Explainable Queries over Event Logs

Sylvain Hallé

Laboratoire d'informatique formelle
Université du Québec à Chicoutimi, Canada

***Abstract*—Added value can be extracted from event logs generated by business processes in various ways. However, although complex computations can be performed over event logs, the result of such computations is often difficult to explain; in particular, it is hard to determine what parts of an input log actually matters in the production of that result. This paper describes a framework to provide explainable results for queries executed over sequences of events, where individual output values can be precisely traced back to the data elements of the log that contribute to (i.e. "explain") the result. This framework has been implemented into the BeepBeep event processing engine and empirically evaluated on various queries.**

I. Introduction

Various kinds of information systems generate streams in the form of sequences of data elements called *event logs*. Sources of event logs are diverse: business process management engines, web servers, sensor networks, instrumented pieces of generic software can all be instructed to record information about their execution to a persistent storage medium. Added value can be extracted from event logs generated by these systems in various ways. Logs can be checked for compliance violations of best practices, adherence to predetermined sequences of events, detect deviations of some data point from a specified value, or be used to calculate various quality metrics. This process can take place after the system has completed its execution (*offline* processing), or compute its results on-the-fly as the events from the source are ingested (*streaming* processing). These two modes of operation are often grouped under the generic term "event stream processing".

Over the past decade, event stream processing systems have seen widespread use, with the advent of solutions such as Amazon Kinesis[1], Apache Flink[2], Siddhi [22] and Esper[3]. These systems provide rich processing capabilities, making it possible to evaluate complex queries over event logs. However, although intricate computations can be performed over these sources of data, the result of such computations is often difficult to explain. For example, a Flink pipeline can calculate some quality metric over instances of a process, and check that it always lies over some given threshold; however, if the result is false, how can one identify the source of the error?

Developers of information systems in all disciplines are facing increasing pressure to come up with mechanisms to describe how a specific result is obtained –a concept called *explainability*. Hence, if a system fails to verify a given property, a counter-example is generally sought after as a means of understanding the source of the problem. This pressure often comes from regulations imposing constraints on the traceability of data processing, such as GDPR and BCBS. Yet, for most of the aforementioned engines, it is hard to determine what parts of an input log actually matter in the production of a given result. A user is typically left with the manual task of querying the log in various ways in order to investigate the reason for a surprising or irregular output result.

In Section II, we shall see that various technologies and frameworks have been developed over the years in order to provide a form of "lineage" or "provenance" information about the output of some computer system. However, none of these systems consider the special problem of explainability for event stream processing; in contrast, existing event stream processing systems provide very little in the way of lineage and explainability —the closest notion being that of finding *alignments* between log events and a business process specification [2].

In this paper, we formally describe a generic framework that can provide explanations for various kinds of queries over event streams, when such queries are expressed as a composition of elementary computation units. First, Section III shall introduce the basic concepts behind event stream processing, and provide a few examples of simple queries that can be run on event logs. Then, Section IV formally defines data lineage in the context of event streams; it takes advantage of the fact that queries are done by composing basic computation units together into event pipelines. Therefore, in order to obtain end-to-end explainability, one is only required to define simple input/output relationships for each of these units separately.

These formal concepts have been concretely implemented into an existing event stream processing library, called BeepBeep [14], which has been extended such that the output produced by a query can be precisely traced back to the individual data elements of the log that contribute to (i.e. "explain") the result. This makes BeepBeep one of the first stream processing engines offering off-the-shelf explainability for arbitrary queries constructed with the building blocks it provides. The implementation is described and tested in Section V, where the impact of the use of provenance on space and time resources is measured experimentally. These results show that, provided a user accepts some performance trade-off, the library can provide articulate and intuitive results.

[1]https://aws.amazon.com/kinesis
[2]https://flink.apache.org
[3]https://espertech.com

II. Related Work

Taken in a broad sense, we call "data lineage" any activity that attempts to link the result of a computation (its outputs) to elements that contribute to this result (its inputs). Depending on the field of study, variants on the notion of lineage have been given different names.

A large amount of work on lineage has been done in the field of databases, where this notion is often called *provenance*. We can distinguish between three types of provenance. The first type is called *why-provenance* and has been formalized by Cui *et al.* [6]. To each tuple t in the output of a (relational) query, why-provenance associates a set of tuples present in the input of the query; the meaning of this set is to collect all the input data that helped to "produce" t. *How-provenance*, as its name implies, keeps track not only of what input tuples contribute to the input, but also in which way these tuples have been combined to form the result [10]. Finally, *where-provenance* describes where a piece of data is copied from [3]. It is typically expressed at a finer level of granularity, by allowing to link individual values inside an output tuple to individual values of one or more input tuple. One possible way of doing this is through a technique called annotation-propagation, where each part of the input is given symbolic "annotations", which are then percolated all the way to the output [1].

