Visualizing Temporal Interval Hierarchies

Nafiz Sadman$^{[0000-0001-6784-0029]}$, Nastaran Kianersi$^{[0009-0009-5005-9645]}$, and Sean Kauffman$^{[0000-0001-6341-3898]}$

Queen's University, Kingston, Canada
{sadman.n,nastaran.kianersi,sean.k}@queensu.ca

Abstract. We introduce a method for visualizing the temporal intervals output by `nfer`, a language and tool for event trace comprehension. Visualizing these intervals is challenging because they may be arbitrarily nested in a hierarchy and because multiple intervals with the same identifier may occur concurrently. Our tool, `nvis`, solves these problems to present the output of `nfer` in a human interpretable, interactive web interface. We introduce an algorithm to solve the hierarchy and concurrency problem, and present a case study using automatically learned rules from our industrial partner.

1 Introduction

`Nfer` is a language and tool for event trace comprehension, designed to process logs from spacecraft and other systems and transform them into more easily consumable abstractions [10, 8, 9]. These logs contain discrete timestamped events, possibly carrying data, that describe state changes, such as a function call or device interrupt. `Nfer` allows for temporal reasoning over such events by applying human- or machine-written [6] *rules* to them to generate a hierarchy of labeled time-intervals. The intervals can then be used by other programs or people to understand how the system has behaved. `Nfer`'s syntax is loosely based on Allen's Temporal Logic (ATL) [3], which is widely used for automated planning, making it easy to adopt.

For humans to use `nfer` for event trace comprehension, the hierarchy of intervals it produces must be visualized. However, displaying the output of `nfer` comes with a unique set of challenges because of its hierarchical and concurrent nature. `Nfer` intervals can have positive *duration*, meaning they label some period of time. However, since these labeled intervals can also carry data, arbitrary numbers of intervals with the same identifier can label the same period, creating a problem of *overlap*. Additionally, each interval is the result of a fixed-point computation on other intervals, which it does not reference explicitly. Determining precisely which intervals are part of the *hierarchy* responsible for an interval is both highly interesting to users and undecidable in general [12].

While many tools visualize logs of events, no other tools we are aware of handle the two problems of concurrency and hierarchy that `nfer` output presents. For example, Shviz [5] displays interactive communication graphs where there is some notion of concurrency, but where events (the elements to be displayed) have no duration and cannot be nested in a hierarchy. Similarly, Tracy [14] visualizes the results of declarative rules applied to label events, but also lacks the same problems with concurrency and hierarchy.

In this work, we introduce algorithms and a tool, `nvis`, to efficiently visualize `nfer` output. In Section 2 we explain the problem in more detail and introduce a through-example. Then, Section 3 describes the algorithms that solve both the concurrency and hierarchy problems. Section 4 introduces the tool using a case study, and Section 5 concludes the paper.

2 Problem

The greatest challenges of displaying the intervals that `nfer` outputs come from their overlapping nature and their hierarchical structure. To explain these phenomena, we now describe the `nfer` language in more detail.

The `nfer` tool applies the rules to an event trace to produce labeled intervals. In this case, an event (η, t, M) is a triple where $\eta \in \mathcal{I}$ is one of a finite set of identifiers labeling what occurred, $t \in \mathbb{N}$ is a timestamp representing when the event happened, and M is a partial function $M : \mathcal{I} \nrightarrow \mathbb{N}$ mapping identifiers to numbers. An event trace is a finite sequence of events. The output of `nfer` is a set of intervals. An interval $i = (\eta, s, e, M) \in \mathbb{I}$ is similar to an event, but with two timestamps $s, e \in \mathbb{N} \centerdot s \leq e$ representing the start and end timestamps of a behaviour over time. We also define accessor functions $id(i) = \eta$, $start(i) = s$, and $end(i) = e$. Although the input to `nfer` is an event trace, we treat events (η, t, M) as intervals with zero duration (η, t, t, M), called *atomic intervals*.

`Nfer` rules, then, transform sets of atomic intervals into a set of intervals with potentially positive duration. `Nfer` rules relate these intervals using temporal relations inspired by ATL [3]. Allen formalized his algebra (ATL) after arguing that an interval-based notion of time was more natural than a point-based system [4]. That is, intervals that represent *state* are easier to reason about than events that represent state *transitions*.

