only true for (B,B), so its *success* counter is incremented on Line 13 and its *average duration* is updated on Line 14. The failure counters for (B,A) and (B,C) are incremented on Line 16. All three *matched* counters are reset to zero on Line 17. Finally, the index for B in $\mathcal{R}$ is replaced with (B,30).

$\mathcal{M}$:

	A	B	C
A	0,0,0,0,0	2,0,0,20,0	0,0,0,0,0
B	0,0,1,0,0	0,1,0,10,10	0,0,1,0,0
C	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0

$\mathcal{R}$:

A	(A,10)
B	(B,30)
C	

- ADDTOMODEL(**(C,40)**) – There are both A and B events in $\mathcal{R}$, so the conditional on Line 4 is taken for those. Both (A,C) and (B,C) have their *matched* counters incremented, their *previous durations* are recorded, and the event is inserted into $\mathcal{R}$.

$\mathcal{M}$:

	A	B	C
A	0,0,0,0,0	2,0,0,20,0	1,0,0,30,0
B	0,0,1,0,0	0,1,0,10,10	1,0,1,10,0
C	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0

$\mathcal{R}$:

A	(A,10)
B	(B,30)
C	(C,40)

- ADDTOMODEL(**(A,50)**) – All three event names have been seen, so the condition on Line 4 is true in all cases. The pairs (A,A), (B,A), and (C,A) all have their *matched* counters incremented on Line 7 and their *previous durations* updated on Line 8. The condition on Line 10 is true again, so the pairs (A,A), (A,B), and (A,C) are checked to see if they have been matched once. The pairs (A,A) and (A,C) have been matched once, so their *success* counters are incremented and average durations updated. However, (A,B) has been matched twice, so its *failure* counter is updated. All three *matched* counters are reset. The index for A in $\mathcal{R}$ is replaced with (A,50).

$\mathcal{M}$:

	A	B	C
A	0,1,0,40,40	0,0,1,20,0	0,1,0,30,30
B	1,0,1,20,0	0,1,0,10,10	1,0,1,10,0
C	1,0,0,10,0	0,0,0,0,0	0,0,0,0,0

$\mathcal{R}$:

A	(A,50)
B	(B,30)
C	(C,40)

- ADDTOMODEL(**(C,60)**) – The pairs (A,C), (B,C), and (C,C) have their *matched* counters incremented and *previous durations* updated. On Line 13, the *success* counter for the pairs (C,A) and (C,C) are incremented and their *average duration* updated, as they have been matched, while on Line 16 the *failure* counter for the pair (C,B) is incremented since its *matched* count is zero. The event is added to $\mathcal{R}$, and the trace is complete.

$\mathcal{M}$:

	A	B	C
A	0,1,0,40,40	0,0,1,20,0	1,1,0,10,30
B	1,0,1,20,0	0,1,0,10,10	2,0,1,30,0
C	0,1,0,10,10	0,0,1,0,0	0,1,0,20,20

$\mathcal{R}$:

A	(A,50)
B	(B,30)
C	(C,60)

- Finalize – After the trace has completed, each cell of the matrix is iterated over, and the *success* count is incremented if the cell's *matched* count is exactly one. The *success* counters are incremented for (B,A) and (A,C) and their *average durations* are updated. Failure counts are only incremented if the call's *matched* count is greater than one, so this is done for (B,C). All *matched* counts are reset to zero and $\mathcal{R}$ is cleared.

$\mathcal{M}$:

	A	B	C
A	0,1,0,40,40	0,0,1,20,0	0,2,0,10,20
B	0,1,1,20,20	0,1,0,10,10	0,0,2,30,0
C	0,1,0,10,10	0,0,1,0,0	0,1,0,20,20

$\mathcal{R}$:

A	
B	
C	

The result of processing the example trace is that the pair (C,A) is output. Although (C,A) only appears once in the *visible* trace, the trace is assumed to be incomplete and contains no evidence to refute the hypothesis that A **before** C is an interesting relation.

The other pairs in $\mathcal{M}$ are rejected for the following reasons.

- (A,A), (B,B), (C,C) – fail condition 1, that the event is not matching itself
- (B,A), (A,B), (C,B), (B,C) – fail condition 2, that the confidence threshold of 1.0 is met
- (A,B), (C,B), (B,C) – fail condition 3, that the support threshold of 1 is met
- (A,C) – fails condition 4, that the average duration is shorter than that of the inverse pair

VIII. CASE STUDIES

We performed a series of case studies to demonstrate the viability of our algorithm. We ran the algorithm on datasets for which we had a handwritten `nfer` specification and compared the rules in those specifications to those that our algorithm mined from the same datasets. Although `nfer` rules concern a hierarchy of intervals, we were only interested in comparing against **before** relations on events rather than general ATL relations on intervals.

