`nfer` – A Tool for Event Stream Abstraction

Sean Kauffman

Aalborg University, Denmark

Abstract. This work describes `nfer`, an open-source tool for event-stream abstraction and processing. `Nfer` implements the Runtime Verification logic of the same name, providing programming interfaces in C, R, and Python. Rules that dictate `nfer`'s behavior can be written in an external Domain-Specific Language (DSL), mined from historical traces, or given using an internal DSL in Python. The tool is designed for efficient online monitoring of event streams and can also operate as an offline tool to process completed logs.

1 Introduction

The exponential increase in the size and complexity of embedded software over time has led to a similar explosion in traces produced by that software [8]. Comprehending and verifying those traces at runtime requires tools with diverse interfaces that can handle large datasets and integrate with existing code.

`Nfer` is a formalism for abstracting and monitoring event streams [15, 13, 14] with an open-source implementation well-suited for a variety of tasks. The `nfer` language is based on Allen's Temporal Logic (ATL) [2] and is designed for expressing relationships between concurrent executions [12]. The implementation is available at http://nfer.io under the GPLv3 license and includes a command-line interpreter, the ability to learn rules from historical traces [11], an embedded monitor compiler, and language integrations with both R and Python.

`Nfer` combines elements of Complex Event Processing (CEP) systems [4, 16, 17], stream-processing frameworks [5, 7], and rule-based logics [3, 9]. Like many of these tools, `nfer` applies rules to event streams to generate new facts either online or offline. However, `nfer` treats time as a first-class citizen and produces temporal intervals carrying data using a rule syntax designed to describe context.

This paper describes the open-source implementation of `nfer`. Section 2 contains a programming guide for the `nfer` language, including a running example. Section 3 describes the `nfer` architecture. Section 4 compares `nfer` to TeSSLa, a popular stream processing tool. The paper concludes in Section 5.

2 Writing nfer Rules

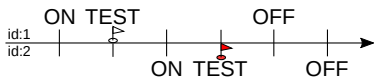
`Nfer`'s external domain-specific language (DSL) is a declarative, rule-based logic for inferring a hierarchy of intervals from an event stream. Rules specify how

new intervals are created from old ones as well as from events. This section uses a running example to illustrate how `nfer` rules are formulated.

Inputs and outputs in `nfer` are temporal intervals of the type $\mathcal{I} \times \mathbb{N} \times \mathbb{N} \times \mathbb{M}$, where $\mathcal{I}$ is the finite set of identifiers (or names), $\mathbb{N}$ is the natural numbers representing begin and end timestamps, and $\mathbb{M}$ is the type of maps from strings to literals. Inputs supplied as events (only one timestamp) are represented interally as *atomic* intervals where the begin and end times are equal. Intervals can have temporal and data relationships, and those relationships define new intervals.

Below is a sequence of six events representing two systems powering on, performing a test, and powering off. In the table on the left, each event is shown as a row in the table with its identifier (name), begin, and end timestamps, as well as two data items: the id of the system and if the test succeeded. On the right of the figure, the same trace is shown on a timeline, with system id:1 above the line and system id:2 below the line. The TEST event with success:false is distinguished by the small shaded flag, where the successful one has a white flag.

Name	**Begin**	**End**	**id**	**success**
ON	10	10	1	
TEST	20	20	1	true
ON	30	30	2	
TEST	40	40	2	false
OFF	50	50	1	
OFF	60	60	2	



We want to capture periods where system runs had test failures. We also know that when two tests occur during a run, one may report a benign failure that should be ignored. We want to flag test failures where no other successful test occurred during the failed test's run. We begin with a simple, flawed rule.

```
OPERATING :– ON before OFF
```

This rule says that when an ON interval is seen **before** an OFF interval, create an interval named OPERATING with a begin time equal to the begin time of ON and an end time equal to the end time of OFF. The words OPERATING, ON, and OFF and all arbitrary and could be the names of any intervals, while the word **before** is a keyword that specifies a temporal relationship.

Unfortunately, this rule does not produce what we intended because `nfer`, by default, only creates *minimal* intervals. A minimal interval is one where no interval with the same name occurs **during** that interval. Only the interval $[30, 50]$ will be reported by this rule, while three intervals will be omitted: $[10, 50]$, $[10, 60]$, and $[30, 60]$. Minimality checking can be disabled to obtain all four intervals.