An `nfer` rule $\eta \leftarrow \lambda$ has two parts: the right-hand side (λ) matches pairs of existing intervals by comparing their timestamps and data maps, and the left-hand side (η) is an identifier that labels the resulting intervals. Note that the timestamps and data map of the resulting intervals are determined by the right-hand side, but the details of this are beyond the scope of this work [8, 9, 11, 12]. Where necessary, we will give the intuition behind the rule semantics and otherwise rely on the reader to follow the relatively human-readable notation.

One important detail is the difference between *inclusive* and *exclusive* rules. Inclusive rules take the form $\eta \leftarrow \eta_1$ op η_2 and match pairs of existing intervals with identifiers equal to η_1 and η_2 where their timestamps satisfy a temporal constraint specified by the operator "op." The intervals produced by these rules have timestamps that depend on both the matched intervals and the temporal constraint. Alternatively, exclusive rules take the form $\eta \leftarrow \eta_1$ **unless** op η_2 and match an existing interval with identifier η_1 where there *does not exist* an interval with identifier η_2 matching the temporal constraint specified by "op." The intervals produced by these rules depend only on the extant interval.

Example 1. Here, we present a simple example of applying `nfer` to abstract an event trace. Suppose an embedded system that produces events on system calls and returns, such as from the QNX Tracelogger utility or Linux's ftrace. The system produces events with the following labels: $\mathcal{I} = \{$ IRQ, IRQRET, TASK $\}$ where IRQ represents an interrupt firing, IRQRET represents it returning, and TASK represents a task being scheduled.

The following `nfer` rule labels the intervals between an IRQ event and an IRQRET having the same n (number) with the label "interrupt":

$$\text{interrupt} \leftarrow \text{IRQ } \textbf{before} \text{ IRQRET } \textbf{where } m_1, m_2 \mapsto m_1(n) = m_2(n) \textbf{ map } \varnothing$$

Note that the **map** part of the rule is empty here, because we have not specified any data to assign to the resulting interval. If the rule is applied to the events $(\text{IRQ}, 1, 1, \{n \mapsto 9\})(\text{TASK}, 5, 5, \{id \mapsto 19\})(\text{IRQRET}, 10, 10, \{n \mapsto 9\})$, it produces the interval $(\text{interrupt}, 1, 10)$, where the identifier "interrupt" comes from the left-hand side of the rule, 1 comes from the IRQ event with $n = 9$, and 10 from the IRQRET event with $n = 9$.

A second rule might label interrupts that schedule a task, and capture the task id that was scheduled:

$$\text{schedule} \leftarrow \text{TASK } \textbf{during} \text{ interrupt } \textbf{where } \varnothing \textbf{ map } m_1, m_2 \mapsto \{id \mapsto m_1(id)\}$$

If this rule is applied in addition to the rule above, it produces the interval $(\text{schedule}, 1, 10, \{id \mapsto 19\})$ because it *matches* a TASK event from the original trace and the interrupt interval produced by the other rule. To see how this creates a unique challenge for visualization, suppose that two other events occur in the same trace: $(\text{IRQ}, 4, 4, \{n \mapsto 7\})(\text{IRQRET}, 12, 12, \{n \mapsto 7\})$. Applying the above rules results in two additional intervals to the two we already saw: $(\text{interrupt}, 4, 12)$ and $(\text{schedule}, 4, 12, \{id \mapsto 19\})$. The interrupt interval is produced because the two new events have matching n values and so are matched by the interrupt rule, and the schedule rule is matched because the TASK event also occurs during the new interrupt interval.

There are two challenges inherent in displaying these new intervals: *overlap* and *hierarchy.* The four intervals that were produced in the example all overlap from time points 4 to 10, with two intervals produced by each rule with the same identifiers. Indeed, the number of possible intervals at any given point, even without recursion, is 2^{2^n} in the number of events [11, 12]. The hierarchy is also unclear directly from the produced intervals, since both schedule intervals share the same TASK event but themselves have different timestamps (inherited from the different interrupt intervals).