There exist various implementations of provenance-aware database systems. Where-provenance has been implemented into Polygen [26], DBNotes [5], Mondrian [9], MXQL [25] and Orchestra [16]. The Spider system performs a slightly different task, by showing to a user the "route" from input to output that is being taken by data when a specific database query is executed [4]. The foundations for all these systems are relational databases, where sets of tuples are manipulated by operators from relational algebra, or extensions of SQL.

Outside the field of databases, the W3C has standardized a data model for provenance information called Prov [11]. The standard includes an ontology that defines multiple provenance relationships, such as "was derived from", "was revision of". A templating system for Prov data has been proposed by Moreau *et al.* [17]; we shall see that it superficially resembles the graph of processors produced in the present work. However, prov-template assumes that, for a given processing task, this graph has the same structure for every input, and only differs in the actual bindings given to its various elements. On the contrary, we shall see that in BeepBeep, some processor chains produce graphs whose structure highly depends on the input given to the pipeline. Moreover, the approach assumes these templates as given, while our proposed work dynamically generates these graphs from a processor chain and an input stream at runtime.

On the stream processing front, few solutions have been developed to provide explanations for queries. Spline [21] is a system that works on top of Apache Spark and attempts to recover lineage information by instrumenting processing jobs; "lineage", in this case, means the topological organization of jobs and data sources that are being used. However, this system does not work at the individual event level, and hence cannot be used to explain the value of a precise output event produced by a Spark pipeline. Apache Atlas[4] provides similar coarse-grained functionaities for jobs running on Hadoop. To the best of our knowledge, no existing work focuses on fine-grained explainability of individual events in a stream processing pipeline.

The notion of explanation shares similarities with the well-known concept of *alignment* in business processes [2], [24]. An alignment is a mapping between the execution of transitions in the process model and the activities observed in a trace from a given event log. Although alignments can be used to investigate compliance violations, this notion is different from the explanations defined in this paper, which consist of finding a subset of the log that suffices to cause a failure. Moreover, alignments are defined only on process models expressed as Petri nets, whereas our notion of explainability extends to arbitrary computations, such as temporal logic and numerical calculations. Explanation can also be seen as a particular case of process querying on logs [8], [19].

III. Event Log Query Processing with BeepBeep

In this section, we shall first describe basic concepts of event log processing, as implemented by the BeepBeep event stream query engine. BeepBeep is a Java library that allows users to easily ingest and transform event streams of various types; the library is free and open source[5]. A detailed description of BeepBeep is out of the scope of this paper, due to space restrictions. For further details, the reader is referred to a complete textbook describing the system [14].

A. Functions and Processors

BeepBeep is organized around the concept of *processors*. In a nutshell, a processor is a basic unit of computation that receives one or more event streams as its input, and produces one or more event streams as its output. A processor produces its output in a streaming fashion: it does not wait to read its entire input trace before starting to produce output events. However, a processor can require more than one input event to create an output event, and hence may not always output something when given an input.

BeepBeep's core library provides a handful of generic processor objects performing basic tasks over traces; they can be represented graphically as boxes with input/output "pipes", as is summarized in Figure 1.

A first way to create a processor is by lifting any function f into processor. This is done by applying f successively to each input event (or n-tuple of input events, for functions that have n arguments), producing the output events. A variant of this process is the `Cumulate` processor, which, as its name implies, accumulates input values according to some function; for example, providing it with the `Addition` function will cause it to output the cumulative sum of all events received so far. Note that `Cumulate` also works with non-numerical events.

[4]https://atlas.apache.org

[5]https://liflab.github.io/beepbeep-3

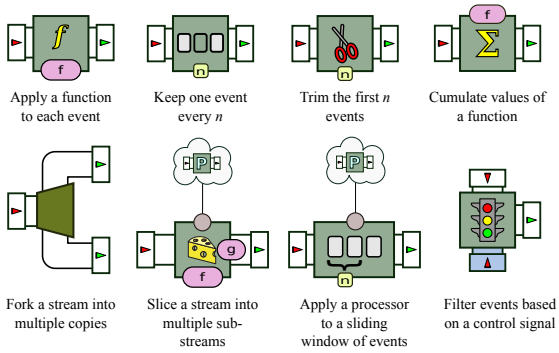

Figure 1: Pictograms for the basic BeepBeep procssors.

A few processors can be used to alter the sequence of events received. The `CountDecimate` processor returns every n-th input event and discards the others. Another operation that can be applied to a trace is trimming its output. Given a trace, the `Trim` processor returns the trace starting at its n-th input event. Events can also be discarded from a trace based on a condition. The `Filter` processor takes two input streams; the events are let through on its first input stream, if the event at the matching position of the second stream is the value `true` ($\top$); otherwise, no output is produced.

Another important functionality of event stream processing is the application of some computation over a window of events. If φ is an arbitrary processor, the `Window` processor of φ of width n sends the first n events (i.e. events numbered 0 to $n-1$) to an instance of φ, which is then queried for its n-th output event. The processor also sends events 1 to n to a second instance of φ, which is then also queried for its n-th output event, and so on. The resulting trace is indeed the evaluation of φ on a sliding window of n successive events. Any processor can be encased in a sliding window, provided it outputs at least n events when given n inputs.

