Log Analysis and System Monitoring with nfer

Sean Kauffman[1]

Aalborg University, Denmark

Abstract

`Nfer` is a tool that implements the eponymous language for log analysis and monitoring. Users write rules to calculate new information from an event stream such as a program log either offline or online. In addition to a command-line program, `nfer` exposes interfaces in Python and R and can generate monitors for embedded systems. `Nfer` is designed to be fast and has been repeatedly demonstrated to outperform similar tools.

Keywords: Log Analysis, Runtime Verification, Trace Abstraction

Metadata

Nr.	Code metadata description	Please fill in this column
C1	Current code version	v1.8
C2	Permanent link to code/repository used for this code version	`https://nfer.io`
C3	Permanent link to Reproducible Capsule	`https://doi.org/10.24433/CO.0237670.v1`
C4	Legal Code License	GPLv3
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	C, Python, R, git, make
C7	Compilation requirements, operating environments & dependencies	xxd, flex, and bison
C8	If available Link to developer documentation/manual	`http://nfer.io`
C9	Support email for questions	seank@cs.aau.dk

Table 1: Code metadata (mandatory)

[1]Funded by the ERC (LASSO) and the Villum Foundation (S4OS).

1. Introduction

`Nfer` is a tool for offline and online log analysis. Users write rules in an intuitive syntax to extract information from a trace of timestamped events. The output of `nfer` is a hierarchy of time periods called *intervals*. Intervals consist of an identifier (the type of interval), a begin time, an end time, and arbitrary data accessed by string lookups.

The `nfer` tool is an open-source implementation of the eponymous language, developed in collaboration between researchers from the the National Aeronautics and Space Administration (NASA) Jet Propulsion Laboratory (JPL) and the University of Waterloo [1, 2, 3]. It was designed to help operators of the Mars Science Laboratory (MSL) (Curiosity rover) and other spacecraft better understand these remote systems. `Nfer` creates abstractions of event streams, meaning the information produced by applying `nfer` rules is easier to comprehend than directly accessing the event stream. `Nfer` abstractions form a hierarchy of temporal intervals: they can be built up from smaller pieces to find meaningful information. Applying an `nfer` specification can also be understood as adding prior knowledge to a trace, transforming it into an easier to comprehend representation.

`Nfer` is especially well suited for analyzing logs from concurrent processes where order is not well defined. Consider the following example. A short section of a log file is shown in Figure 1 with output from system calls. Each line in the log has a name, a timestamp, and some data associated. To debug a problem using this log, a typical workflow might involve restructuring the data to sort by timestamp, filtering the data by pid, and then manually searching the result for anything concerning. If such a task is performed regularly, an engineer might write a script to do this. `Nfer` allows a user to write rules to perform these tasks easily and without ad hoc solutions that are difficult to maintain. In the example, the rule finds a `lock_irqsave` event during the interval between `mutex_lock` and `mutex_unlock` events with the same `pid`. The result is the generation of an interval named `locksave` over the period when the mutex was locked with the saved `irq` as its data.

Originally developed to inspect spacecraft telemetry, `nfer` was applied to warning analysis for MSL [2]. `Nfer` was later used to search for errors in embedded system call log with faults from ionizing radiation [4, 3]. `Nfer` has also been integrated into the Palisade framework for distributed embedded system anomaly detection and used to find gear change anomalies in an autonomous vehicle [5].

Other work has been inspired by `nfer`, as well. An algorithm for mining response properties from real-time embedded system logs was developed using `nfer` and is implemented in the tool [6]. One work found that it was

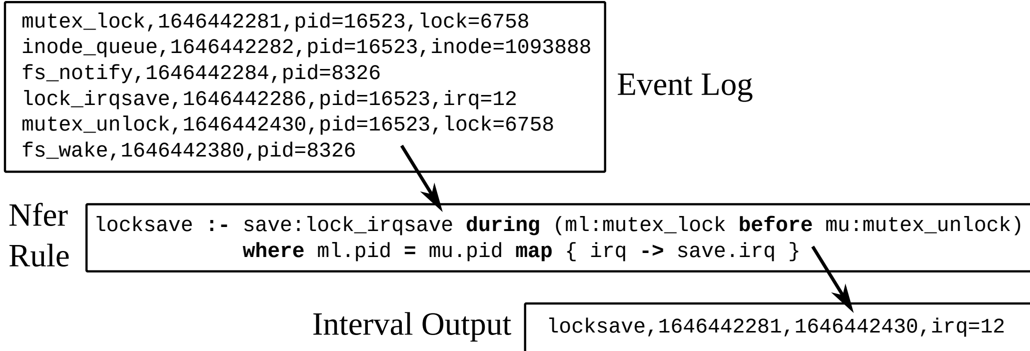

Figure 1: An example of extracting information from a log using `nfer`

possible to outperform Scala `nfer` (see Sec. 3) using six times the number of lines of Cobra, a C-like imperative language for source code analysis [7]. Recently, the evaluation complexity of `nfer` was analyzed and found to be PTIME-complete when deployed in its most common configuration [8], although the full formalism is Turing-complete.