3 Solution

We present a methodology to address hierarchical and overlapping intervals by dividing the period of the trace into segments (*bucketing*) where the number of intervals in a segment defines the shade of its color, and an over-approximation of the hierarchy is derived by ignoring data. Throughout this section, we use I and O to describe the input and output of `nfer` respectively, where $I, O \subseteq \mathbb{I}$ and I consists only of atomic intervals.

We first group the intervals in I and O by time segment to resolve the overlapping problem. To group by segment, we first divide the period of the input into l equal-length, non-overlapping bins from $\min_{1 \leq i \leq |I|} \mathit{start}(I_i)$ to $\max_{1 \leq i \leq |I|} \mathit{end}(I_i)$, $\theta_1 \cdots \theta_l$. For each time segment, we associate an interval when its time overlaps with that segment. For example, consider two subsequent time segments θ_i, θ_j where $\exists s, m, e \in \mathbb{N}$ such that $\theta_i = (s, m]$ and $\theta_j = (m, e]$. For some interval $o \in O$, we associate it with θ_i if $\mathit{start}(o) \leq m$ and $\mathit{end}(o) > s$ and with θ_j if $\mathit{start}(o) \leq e$ and $\mathit{end}(o) > m$. In other words, the algorithm includes each interval in a segment if any part of the interval intersects with the period of the segment. This grouping allows visualizing overlapping intervals by shading the segment based on the interval count. The complexity of this bucketing algorithm is $\mathcal{O}(|O|)$, since one can derive the segment given the start and end timestamps of a trace, the number of segments, and a timestamp of interest.

We illustrate the bucketing procedure for Example 1 in Figure 1. With three segments of length four, the leftmost segment has one schedule and interrupt interval, while the other two segments include two of each interval since all four intervals intersect those segments.

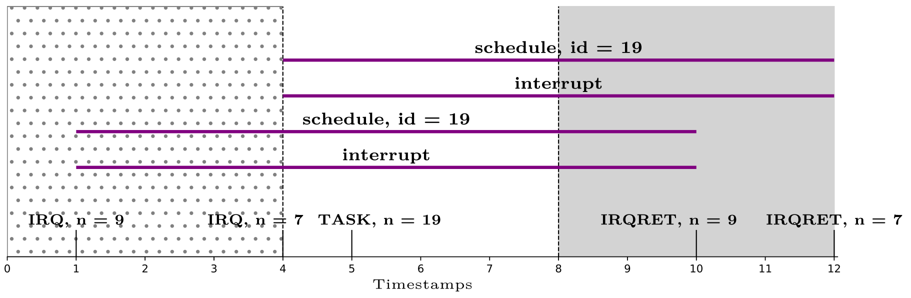

Fig. 1: Bucketing the intervals from Example 1.

The second objective is to solve hierarchical intervals. Finding the precise hierarchy of intervals that led to the creation of a specific interval is undecidable [12], so we introduce a complete but not sound method of computing the hierarchy (an over-approximation) that only considers temporal relations. We believe that, in practice, there are not many examples where uncertainty exists over which interval was used, i.e., there are not many cases where more than two parents will be found. We would rather show users all options and let them decide what is meaningful instead of omitting possibly correct choices.

We now give an algorithm to find valid parents for an interval. An interval $p \in I \cup O$ is a valid parent for a given child interval $c \in O$ iff there exists a rule $\eta \leftarrow \lambda$ in the set of rules Δ where $id(c) = \eta$ and the timestamps of p and c satisfy the temporal constraints of λ. This ignores the data map for the purpose of defining the hierarchy, which results in unsound parents being included, transforming the problem to evaluation of the data-free fragment of `nfer`, which is in PTime [7].