In the case of business processes, a log can contain interleaved sequences of events for multiple process instances. The sub-sequence of events belonging to the same process instance is called a *slice*; applying a separate processing to each such sub-sequence will be called *slicing*. To this end, BeepBeep provides a processor called `Slice`, which is one of the most complex of the core library. It uses a function f to separate an input stream into several sub-streams. Each of these sub-streams is sent to a different instance of some processor P, and the output of each copy is aggregated by another function g.

B. Pipes and Palettes

In order to create complex computations, processors can be composed (or "piped") together, by letting the output of one processor be the input of another. An important characteristic of BeepBeep is that this piping is possible as long as the type of the first processor's output matches the second processor's input type. Such pipes can easily be created by using Java as the glue code.

If chains of basic processors are not sufficient to accomplish the desired computation, BeepBeep makes it possible to extend its core with various packages of domain-specific processors and functions, called *palettes*. The main advantage of the palette system is its modularity: apart from a small core of common objects, a user is required to load only the palettes that are relevant to the computing task at hand. BeepBeep's "standard library" offers more than a dozen such palettes; we briefly describe in the following those of particular interest in the context of business process logs.

1) Finite-State Machines: A frequent use of stream processing is to check whether the events inside a log follow a specific sequence, and trigger a warning as soon as a violation is observed. Specifying the allowed event sequences can be done, among other things, by means of a finite-state automaton. BeepBeep's `Fsm` palette allows users to create *Moore machines*, a special case of automaton where each state is associated to an output symbol. This Moore machine allows its transitions to be guarded by arbitrary functions; hence it can operate on traces of events of any type.

By associating states of the FSM to, e.g. Boolean values, a Moore machine can act as a *monitor*: when fed events from a log, it can be instructed to output the value `true` (or no value at all) as long as the input sequence is a valid path, and return `false` when the last event received does not correspond to an acceptable transition in the current state of the automaton. In such a way, Moore machines can be used to verify compliance of a sequence of events to a specification, such as a model of a business process.

2) Linear Temporal Logic: Similar to the `Fsm` palette, the `Ltl` palette makes it possible for users to write conditions on event sequences using Linear Temporal Logic (LTL) [18]. We recall that LTL, in addition the usual Boolean connectives, provides four temporal *operators* that apply on an arbitrary formula φ. The temporal operator $\mathbf{G}$ means "globally": the formula $\mathbf{G}\,\varphi$ means that formula φ is true in every event of the trace. The operator $\mathbf{F}$ means "eventually"; the formula $\mathbf{F}\,\varphi$ is true if φ holds for some future event of the trace. The operator $\mathbf{X}$ means "next"; it is true whenever φ holds in the next event of the trace. Finally, the $\mathbf{U}$ operator means "until"; the formula $\varphi\,\mathbf{U}\,\psi$ is true if φ holds for all events until some event satisfies ψ.

Each of these temporal operators is implemented as a `Processor` object, and chaining such processors appropriately allows users to create pipes that can be used to evaluate any arbitrary LTL formula. Each LTL processor for an LTL formula φ applies the following semantics: the i-th output event is the verdict produced by a monitor evaluating the input trace starting at event i.

Typically, temporal processors produce bursts of output events for multiple inputs at the same time, once a specific value (true or false) is received in the input stream. Consider the case of operator $\mathbf{G}\,\varphi$. The processor for this operator takes as input a stream of Boolean values, corresponding to the evaluation of φ on each input event. Given the input stream $\top, \top, \bot, \top$, the processor will produce the output stream $\bot, \bot, \bot$: indeed, the property $\mathbf{G}\,\varphi$ is definitely false for the trace prefixes starting

in each of the first three input events. However, those three outputs can only be produced once input event $\perp$ at position 3 has been received. Similarly, a definite verdict cannot yet be computed for the input prefix starting at event 4. A similar reasoning applies to the remaining operators.

IV. An Explanation Framework for Stream Queries

We present in this section a definition of lineage for elementary computation units taking event streams as their input, and producing event streams as their output. Although these concepts closely match the principles behind the BeepBeep event stream library, they can easily be generalized to any type of computation organized into a form of pipeline.

More precisely, in the present context "lineage" will correspond to the association that can be established between a specific output event produced by a processor, and the input events that are involved in the production of this output. This is where the concept of composition can be put to good use. Since complex chains are obtained by piping basic units into graphs, it suffices to define input/output associations for each unit separately. By virtue of composition, it will then be possible to retrace output events all the way up to the original inputs of a pipe, by simply following the chain of associations from each processor to the next upstream unit.

The end result of such tracking is a directed acyclic graph (DAG) which, from a given output event, follows the input/output associations in the chain all the way up to the original inputs. As we shall see, the relationship between the input and the output can be many-to-many; this is why the generated structure is generally a graph, and not a linear chain of nodes.