2. Language and Tool

`Nfer` applies a list of rules to a sequence of timed events (treated as intervals with zero duration), generating time intervals that carry data. An *inclusive* `nfer` rule has the form η **:-** $\eta_1 \oplus \eta_2$ **where** φ **map** ψ, where η, η_1, and η_2 are identifiers, $\oplus$ is a temporal operator, φ is a data predicate, and ψ is a data function. Given a set of intervals, a rule generates new intervals with the identifier η when there exist intervals in the set with the identifiers η_1 and η_2 that satisfy the temporal condition $\oplus$ and the data condition φ. The generated intervals have timestamps and data determined by applying $\oplus$ and ψ to the matched intervals, respectively. Rules may depend on one another and are applied until reaching a fixed point. *Exclusive* rules are similar, but test for the *non-existence* of one of the intervals.

The rule syntax supported by the `nfer` tool supports many conveniences, some of which are shown in Figure 1. The figure shows how temporal operators may be nested (**before** is nested within **during**), identifiers may be assigned short labels (`save` labels `lock_irqsave`), and φ and ψ functions are specified using a familiar expression syntax.

The `nfer` tool is written in C and is available under the GPLv3 license [9]. The tool exposes four interfaces to apply specifications to traces and calculate intervals: 1. command line, 2. compiled monitor, 3. Python, and 4. R. The tool implements the `nfer` semantics for offline log analysis, but also supports

online event-stream analysis. Online execution requires events to occur in-order for the full `nfer` language, but the requirement is dropped when only using *inclusive* rules [8].

The simplest and most feature-rich way to use `nfer` is on the command line. `Nfer` is available as an executable that reads a specification from a file and accepts events either in a log file for offline processing or on *standard in* for online processing. The command-line interface supports mining response patterns (called **before** rules, in `nfer`) from real-time system traces [6], windowing optimizations [3], and can generate a compiled monitor. Produced intervals are sent to *standard out*.

The fastest way to execute `nfer` is using a compiled monitor. `Nfer` supports building monitors for use in embedded systems. These monitors are generated from a specification and do not use dynamic memory allocation or recursion. Monitor caches must be carefully configured to avoid memory overflows and this can be done semi-automatically using the command-line interface to calculate safe parameters for an input log.

`Nfer` also exposes language Application Programming Interfaces (APIs) in Python and R. The interfaces are designed to permit integration with code written in the host language with `nfer`. The Python API is focused on monitoring execution of Python programs and includes automatic program instrumentation and a web-based graphical display interface. The R API is focused on processing data and supports input and output via R's native data frames via either user supplied specifications or mined rules.

3. Performance Evaluation

`Nfer` is not the only tool designed for log analysis and, in fact, it is not even the only tool that implements the `nfer` language. A prototype tool implementing `nfer` *inclusive* rules was written at JPL in Scala using a different monitoring algorithm [3]. This tool, which we will refer to as *Scala* `nfer`, was not originally available to the public[2] which motivated the C implementation of the same language. Now, however, the C version has many more features than Scala `nfer` and drastically outperforms it.

We performed an experiment[3] to demonstrate the performance difference between the version of `nfer` presented in this work (C `nfer`) and the prototype developed in Scala, the results of which are shown in Figure 3. This experiment uses the specification developed to analyze system calls logs from a QNX system subjected to ionizing radiation we called the LANL case study

[2]It has since been made open-source: `git@github.com:rv-tools/nfer.git`.

[3]Experiment files available at `https://bitbucket.org/seanmk/nfer-bench`.

in [3]. In the figure, the Y-axis represents the (mean) time needed to process a trace and the X-axis shows how many events were present in the trace. All evidence [3, 10] suggests (C) `nfer` is faster than Scala `nfer` for any workload due to the speed of C and `nfer`'s many optimizations.