We define predicates $\textsc{match}_k : \mathbb{O} \to \mathbb{I} \times \mathbb{N} \times \mathbb{N} \to \mathbb{B}$ for $k \in \{L, R\}$ (Left, Right) where $\mathbb{O}$ is the set of the 8 inclusive `nfer` operators $\{\textbf{before}, \textbf{meet}, \textbf{coincide}, \textbf{during}, \textbf{start}, \textbf{finish}, \textbf{overlap}, \textbf{slice}\}$ [9] and $\mathbb{B} = \{true, false\}$. For $\textsc{match}_k(\text{op})(i, s, e)$, $i \in \mathbb{I}$ is a possible parent interval, and $s, e \in \mathbb{N}$ are timestamps of the given child interval. The parameter $k \in \{L, R\}$ represents the "side" of the operator. The predicates hold if i satisfies the temporal constraints of op for the k side of the operator. For example, $\textsc{match}_L(\textbf{before})(i_1, s, e) = start(i_1) = s \wedge end(i_1) \leq e$ and $\textsc{match}_R(\textbf{before})(i_2, s, e) = start(i_2) \geq s \wedge end(i) = e$. These predicates are defined based on the temporal operators from `nfer` [8, 9, 11, 12] in a straightforward way, so we do not repeat the definition here.

Algorithm 1 Processing Rule Associations

```
1: procedure $\textsc{Hierarchy}((\eta, s, e, M))$                          ▷ Argument is an interval
2:     parents = $\varnothing$
3:     for $\eta_1 \leftarrow \lambda \in \Delta.\ \eta_1 = \eta$ do                     ▷ Loop over rules with $\eta$ on LHS
4:         if $\lambda \in \mathcal{I}$ then                                          ▷ If rule is atomic
5:             parents = $\textsc{findParents}(\textbf{coincide}, L, \lambda, s, e)$
6:         else if $\lambda = \eta_1$ op $\eta_2$ then                              ▷ If rule is inclusive
7:             parents = $\textsc{findParents}(\text{op}, L, \eta_1, s, e) \cup \textsc{findParents}(\text{op}, R, \eta_2, s, e)$
8:         else if $\lambda = \eta_1$ unless op $\eta_2$ then                       ▷ If rule is exclusive
9:             parents = $\textsc{findParents}(\textbf{coincide}, L, \eta_1, s, e)$
10:    return parents
11: procedure $\textsc{findParents}(\text{op}, k, \eta_p, s, e)$
12:    parents = $\varnothing$
13:    for $i \in I \cup O : id(i) = \eta_p \wedge \textsc{match}_k(\text{op})(i, s, e)$ do        ▷ Loop over intervals
14:        parents $\leftarrow$ parents $\cup \{i\}$
15:        if $i \in O$ then
16:            parents $\leftarrow$ parents $\cup\ \textsc{Hierarchy}(i)$          ▷ Recurse if parent is not input
17:    return parents
```

Example 2. We now revisit the rules and intervals from Example 1 to explain Algorithm 1. Our goal is to determine the hierarchy that produced the interval $x = (\text{schedule}, 1, 10, \{id \mapsto 19\})$. We call the function $\textsc{Hierarchy}$ with x. For this example, $\eta = \text{schedule}$, $s = 1$, $e = 10$ and the map M is ignored. In Algorithm 1, I, O, and Δ are considered globally accessible to simplify presentation.

In Line 3, we search the rules $\eta_1 \leftarrow \lambda \in \Delta$ that produce η. λ can be either an atomic rule, consisting of only an identifier in $\mathcal{I}$ or a combination of two identifiers η_1, η_2 with an `nfer` operation (note that we ignore the map part of rules). If λ is atomic (i.e., $\lambda \in \mathcal{I}$), we satisfy the condition in Line 4, otherwise we either satisfy Line 6 for *inclusive* rules or Line 8 for *exclusive* rules [12]. In our example, we search the rules until finding the rule from Example 1 where the left-hand side is "schedule". The right-hand side for the rule (ignoring the map) is TASK **during** interrupt, an inclusive rule (Line 6). We then call $\textsc{Parents}(\mathbf{during}, L, \text{TASK}, 1, 10)$ and $\textsc{Parents}(\mathbf{during}, R, \text{interrupt}, 1, 10)$.