A. The Derivation Operator

We shall designate by $\overline{a} = a_1, \ldots, a_n$ a finite sequence of *events* taken over a domain A; the set A^* represents all sequences of events from A. The length of a sequence $\overline{a}$ is noted $|\overline{a}|$, and the i-th element of $\overline{e}$ is noted $\overline{a}[i]$. The special notation $\overline{a}[-i]$ will read events from the end; hence $\overline{a}[-1]$ represents the last event of $\overline{a}$. Formally, a processor is a transducer $\pi : (A^*)^m \to (A^*)^n$ that maps m input streams into n output streams. We expect processors to be monotonic: given input sequences $\overline{a}_1, \ldots \overline{a}_m$ and $\overline{a}'_1, \ldots \overline{a}'_m$, if $\overline{a}'_i$ is a prefix of $\overline{a}_i$ for all i, then $\pi(\overline{a}'_1, \ldots \overline{a}'_m)$ is a prefix of $\pi(\overline{a}_1, \ldots \overline{a}_m)$. In other words, a processor cannot "rewrite the past" by changing events that have already been output. An event *front* is the m-uple of events at the same position in all m input or output streams. A formal definition of all processors is out of the scope of this paper and has already been covered [13]. For the purpose of the discussion, it suffices to know that all processors presented so far admit a straightforward definition following these conventions.

In order to define explainability for processors, we introduce an additional relation called the *derivation operator*. Given a processor π, the derivation of π, noted $\partial\pi$, is another processor that receives the same input events as π, but produces a sequence of *lineage functions* $\ell : \mathbb{N}^2 \to 2^{\mathbb{N}^2}$. The intuition behind each distinct function ℓ is that, if $f(x, y) = \{(x'_1, y'_1), \ldots, (x'_k, y'_k)\}$, the x-th event of the y-th output of π is associated to the x'_1-th event of the y'_1-th input of π, and the x'_2-th event of the y'_2-th input of π, and so on. In other words, ℓ maps each output event to zero or more input events.

Note that $\partial\pi$ is itself a processor; hence this sequence of lineage functions may depend on the actual input events that are ingested. However, we expect this sequence to be well-defined: if ℓ and ℓ' are two consecutive functions produced by $\partial\pi$, $\ell(x, y) = S$ implies $\ell'(x, y) = S$. This means that successive functions preserve all input/output associations that have already been defined, and may only add new ones. Given input sequences $\overline{a}_1, \ldots, \overline{a}_m$, we shall denote by $\partial\pi/\partial\overline{a}_1 \ldots \partial\overline{a}_m$ the last function ℓ produced by $\partial\pi$ when given the input $(\overline{a}_1, \ldots, \overline{a}_m)$. We shall also abuse notation, and extend the notion of derivation to a function $f : A^m \to A^n$; the derivation $\partial f/\partial a_1 \ldots \partial a_m$ is the lineage function that associates the outputs of f to its inputs, when evaluated with the arguments $a_1, \ldots, a_m$. In such a case, the first element of each tuple will always be 1.

Under this framework, in order to provide explainability for processors, it suffices to define the behavior of the derivation operator ∂ for each processor π. However, rather than providing a definition for $\partial\pi$ directly, it is typically easier to define it implicitly through the function ℓ resulting from $\partial\pi/\partial\overline{a}_1 \ldots \partial\overline{a}_m$. Assuming that ∂ is well-defined, this ensures a unique definition for $\partial\pi$.

Note that our formal definitions make no assumption over *how* outputs are associated to inputs. This means that multiple derivation operators could be defined, each carrying a different intuitive meaning. However, the present work is motivated by the need to "explain" outputs of a stream query; therefore, we shall present definitions for ∂ geared towards the identification of input events whose presence is necessary and sufficient to produce a given output event.

This can be illustrated by a simple example. With the function $f(x, y) = x + y$, one can see that any value f can produce always depends on its two operands, x and y. However, the case of function $g(x, y) = xy$ is different; typically, the knowledge of both x and y is required to explain the output value, but not always: when $x = 0$ and $y = 1$, the fact that $g(x, y) = 0$ can be explained solely by the value of x. A similar argument could be done with Boolean connectives.

B. Derivation for Core Processors

Most of BeepBeep's core processors have relatively straightforward association rules. The `CountDecimate` processor, whose task is to keep every n-th event and discard the others, registers an association between input event at position i and output event at position ni. Hence, $\partial\pi/\partial\overline{a} \triangleq (x, 1) \mapsto \{(nx, 1)\}$. The `Trim` processor, which discards the first n events, registers an association between input event at position i and output event at position $i - n$ (for every $i \geq n$); formally, $\partial\pi/\partial\overline{a} \triangleq (x, 1) \mapsto \{(x - n, 1)\}$. The `Fork` processor simply replicates the input events to its outputs; the i-th input event is

associated to the i-th output event of every output pipe, hence $\partial\pi/\partial\overline{a} \triangleq (x, y) \mapsto \{(x, 1)\}$.