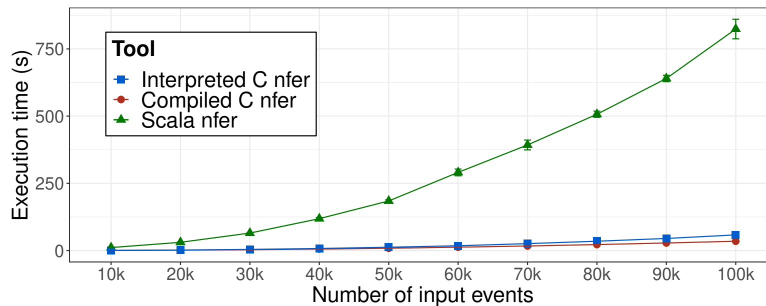

Figure 2: Execution time of the C and Scala implementations in the LANL case study

`Nfer`, with its Turing-complete language, has been demonstrated to be faster than many other comparable tools. Prolog and LogFire took many hours to complete tasks that `nfer` accomplished in seconds [3]. A comparison with the Complex Event Processing system Siddhi showed that `nfer` had 35 times lower latency when monitoring that benchmark [5]. A compiled TeSSLa monitor was very fast and outperformed interpreted `nfer`, but a compiled `nfer` monitor was faster [11]. MonAmi and `DejaVu` needed to run offline to compete with the Scala version of `nfer` from JPL but were unable to match the speed of the C version described here [10].

4. Conclusion

`Nfer` is a tool for log analysis and monitoring that applies rules to time-stamped events to generate a hierarchy of interval abstractions. Work on `nfer` is ongoing, both from the theoretical and practical side. One extension to the theory is to examine the satisfiability of `nfer` rules given a set of possible input identifiers. On the practical side, many improvements are planned including extensions to testing, regular expression matching of strings, and further refinement of the Python API. There is also a plan to include visualization capabilities in every interface (not only Python). Finally, documentation and dissemination continue to be priorities as `nfer` gains visibility with researchers outside its core area.

References

[1] S. Kauffman, R. Joshi, K. Havelund, Towards a logic for inferring properties of event streams, in: Leveraging Applications of Formal

Methods (ISoLA'16), Vol. 9953 of LNCS, Springer, 2016, pp. 394–399. doi:10.1007/978-3-319-47169-3_31.

[2] S. Kauffman, K. Havelund, R. Joshi, nfer–a notation and system for inferring event stream abstractions, in: Runtime Verification (RV'16), Vol. 10012 of LNCS, Springer, 2016, pp. 235–250. doi:10.1007/978-3-319-46982-9_15.

[3] S. Kauffman, K. Havelund, R. Joshi, S. Fischmeister, Inferring event stream abstractions, Formal Methods in System Design 53 (2018) 54–82. doi:10.1007/s10703-018-0317-z.

[4] A. Narayan, S. Kauffman, J. Morgan, G. M. Tchamgoue, Y. Joshi, C. Hobbs, S. Fischmeister, System call logs with natural random faults: Experimental design and application, in: Silicon Errors in Logic – System Effects (SELSE'17), IEEE, 2017.

[5] S. Kauffman, M. Dunne, G. Gracioli, W. Khan, N. Benann, S. Fischmeister, Palisade: A framework for anomaly detection in embedded systems, Journal of Systems Architecture 113 (2021) 101876. doi:10.1016/j.sysarc.2020.101876.

[6] S. Kauffman, S. Fischmeister, Mining temporal intervals from real-time system traces, in: Software Mining (SoftwareMining'17), IEEE, 2017, pp. 1–8. doi:10.1109/SOFTWAREMINING.2017.8100847.

[7] G. J. Holzmann, Comparing Two Methods for Checking Runtime Properties, Springer, 2021, pp. 127–133. doi:10.1007/978-3-030-87348-6_7.

[8] S. Kauffman, M. Zimmermann, The complexity of evaluating nfer, in: Theoretical Aspects of Software Engineering (TASE'22), Vol. 13299 of LNCS, Springer, 2022, pp. 388–405. doi:10.1007/978-3-031-10363-6_26.

[9] S. Kauffman, Website, http://nfer.io/, accessed: Jan 2022 (2022).

[10] K. Havelund, M. Omer, D. Peled, Monitoring first-order interval logic, in: Software Engineering and Formal Methods (SEFM), Springer, 2021, pp. 66–83. doi:10.1007/978-3-030-92124-8_4.

[11] S. Kauffman, nfer – a tool for event stream abstraction, in: Software Engineering and Formal Methods (SEFM'21), Vol. 13085 of LNCS, Springer, 2021, pp. 103–109. doi:10.1007/978-3-030-92124-8_6.