The first call, with $\eta_p = \text{TASK}$, searches $I \cup O$ for an interval with a matching identifier that meets the temporal constraint specified by $\textsc{match}_L(\mathbf{during})(i, s, e) = \mathit{start}(i) \geq s \wedge \mathit{end}(i) \leq e$. This is satisfied by $i_1 = (\text{TASK}, 5, 5, \{\mathit{id} \mapsto 19\})$ and since $i_1 \in I$ (not O), it is added to parents and we return. The second call to $\textsc{Parents}$, with $\eta_p = \text{interrupt}$, searches for intervals that meet $\textsc{match}_R(\mathbf{during})(i, s, e) = \mathit{start}(i) = s \wedge \mathit{end}(i) = e$, met by $i_2 = (\text{interrupt}, 1, 10)$. Since $i_2 \in O$, we then recurse, calling $\textsc{Hierarchy}$ on it.

The recursive call searches for rules that produce interrupt intervals, yielding the other rule from Example 1. The right-hand side IRQ **before** IRQRET is again an inclusive rule, yielding calls to $\textsc{Parents}(\mathbf{before}, L, \text{IRQ}, 1, 10)$ and $\textsc{Parents}(\mathbf{before}, R, \text{IRQRET}, 1, 10)$. Although Example 1 includes two copies of both events, Algorithm 1 will choose the correct ones because of the temporal constraints $\textsc{match}_k(\mathbf{before})$ we defined earlier: $(\text{IRQ}, 1, 1, \{n \mapsto 9\})$ and $(\text{IRQRET}, 10, 10, \{n \mapsto 9\})$.

Theorem 1 (Algorithm 1 Completeness). *Given an interval $i \in I \cup O$ and its ancestors $A \subseteq I \cup O$ from applying the rules in Δ, $\textsc{Hierarchy}(i) \cap A = A$.*

Proof. We give a proof sketch using induction. For the base case, choose any $i \in I$ ($A = \varnothing$). The induction hypothesis is that if a recursive call to $\textsc{Hierarchy}$ contains all ancestors of a parent, then the algorithm will find all parents and their ancestors. Any parent must appear in $I \cup O$, there must be a rule that matches it, and the predicate $\textsc{match}$ must hold by definition. □

4 Nvis: Design and Case Study

In this section, we detail the design of `nvis`, our tool to visualize hierarchical and overlapping intervals. The back-end processing server is scripted with Python's asynchronous HTTP package [2]. We use HTML and D3.JS to render front-end visualizations, and connect the client-server with Python's SocketIO package [1]. The code is publicly available on GitHub [13].

We present an overview of our tool in Figure 2. The data visualized in the figure is from the case study presented below. Each interval identifier is also uniquely color-coded to clearly differentiate the intervals. The shade of the color in each segment is dependent on the number of corresponding intervals in the segment; the darker the shade of the interval segment, the more times an interval occurred within that segment.

To illustrate the practical application of `nvis`, we visualize the logs generated by a Mitsubishi heat pump obtained from our industrial partner Cedar Heat.

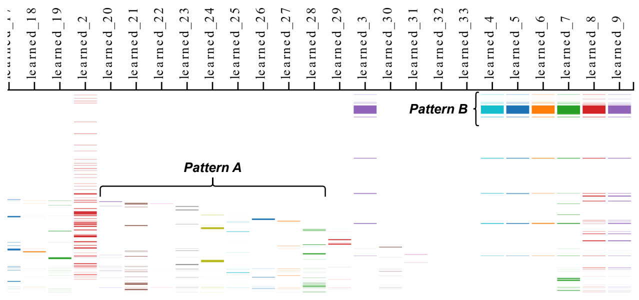

Fig. 2: Overview of `nvis`, our tool to visualize temporal intervals.

These logs contain key-value pairs of system-related information such as run time and baud rate, as well as physical quantities measured by sensors, e.g., temperature and power consumption. Each monitored value is associated with a timestamp, forming a time series. To prepare this case study, we first mapped the time series to an event trace through a discretization step. Then, we used the rule mining feature of `nfer` [6] to extract temporal relations between the events in the trace, and applied the rules to the trace, again using `nfer`, to obtain the intervals. Because they are mined, each rule is labeled with a generic identifier in the format 'learned_id', where 'id' is an incremental counter. This example demonstrates a case where no context is provided with the rules as `nfer` and similar rule mining tools adopt an abstract approach to rule mining. Although this represents an extreme case, `nvis` remains useful when rules are defined manually and carry meaningful identifiers.