The `Window` processor, which applies a processor P on a sliding window of n events, introduces a level of indirection. In order to produce the i-th output event from a stream of events $e_0, e_1, \ldots$, the processor evaluates an instance of P, called P_i, with the interval of events $[e_i, e_{i+n-1}]$. One must first compute the associations of this sub-evaluation, which yields a lineage function $\ell_i = \partial P_i/\partial[e_i, e_{i+n-1}]$. From this, we extract the associations of the last event produced by P, i.e. $S_i = \ell_i(n)$. Since the k-th event given to P_i actually corresponds to the $(k + i)$-th event ingested by the `Window` processor, the associations in S must be shifted by n positions, resulting in the set $S'_i = \{(x, 1) : (x + i, 1) \in S_i\}$. The lineage function for this processor can therefore be defined as $\partial\pi/\partial\overline{e} \triangleq (x, 1) \mapsto S'_x$.

As mentioned earlier, some processors will actually record different associations depending on the actual stream they receive. The lineage function for the `ApplyFunction` processor is determined by the associations of the underlying function f that is being applied on each event front. That is, $\partial\pi/\partial\overline{e}_1 \ldots \partial\overline{e}_m \triangleq (x, y) \mapsto (\partial f/\partial\overline{e}_1[x] \ldots \partial\overline{e}_m[x])(y)$. As we have seen above, some of these functions may associate their output to all or part of their input arguments, depending on their values.

Similarly, the `Cumulate` processor generally associates the i-th output event to all input events up to the i-th: this is consistent with the fact that the processor computes the progressive "accumulation" of all input events received so far, for a given function f (i.e. $\partial\pi_f/\overline{a} \triangleq (x, 1) \mapsto \{(i, 1) : i \leq x\}$). However, this default behaviour may be overridden depending on the cumulative function being used. Take for example an instance of `Cumulate` processor applied on a stream of Boolean values, using logical conjunction as its function. On the input stream $\top, \top, \bot, \top$, the processor will return the output stream $\top, \top, \bot, \bot$ –that is, as soon as a false value is received, the processor's output will be false forever. To explain why a given output event at position i is false, it suffices to point to an input event at position $j \leq i$ whose value is false. Therefore, we have the particular case $\partial\pi_\wedge/\partial\overline{a} \triangleq (x, 1) \mapsto \{(i, 1) : i \leq x\}$ if $\overline{a}$ contains only $\top$ values, and $\partial\pi_\wedge/\overline{a} \triangleq (x, 1) \mapsto \{(j, 1)\}$ if $x \geq j$ and j is the index of the first position of value $\bot$ in $\overline{a}$. A dual reasoning can be made for the case of disjunction.

Finally, among all of BeepBeep's core processors, `Slice` is the one with the most complex associations. As a reminder, `Slice` creates multiple instances of a processor P, and dispatches an input event to an instance of P based on the value returned by a slicing function f. The last output value produced by each instance of P is then aggregated using another function g. Let F be the image of f, i.e. the set of slice identifiers; we can suppose with loss of generality that F is the set of integers $[1; k]$. We can assume the existence of a function $p : [1; k] \times A^* \to A^*$, such that $p(s, \overline{a})$ produces the sub-trace corresponding to events of slice s. Similarly, let $\mu : [1; k] \times \mathbb{N} \to \mathbb{N}$ such that $\mu(s, i) = j$ if the i-the event of slice s corresponds to the j-th input event of the global trace.

We can first obtain a set of associations $S = (\partial g/\partial P(p(1, \overline{a}))[-1] \ldots \partial P(p(k, \overline{a}))[-1])(|\overline{a}|)$, which associates the output produced by g to the outputs of each slice, when evaluated using the last event produced by each instance of P. On each slice, the derivation of P can then be applied to obtain the association to their inputs, yielding a series of sets $S_i = (\partial P/\partial p(i, \overline{a}))(|\overline{a}|, 1)$. Each such set contains the positions of events of the sub-trace for slice i that are associated to the last event produced by P for slice i. Since these positions are relative to the sub-trace of each slice, this set must be transformed into positions of the global trace, yielding $S'_i = \{(\mu(i, x), 1) : (x, 1) \in S_i\}$. Given an input trace $\overline{a}$, we then have that $(\partial\pi/\partial\overline{a})(|\overline{a}|) \triangleq \bigcup_{i=1}^{k} S'_i$.

C. Derivation for State-Based Processors

Once derivation operators have been defined for the basic processors, we shall now describe derivations for processors that express sequential relationships between events, namely to state machines and temporal logic.

1) Finite-State Machines: When a violation to a compliance constraint expressed as a Moore machine is found in a log, existing tools, such as monitors, typically stop at the first event that makes the sequence non-compliant, and declare failure. The location in the trace where the monitor stops can already give some information to the user about the cause of the violation, but only in a fragmentary manner. Depending on the specification, the failure may be the result of the interplay between several events in the past that end up in a violation, and this information is not readily available by a classical monitor with a pass/fail verdict.