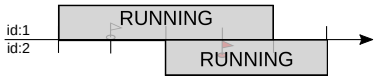
To generate the two intervals we intend, however, we need to apply a **manual constraint**. The **where** keyword specifies a manual constraint that must be satisfied in addition to those of the temporal relation. The constraint is an expression that must evaluate to a Boolean value and may refer to the timestamps and data of the intervals specified in the temporal relation part of the rule. Data and timestamps are specified by separating the interval name to reference and

its datum name with a period. This rule will generate the desired intervals by specifying that the id of the ON and OFF intervals must be equal.

```
RUNNING :– ON before OFF where ON.id = OFF.id map { id –> ON.id }
```

Note that the rule also adds the id of the system to the generated RUNNING intervals. New intervals have empty data maps by default, but data may be specified using the **map** keyword. Map expects a list of keys and associated expressions listed inside curly braces. Map expressions may return any type.

Name	Begin	End	id
RUNNING	10	50	1
RUNNING	30	60	2



Note that the temporal operator (e.g., **before**) typically defines the timestamps of generated intervals. The **begin** and **end** keywords specify expressions that manually override the begin and end timestamps of intervals created by the rule. This allows a rule to specify precisely the interval of interest, for example specifying a period 30 seconds after an event occurs.

Next, we want to identify the system executions where the test succeeded or failed. We can do that with the following rules.

```
TESTING :– t:TEST during r:RUNNING where t.id = r.id map {s –> t.success}
FAILURE :– TESTING where !TESTING.s
```

The first rule produces a TESTING interval when a TEST occurs **during** a RUNNING. The rule uses a different temporal operator (**during**, not **before**). It also uses *labels*, prepended to interval names with colons, to provide shorter handles to reference events in expressions. Labels are required when the temporal operator refers to two intervals with the same name.

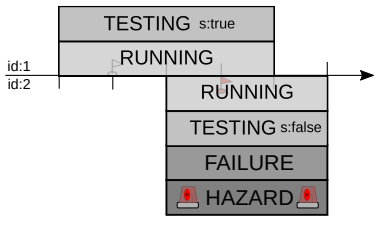
The second rule produces a FAILURE interval when a TESTING interval has its *s* datum set to false. We mapped the *s* datum of TESTING to the *success* datum of TEST in the previous rule. The FAILURE rule has no temporal operator and will match all TESTING intervals.

To complete the specification, we need a rule that identifies when a FAILURE occurs without another TEST succeeding during the same period. `Nfer` supports testing for the absence of intervals using the unless keyword.

```
HAZARD :– FAILURE unless contain TEST where TEST.success
```

This rule matches a FAILURE interval when no TEST succeeded in the same period. In our example, the successful test occurred before the FAILURE interval began, so a HAZARD is produced.

Name	Begin	End	id	s
RUNNING	10	50	1	
TESTING	10	50		true
RUNNING	30	60	2	
TESTING	30	60		false
FAILURE	30	60		
HAZARD	30	60		



`Nfer` also supports mining rules from historical traces [11]. These can be useful as either the basis of a new specification or to check the accuracy of human-written rules. `Nfer`'s mining algorithm works best when traces are generated by highly periodic systems, such as an embedded system running a real-time scheduler. The mining algorithm generates only **before** relations from events and does not yet support learning a hierarchy of rules.

To mine rules from the example in this section, we must extract the events from only one of the systems. Passing such a trace to the `nfer`'s mining algorithm yields the following learned rules that describe **before** relations that hold between the three input event names.

```
learned_0 :- ON before TEST
learned_1 :- ON before OFF
learned_2 :- TEST before OFF
```

3 Nfer's Architecture

`Nfer` is designed for low-latency operation with minimal memory use. Every interface to `nfer` uses the same core components, written in C, with minimal external dependencies. Each interface then combines the `nfer` core with capabilities specific to its intended use-cases.

The `nfer` architecture is shown in Figure 1. In the figure, each interface is shown as a separate box, with its sub-components shown as internal labeled boxes. Every interface shares the `nfer` core, made up of the optimized data-structures and algorithms for executing the `nfer` monitoring algorithm.