Pattern A in Figure 2 is an example of intervals appearing successively, possibly indicating causal relationships between the events. In *Pattern B*, however, multiple rules apply simultaneously, revealing concurrent system behavior. This visual representation facilitates an intuitive understanding of the underlying system behavior by allowing users to easily observe overlapping activities and sequential dependencies.

5 Conclusion

We presented a method to visualize overlapping intervals from `nfer`, allowing pattern analysis in a human-interpretable setting. The method buckets intervals into time segments, each with its own count of occurrence. We proposed `nvis`, a visualization tool that depicts unique colors for each interval, and the shade of each color is based on the count of intervals occurring in each segment. This allows visualization of overlapping intervals. We also proposed an algorithm to recursively find interval hierarchies and extract valid parents producing a specific interval. Finally, we demonstrated `nvis` on an industrial case study, visualizing concurrent and sequential intervals and revealing informative patterns.

References

1. Python socketio, https://python-socketio.readthedocs.io/en/stable/, accessed: 2024
2. Welcome to aiohttp - aiohttp 3.11.11 documentation, https://docs.aiohttp.org/en/stable/, accessed: 2024
3. Allen, J.F.: Maintaining knowledge about temporal intervals. Communications of the ACM **26**(11), 832–843 (1983)
4. Allen, J.F.: Towards a general theory of action and time. Artificial intelligence **23**(2), 123–154 (1984)
5. Beschastnikh, I., Wang, P., Brun, Y., Ernst, M.D.: Debugging distributed systems: Challenges and options for validation and debugging. Communications of the ACM **59**(8), 32–37 (Aug 2016)
6. Kauffman, S., Fischmeister, S.: Mining temporal intervals from real-time system traces. In: International Workshop on Software Mining (SoftwareMining'17). pp. 1–8. IEEE (2017). https://doi.org/10.1109/SOFTWAREMINING.2017.8100847
7. Kauffman, S., Guldstrand Larsen, K., Zimmermann, M.: The complexity of data-free nfer. In: International Conference on Runtime Verification (RV'24). pp. 1–16. Springer (10 2024). https://doi.org/10.48550/arXiv.2407.03155
8. Kauffman, S., Havelund, K., Joshi, R.: nfer–a notation and system for inferring event stream abstractions. In: International Conference on Runtime Verification (RV'16). LNCS, vol. 10012, pp. 235–250. Springer (2016). https://doi.org/10.1007/978-3-319-46982-9_15
9. Kauffman, S., Havelund, K., Joshi, R., Fischmeister, S.: Inferring event stream abstractions. Formal Methods in System Design **53**, 54–82 (2018). https://doi.org/10.1007/s10703-018-0317-z
10. Kauffman, S., Joshi, R., Havelund, K.: Towards a logic for inferring properties of event streams. In: International Symposium on Leveraging Applications of Formal Methods (ISoLA'16). LNCS, vol. 9953, pp. 394–399. Springer (2016). https://doi.org/10.1007/978-3-319-47169-3_31
11. Kauffman, S., Zimmermann, M.: The complexity of evaluating nfer. In: International Symposium on Theoretical Aspects of Software Engineering (TASE'22). LNCS, vol. 13299, pp. 388–405. Springer (07 2022). https://doi.org/10.1007/978-3-031-10363-6_26
12. Kauffman, S., Zimmermann, M.: The complexity of evaluating nfer. Science of Computer Programming **231** (2024). https://doi.org/10.1016/j.scico.2023.103012
13. Sadman, N., Kianersi, N., Kauffman, S.: Github, https://github.com/Nafiz95/Nvis, accessed: 2024
14. Zamfirov, F., Dams, D., Seraj, M., Serebrenik, A.: Encoding domain knowledge in log analysis. In: 40th IEEE International Conference on Software Maintenance and Evolution. pp. 224–236. IEEE (Oct 2024). https://doi.org/10.1109/ICSME58944.2024.00030