With the help of a derivation operator, a finite-state machine can provide finer information about the occurrence of a violation. Let $M = \langle S, s_0, T, \delta\rangle$ be a finite-state automaton, where S is a set of states, $s_0 \in S$ the initial state, T be a set of transition labels and $\delta : S \times T \to T$ be a labeled total transition function. Define $\lambda : S \to \{\top, \bot\}$ as a function that associates a Boolean value to each state, acting as the monitor's verdict. We shall further assume that states labeled with failure ($\bot$) are sink states; the set $S_\bot \subseteq S$ represents all such states. Since explanation is typically concerned with failures, we shall define $\partial\pi/\partial\overline{e} \triangleq (x, 1) \mapsto \{(x', 1) : 1 \leq x' \leq x\}$ when $\pi(\overline{e}) = \top$: when no failing verdict is produced, the output result is associated to all the input events ingested so far. What remains to be defined are the input/output associations in the case of a fail verdict.

There exist multiple ways of defining what amounts to a counter-example for a run in such an automaton. A first possibility is to associate any $\bot$ output to the first input event reaching a sink state. Formally, this is expressed as $\partial\pi/\overline{e} \triangleq \{(k, 1)\}$, where k is the smallest value such that $\pi(\overline{e}[1] \ldots \overline{e}[k]) = \bot$. However, merely returning the first event causing an immediate failure may not provide enough context. An alternate, and perhaps more insightful, derivation could be $\partial\pi/\partial\overline{e} \triangleq \{(k_1, 1), \ldots, (k_c, 1)\}$, where the k_i are a strictly increasing sequence of integers such that the subsequence $\overline{e}[k_1] \ldots \overline{e}[k_c]$ is a loopless run of the finite-state machine producing the verdict $\bot$. In other words, the derivation

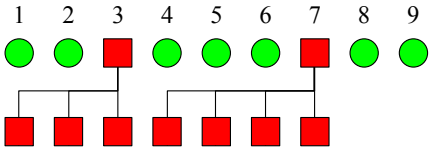

Figure 2: Input/output asssociations for the LTL **G** processor.

associates to a failing verdict the shortest sub-path in the input that produces the output.

2) Linear Temporal Logic: As we have seen, LTL is an alternate way in which compliance constraints on event sequences can be expressed. Since each temporal operator is a stand-alone processor, derivation can be defined in a simple way for each elementary operator separately.

Consider the case of the processor for operator **G**. By virtue of the semantics of LTL, we know that this processor delays the production of output events as long as its inputs are true; once a false event is received, it produces a burst of $\bot$ output values. For a $\bot$ event that is emitted at position j, an association is recorded with the last input event at position $i \geq j$ whose value is false. For an input trace of Boolean values $\overline{a}$, define $S^{\overline{a}}_{\bot,k} = \{i : k \leq i \leq |\overline{a}| \text{ and } \overline{a}[i] = \bot\}$. Then, we can stipulate that $\partial\pi/\partial\overline{a} \triangleq (x, 1) \mapsto \{(\min(S^{\overline{a}}_{\bot,x}), 1)\}$ if $S^{\overline{a}}_{\bot,x} \neq \emptyset$.

This is illustrated in Figure 2. The top row of the figure represents an input stream of Boolean values, with circles representing $\top$, and squares representing $\bot$. The bottom row shows the output produced by the G processor. Lines record the associations established between the inputs and the outputs. As one can see, the first three output events are associated with the first false value. Indeed, the verdict produced by the monitor for these three trace prefixes is "caused" by the presence of value $\bot$ at position 3. However, this event has no bearance on the output values produced for positions 4–7; they are rather caused by the presence of $\bot$ at input position 7. A similar reasoning can be applied to the other temporal operators.

D. Examples

These basic input/output associations appear relatively straightforward when taken in isolation, but turn out to provide surprisingly articulate and intuitive results, when processors are composed to form complex computation chains. In the following, we illustrate these principles by presenting a few simple examples of stream queries and their associated explanations. They are by no means a complete showcase of BeepBeep's functionalities.

1) Window Product: As a first example, consider the processor chain illustrated in Figure 3. This chain takes as input a stream of numerical values; it computes the product of each sequence of three successive values, checks whether this product is not equal to zero. This chain introduces a special processor, not described earlier, at the bottom of the figure, which simply turns any input event into a predefined constant —in this case, the value 0. Intuitively, the output of this chain can be translated as the assertion "the product of any three successive values must be greater than zero". Consider

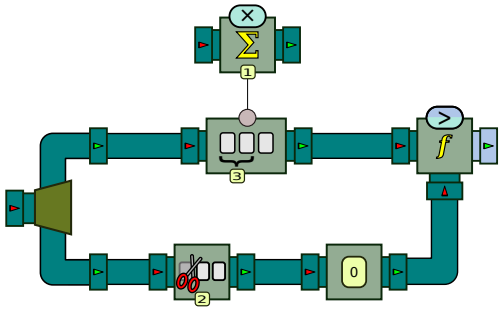

Figure 3: The BeepBeep chain of processors for the *Window product* query.