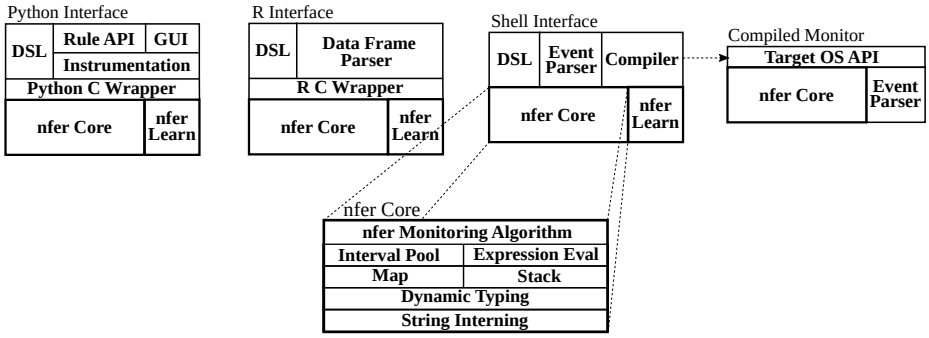


Fig. 1: `nfer` architecture, with labeled components for each interface

The `nfer` core consists of several custom data-structures designed to work together for efficiency. In `nfer`, all strings are interned and subsequently referenced by a zero-based integer identifier. This permits the map implementation to store values in an array indexed by string ids, making map lookups a simple

memory offset calculation. Interning means string comparisons become integer comparisons and memory use for strings is reduced. Expression evaluation for `nfer`'s DSL is performed using a reverse-polish algorithm and a custom stack implementation. The `nfer` core excludes all recursion to facilitate embedded system operation including in expression evaluation, the interval pool's merge-sort implementation, and the `nfer` monitoring algorithm [14].

`Nfer`'s language bindings in R and Python are implemented as native language wrappers around the compiled `nfer` core. The R library is designed for data processing and integrates closely with R's native data structures. In R, `nfer` rules may be loaded from file or mined and then applied to a data frame of events to produce a data frame of intervals. The Python module (available via PyPi as NferModule) includes instrumentation code for Python programs, a native Python rule DSL, and a Graphical User Interface (GUI) for visualizing intervals at runtime. By using the compiled `nfer` core for interval processing, both tools are much faster than if the language was implemented natively.

An `nfer` specification may be compiled to a C program using the shell interface. Compiled monitors include the `nfer` core but use only static memory allocation, with the size of components set via compile-time configuration. Static memory allocation reduces the time needed to handle complex specifications but results in higher memory use, since sufficient space must be configured for any expected workloads. `Nfer` can suggest memory settings given a specification and trace. Compiled monitors are designed for embedded use and have been integrated with Linux and ERIKA Enterprise.

4 Comparison with TeSSLa

TeSSLa is a stream-processing language and tool designed for efficiently checking logical properties and computing temporal metrics from a trace [5]. Like `nfer`, TeSSLa can compute rich abstractions of a trace online using a formal language specification. Embedded TeSSLa [6], which runs on reconfigurable hardware, cannot use the dynamic data structures necessary to emulate `nfer`.

One important difference between `nfer` and TeSSLa is the simplicity of a specification to produce temporal intervals. As a general stream-programming framework, TeSSLa is capable of producing intervals but doing so requires more complex rules. For example, to implement the four rules from Section 2 in TeSSLa requires a specification of at least 34 lines.

We conducted an experiment using the example rules from Section 2. The specification is simple, but represents a typical use for `nfer` and is complex enough to demonstrate `nfer`'s speed. The TeSSLa specification we used and related documentation is available in the `doc/tessla` directory of `nfer`'s source code repository [1]. We ran two configurations of `nfer`, one interpreting the specification through the shell interface and one using a compiled monitor. We compared these with a compiled TeSSLa 1.2.2 monitor running on Oracle JRE 11.0.12. We generated system logs with varying numbers of operations where

each operation resulted in three events. We ran each tool on each log ten times, allocating one core of an AMD EPYC 7642 running at 1.5 GHz.