We implemented our algorithm in the C programming language and support both a command-line tool and an R-language interface. Our tool currently loads the entire trace into memory before processing, but this is an implementation detail, not a requirement of the algorithm. We ran the command-line version of the tool, compiled using the GNU Compiler Collection (GCC) 4.9.4 with -O3 optimizations.

Experiments were performed in Linux 4.9.6 on an Intel Core i5 at 2.4 GHz with 16 GB of RAM. We obtained execution time information from the GNU *time* command and memory usage from the Valgrind Massif heap profiler. We ran each experiment 20 times and took the mean of their execution times, while the memory usage was fully deterministic.

All experiments were run with the confidence threshold $\mathcal{C}_t = 0.90$ and the support threshold $\mathcal{S}_t = 10$. Tuning these parameters is difficult without an established ground truth the algorithm is attempting to match for a dataset. We found, through experimentation, that these values were effective in cases where such a ground truth was known.

A. SSPS Dataset

The Sequential Sense-Process-Send (SSPS) dataset consists of application logs from software 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. Logs were collected using the QNX tracelogger utility. The tested dataset contained 1,451,193 events broken up into 404 trace files. Our tool ran on this dataset in 0.88 seconds and used 14.1 MB of memory.

The SSPS system executes a periodic process with three phases: acquisition, processing, and communication. Data acquisition is accomplished using the *dd* utility to copy random data to a file. There are two phases of data processing. The first creates a checksum for the data using *cksum* and the second compresses the data using *bzip2*. The communication phase consists of using *scp* to transfer the compressed file to another host on the network. After the communication phase, the process is made pseudo-periodic by sleeping for a period.

The system logs an event between each phase, and between the two parts of the processing phase. This complicates log comprehension, as each event must serve as both the end of the previous phase and the beginning of the next. The trace also contains *anomalies*, in that the communication phase is sometimes absent. These issues are illustrated in Figure 5, which shows the phases of execution and their relationship to the events in the trace. In the figure, event `D` and *scp* interval are highlighted because they are absent from anomalous traces.

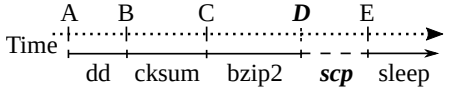

Fig. 5. SSPS events and application phases

Another challenge of the SSPS dataset was that it was broken up into 404 separate traces that mostly contained events related to interrupt handling. Each trace contained, on average, only 8.6 events related to phases of the application. This meant that a single trace usually did not contain events from two full

cycles of the main loop. These traces were non-contiguous, so events were missing between them and they could not simply be concatenated together and treated as one trace.

Our algorithm mined relations that established the sequential nature of the main application with no incorrect rules.

The most interesting aspect of the mined relations from the SSPS dataset was that some of the rules defined the BJI of a phase rather than its duration. This highlighted a problem *with the dataset* rather than with our algorithm. We discovered that the system that generated the logs had an average CPU utilization of 84%. As this is above the theoretical safe limit for a real-time system of 69% (see Section V-C) we should expect some tasks' BJIs to become shorter than their durations.

B. LANL Dataset

The System Call Logs with Natural Random Faults (LANL) dataset consists of application logs from a simulation of an automotive cruise-control application on a CPU under ionizing radiation bombardment [29]. The DUT was a Xilinx ZC706 featuring a XC7Z045 System-on-a-Chip (SoC) running version 6.6.0 of the QNX real-time operating system. The dataset contains faults from placing the SoC in the path of a high energy neutron beam at the Los Alamos Neutron Science Center (LANSCE) facility in New Mexico, USA. The dataset contained 100,000 events. Our tool ran on this dataset in 0.06 seconds and used 12.6 MB of memory.

The `nfer` specification for the LANL dataset defines request-response behavior for sensors, controllers, and actuators and then defines nominal and off-nominal relations between them. The cruise-control application that generated the dataset executes in a polling loop, but most events occur in response to external factors. The intervals that are meant to be derived from the event stream are the request-response patterns, where the response may be nominal or off-nominal.

The `nfer` specification for the LANL dataset defines periods of activity for each component and then defines the nominal and off-nominal behavior during those periods. During the period of activity for the speed sensor, the nominal behavior is for the speed value to be sent and no off-nominal behavior is defined. During the period of activity for the actuators, the nominal behavior is for a unanimous response to be received after a request to the controllers for commands is sent and the off-nominal behavior is for a non-unanimous or non-quorum response to be received. During the period of activity for the controllers, the nominal behavior is for the correct actuator to acknowledge receipt after a controller sends its command and the off-nominal behavior is for an incorrect actuator to acknowledge receipt of the command.