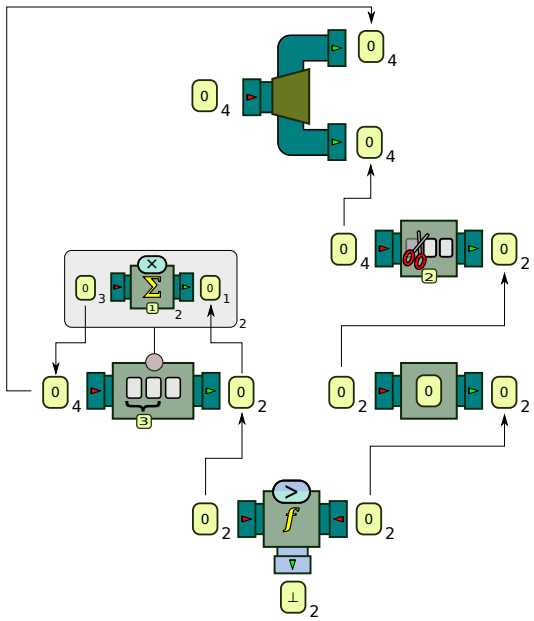

Figure 4: An explanation graph for the *Window product* query.

the input stream $3, 1, 4, 0, 5, 9, 2$ given to this pipeline. The output produced for this prefix will be the stream of Booleans $\top, \bot, \bot, \bot, \top$. Indeed, the first window of three events $(3, 1, 4)$ has a non-null product; however, it is easy to see that the next three windows, which all contain the number 0, have a product equal to zero and cause the emission of value $\bot$.

Suppose we want an explanation for the reason the second event of this output is false. It is possible to start from the output of the chain and apply the derivation operator ∂ on the last processor. This will result into a lineage function $\ell(x)$, which can then be evaluated at $x = 2$ to get the input events associated to the second output event. The process can then be repeated on upstream processor; this will produce a directed acyclic graph whose structure is depicted in Figure 4. The graph is read from bottom to top; each input or output event is represented with a number corresponding to its relative position in the stream in question.

Special attention should be given to the explanation for the result of the `Window` processor (left branch). This processor outputs a zero as its second event because the internal instance

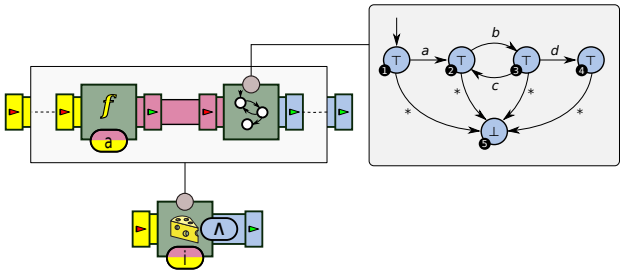

Figure 5: The BeepBeep chain of processors for the *Process lifecycle* property.

of the `Cumulate` processor associated to the second window returned zero. However, the reason for this null value is not explained by the whole window, but by the single 0 that corresponds, in this case, to the third event of the window. Ultimately, the whole graph converges back to a single input event, which is the zero value at position 4 in the input stream. This is in line with the intuition that output $\bot$ at position 2 is indeed caused by the presence of this zero in the input. Oftentimes, only the input/output associations of the extremities of the chain are relevant; in such a case, the graph can be "flattened" by keeping only the set of original input events that are mapped to a given output.

It is important to stress that this explanation graph depends on the output event chosen and the actual input stream given to the pipeline. Mere knowledge of the processor graph can be seen as lineage (similar to the information provided by Spline or Atlas), but is too coarse-grained to count as an *explanation* of a result.

2) Process Lifecycle: A second example is shown in Figure 5. This time, input events are assumed to be tuples of the form (i, a), where i is some numerical identifier, and a is the name of an action. This basic format is appropriate to represent a simple kind of business process log, where multiple interleaved process instances are distinguished by their value of i, and each instance is made of a sequence of actions. This use case is a prime example of the `Slice` processor, which in this case is used to separate events of each process instance based on their `id`, and feeds each sub-sequence into a chain that first fetches the `action` field of each event, and updates the state of a Moore machine accordingly.

In this particular case, one can see that the Moore machine for each instance has transitions to a "sink" state that produces value "false" ($\bot$). Any sequence that follows the intended pattern has the machine remain in a state that produces the value "true" ($\top$). Written as a regular expression, the language accepted by this machine corresponds to the string $a(bc)^+d$. The output of each Moore machine is aggregated into a Boolean conjunction; therefore, for the global processor chain to return $\top$, each currently active process instance must follow the intended lifecycle —otherwise the chain returns $\bot$.