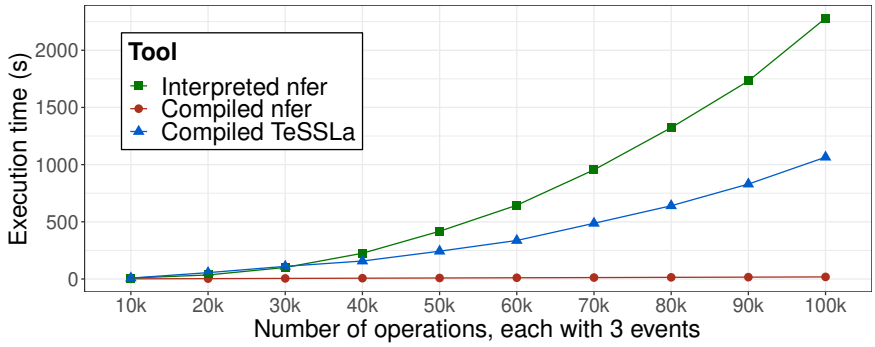


Fig. 2: Execution time used in the experiment

Figure 2 shows the result of the comparison, where lower execution times are better. In the figure, each mean execution time is shown as a point and the standard deviation is omitted as the error bars are too small to be visible. Although the interpreted version of `nfer` takes around twice the time of the compiled TeSSLa monitor, the compiled `nfer` monitor is much faster. We do not report on memory use since TeSSLa uses the Java Virtual Machine (JVM), making memory utilization difficult to separate from memory allocation.

There are other Runtime Verification (RV) tools that may be interesting to compare to `nfer`. In [10], we compared the latency of an `nfer` integration into a Python framework with the CEP system, Siddhi [17] and found `nfer` to be over 35 times faster. Some stream RV tools, such as RTLola [7], do not support dynamic data structures and, as such, cannot emulate the full `nfer` language.

5 Conclusion

The open-source implementation of the `nfer` logic described in this paper is distributed via the GPLv3 license. The tool is easy to install and use in the Unix command-line, R, and Python. It provides efficient online monitoring of event streams and offline analysis of timed data. `Nfer` may be used on embedded systems with no dynamic memory, and it can be used to visualize Python program execution in real-time.

The `nfer` project continues to evolve. Future work includes support for new data formats, MISRA-C compliance for compiled monitors, multi-threading support, and performance improvements. In-progress work will characterize the complexity of different `nfer` language subsets. Check http://nfer.io for updates.

References

1. Nfer web site, http://nfer.io/, accessed: 2021-10-11
2. Allen, J.F.: Maintaining knowledge about temporal intervals. Communications of the ACM 26(11), 832–843 (1983)
3. Barringer, H., Havelund, K.: TraceContract: A Scala DSL for trace analysis. In: Formal Methods (FM'11). LNCS, vol. 6664, pp. 57–72. Springer (2011)
4. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: A scalable continuous query system for internet databases. In: International Conference on Management of Data (ACM SIGMOD'00). pp. 379–390. ACM (2000)
5. Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., Thoma, D.: TeSSLa: Temporal stream-based specification language. In: Formal Methods: Foundations and Applications. LNCS, vol. 11254, pp. 144–162. Springer (2018)
6. Convent, L., Hungerecker, S., Scheffel, T., Schmitz, M., Thoma, D., Weiss, A.: Hardware-based runtime verification with embedded tracing units and stream processing. In: Runtime Verification (RV'18). LNCS, vol. 11237, pp. 43–63. Springer (2018)
7. Faymonville, P., Finkbeiner, B., Schwenger, M., Torfah, H.: Real-time stream-based monitoring (2019)
8. van Genuchten, M., Hatton, L.: Compound annual growth rate for software. IEEE Software 29(4), 19–21 (2012)
9. Havelund, K.: Rule-based runtime verification revisited. International Journal on Software Tools for Technology Transfer 17(2), 143–170 (4 2015)
10. Kauffman, S., Dunne, M., Gracioli, G., Khan, W., Benann, N., Fischmeister, S.: Palisade: A framework for anomaly detection in embedded systems. Journal of Systems Architecture 113, 101876 (2021)
11. 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)
12. Kauffman, S., Fischmeister, S.: Event stream abstraction using nfer: Demo abstract. In: International Conference on Cyber-Physical Systems (ICCPS'19). pp. 332–333. ACM Press (2019)
13. 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)
14. Kauffman, S., Havelund, K., Joshi, R., Fischmeister, S.: Inferring event stream abstractions. Formal Methods in System Design 53, 54–82 (2018)
15. 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)
16. Luckham, D.: The power of events: An introduction to complex event processing in distributed enterprise systems. In: Rule Representation, Interchange and Reasoning on the Web. LNCS, vol. 5321. Springer (2008)
17. Suhothayan, S., Gajasinghe, K., Loku Narangoda, I., Chaturanga, S., Perera, S., Nanayakkara, V.: Siddhi: A second look at complex event processing architectures. In: Workshop on Gateway Computing Environments (GCE'11). pp. 43–50. ACM (2011)