Our algorithm found every relation describing nominal behavior for the speed sensor and actuator components, but was unable to find off-nominal behavior relations. This is not surprising, as the faults that are part of the off-nominal behavior occur infrequently in the trace. As a result, the relations that include events from faults do not have enough support to be mined.

More surprisingly, many nominal behavior relations describing the controller functionality were missing from the mined rules. The only relation we mined from the controller behavior was from two events that appeared sequentially in the source code with only inter-process communication occurring between them. We discovered that the missing rules were explained by the parallel execution of multiple controllers. This design caused events to appear in non-deterministic order.

Figure 6 shows the problem of the parallel execution of multiple controllers in the LANL dataset. The events from every controller shared the same names, so it became impossible to find the correct relations without having a way to differentiate between them. We discovered that the controllers logged their process identifiers as data parameters in the trace, so we would be able to differentiate between the processes if we supported the notion that data parameters should be equal, as in the case of [11]. For now, we suggest that users choose unique event names per process instance. We plan to incorporate data parameters into our algorithm in future work.

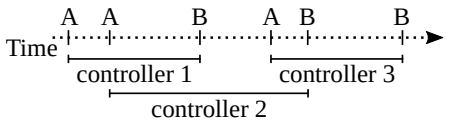

Fig. 6. LANL parallel controller execution

C. MSL Dataset

The Mars Science Laboratory (MSL) dataset consists of telemetry in the form of EVent Reports (EVRs) received from the Curiosity rover on Mars. Experiments with these logs were made possible through collaboration with researchers at the National Aeronautics and Space Administration (NASA) Jet Propulsion Laboratory (JPL) in Pasadena, California, USA. The events in the MSL logs are generated by the rover when commands are executed from daily activity plans uploaded by controllers on Earth. The logs cover a period of about 60 days and are filtered only to contain the events relevant to the available `nfer` specification, written for the case study in [1]. The dataset contained 49,999 events. Our tool ran on this dataset in 0.64 seconds and used 45.7 MB of memory.

The `nfer` specification for the MSL dataset defines situations where errors that have been reported by the spacecraft can safely be ignored. We specifically analyzed one set of rules that defines a benign race condition where the routine servicing a radio reports missing telemetry due to thread starvation. The intervals that are meant to be derived take the form of a command being *dispatched* and later *completing*. The same pattern of *dispatch* **before** *complete* follows for 464 different types of commands in the trace, all intermingled together.

Our algorithm found 117 pairs of *dispatch* **before** *complete* relations for the same command. Encouragingly, no *complete* **before** *dispatch* rules were mined for the same command, meaning that our assumption that the duration should be shorter than the BJI held throughout.

A scientist at JPL examined the mined rules from running our algorithm on the MSL dataset. He found that the rules

captured some useful information about how the spacecraft behaves, and we were able to use his analysis to verify that a sequential task was correctly identified. For example, a set of rules successfully describes a periodic activity that loads a schedule table for the Rover Environmental Monitoring Station (REMS) instrument that monitors Mars' weather. The table tells the REMS instrument what data to collect and consists of a sequence of six commands. Our algorithm found relations between *dispatch* and *complete* events for each command, but it also found relations for the different commands in sequence.

IX. Conclusion

In this work, we present an algorithm for mining interesting temporal intervals from real-time system traces. Our algorithm uses black-box, passive learning to mine ATL **before** relations in linear time and constant space complexity in the size of the trace. We justify our heuristics and present three case studies that demonstrate the value of our contribution.

Our choices of heuristics to define an interesting interval are based on our interest in mining representations of the duration of processes in real-time systems. We rely on the minimality of these intervals to avoid the use of a sliding window in our algorithm and support arbitrarily long and infrequent relations. We use a property of real-time systems scheduling to support incomplete traces and improve the quality of our results.

Our ongoing and future work includes supporting data parameters in our mining algorithm. We describe in Section VIII-B how some intervals in which we are interested cannot be mined because it would require differentiating between processes based on their data parameters. `Nfer` supports arbitrary expressions on data maps, so some discretion must be used to avoid drastic increases in runtime complexity.

Acknowledgements

We would like to thank Rajeev Joshi and Klaus Havelund at NASA JPL for enabling the MSL case study and the three anonymous reviewers for their valuable suggestions.

References

[1] S. Kauffman, K. Havelund, and R. Joshi, "nfer – a notation and system for inferring event stream abstractions," in *Runtime Verification: 16th International Conference, RV 2016, Madrid, Spain, September 23–30, 2016, Proceedings*, ser. LNCS, Y. Falcone and C. Sánchez, Eds., vol. 10012. Springer, 2016, pp. 235–250.

[2] F. Masseglia, M. Teisseire, and P. Poncelet, "Sequential pattern mining," in *Encyclopedia of Data Warehousing and Mining*. IGI Global, 2005, pp. 1028–1032.

[3] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Software Engineering, 1999. Proceedings of the 1999 International Conference on*. IEEE, 1999, pp. 411–420.

[4] H. Mannila, H. Toivonen, and A. Inkeri Verkamo, "Discovery of frequent episodes in event sequences," *Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 259–289, 1997.

[5] P.-s. Kam and A. W.-c. Fu, *Discovering Temporal Patterns for Interval-based Events*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 317–326.

[6] F. Höppner, "Discovery of temporal patterns," in *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 2001, pp. 192–203.

[7] S. De Amo, A. Giacometti, and W. P. Junior, "Mining first-order temporal interval patterns with regular expression constraints," in *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 2007, pp. 459–469.

[8] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal API rules from imperfect traces," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 282–291.

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

[10] C. Le Goues and W. Weimer, "Specification mining with few false positives," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2009, pp. 292–306.

[11] G. Reger, H. Barringer, and D. Rydeheard, "A pattern-based approach to parametric specification mining," in *28th IEEE/ACM Intl. Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 658–663.

[12] ——, "Automata-based pattern mining from imperfect traces," *SIGSOFT Softw. Eng. Notes*, vol. 40, no. 1, pp. 1–8, Feb. 2015.

[13] C. Lemieux, D. Park, and I. Beschastnikh, "General LTL specification mining (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2015, pp. 81–92.

[14] G. Cutulenco, Y. Joshi, A. Narayan, and S. Fischmeister, "Mining timed regular expressions from system traces," in *Proceedings of the 5th Intl. Workshop on Software Mining*, Singapore, 2016, pp. 3 – 10.

[15] F. Mörchen, "Unsupervised pattern mining from symbolic temporal data," *ACM SIGKDD Explorations Newsletter*, vol. 9, no. 1, pp. 41–55, 2007.

[16] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proceedings of the Eleventh International Conference on Data Engineering*, Mar 1995, pp. 3–14.

[17] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *SIGMOD Rec.*, vol. 29, no. 2, pp. 1–12, May 2000.

[18] J. Han, J. Pei, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, "Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth," in *proceedings of the 17th international conference on data engineering*, 2001, pp. 215–224.

[19] X. Yan, J. Han, and R. Afshar, "Clospan: Mining: Closed sequential patterns in large datasets," in *Proceedings of the 2003 SIAM International Conference on Data Mining*. SIAM, 2003, pp. 166–177.

[20] M. J. Zaki, "SPADE: An efficient algorithm for mining frequent sequences," *Machine Learning*, vol. 42, no. 1, pp. 31–60, 2001.

[21] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential pattern mining using a bitmap representation," in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '02. New York, NY, USA: ACM, 2002, pp. 429–435.

[22] J. Wang and J. Han, "BIDE: efficient mining of frequent closed sequences," in *Proceedings. 20th International Conference on Data Engineering*, March 2004, pp. 79–90.

[23] B. Ding, D. Lo, J. Han, and S. C. Khoo, "Efficient mining of closed repetitive gapped subsequences from a sequence database," in *2009 IEEE 25th Intl. Conference on Data Engineering*, March 2009, pp. 1024–1035.

[24] T.-D. B. Le and D. Lo, "Beyond support and confidence: Exploring interestingness measures for rule-based specification mining," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 331–340.

[25] O. Iegorov, R. Torres, and S. Fischmeister, "Periodic task mining in embedded system traces," in *23rd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Pittsburgh, PA, USA, Apr 2017.

[26] A. Bergmayr, H. Bruneliere, J. Cabot, J. Garcia, T. Mayerhofer, and M. Wimmer, "fREX: fUML-based reverse engineering of executable behavior for software dynamic analysis," in *Modeling in Software Engineering (MiSE), 2016 IEEE/ACM 8th International Workshop on*. IEEE, 2016, pp. 20–26.

[27] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1, pp. 129–154, 2005.

[28] P. A. Laplante *et al.*, *Real-time systems design and analysis*. Wiley New York, 2004.

[29] A. Narayan, S. Kauffman, J. Morgan, G. M. Tchamgoue, Y. Joshi, C. Hobbs, and S. Fischmeister, "System call logs with natural random faults: Experimental design and application," in *13th Workshop on Silicon Errors in Logic, System Effects*, Boston, MA, USA, 2017.