Consider for example the following sequence of actions: $(1, a), (2, a), (2, b), (1, b), (2, c), (2, d)$. The processor's output for this prefix will be the sequence of Booleans $\top, \top, \top, \top, \top, \bot$. As one can see, this sequence of events contains two interleaved process instances, labeled 1 and 2. The sequence of actions for process 1 follows the intended pattern (ab), while the sequence of actions for process 2 ($abcd$) violates the lifecycle on the last event.

The iterated use of the derivation operator for each processor in the chain ultimately points to two events of the input log: tuples $(2, a)$ and $(2, d)$, corresponding to the second and sixth elements. This result provides two interesting pieces of information: first, the ID of the process that causes the global error, in this case process #2. Second, the derivation operator identifies a minimal sub-trace for this process that causes the error. Here, we can see that in the complete trace $abcd$, according to the definition of ∂ for state machines, the loop bc has no impact on the erroneous result and is therefore not included in the explanation.

3) LTL Property: Our last event log query involves Boolean connectives and LTL temporal operators. Its processor chain is shown in Figure 6. In this case, we assume the input events are lines of a CSV file, each containing a tuple $(action, p)$, where $action$ is an action name and p is an arbitrary numerical value. The chain decomposes this tuple by fetching the value of p (top branch) and the value of a (bottom branch). The condition $p < 0$ is evaluated on the top branch; the condition $action = a$ is evaluated on the bottom branch, for some predefined action name a.

The Boolean streams corresponding to these conditions are then sent through a piping of Boolean connectives and LTL operators. The end result is also a Boolean stream, which amounts to the evaluation of an LTL formula shown in the caption of Figure 6. Intuitively, this expression can be formulated as "every input event with a negative value for p must be followed by two successive events whose action is a".

As an example, consider the input stream made of the following four tuples $(b, 1), (c, -2), (a, 0), (d, 0)$. One can see that the output of the processor chain, after ingesting these four events, will be the sequence $\bot, \bot$. According to the semantics of LTL operators, this is caused by the fact that the trace suffixes starting at the first and second event violate the condition expressed above: they both contain an event with $p < 0$ that is not followed by two successive a. No definite verdict can be yet reached for the sub-traces that start at the third and fourth event; this is why no output event has been produced for these two inputs.

The repeated use of the derivation operator will retrace the first output event ($\bot$) to the inputs $(c, -2)$ and $(d, 0)$. This corresponds to a "witness" of the fact that an event with $p < 0$ has been seen, and that the second event that follows it does not have a as its action. Notice how event $(a, 0)$ is not part of the explanation, as it does not cause the erroneous verdict.

V. Implementation and Experiments

The theoretical concepts presented in the previous section have been concretely implemented into the BeepBeep event stream processing engine.

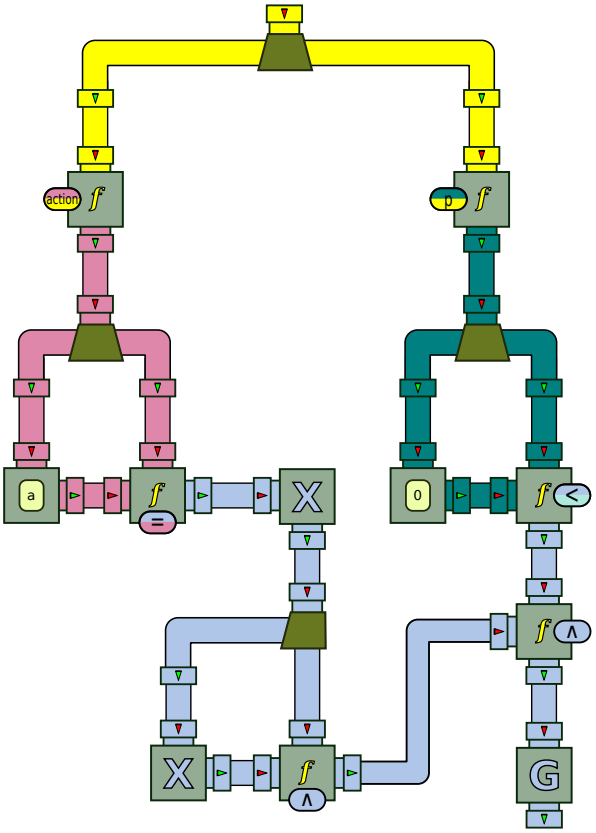

Figure 6: The BeepBeep chain of processors that checks the LTL property $\mathbf{G}\,(p < 0 \rightarrow \mathbf{X}\,(\text{action} = a \wedge \mathbf{X}\,(\text{action} = a)))$.

A. Implementation Details

The goal of these additions and modifications is to make lineage as transparent as possible to the end user. The implications of this requirement are twofold. First, all modifications must preserve backward compatibility: existing programs using BeepBeep without lineage should still be valid programs under the new version. Second, benefiting from data lineage in a program should require as few modifications as possible to a processor chain; that is, lineage should come at a little cost in terms of added complexity to the glue code.

All explainability functionalities in BeepBeep are centered around a singleton object called the *event tracker*. The sole purpose of this object is to answer lineage queries: given an output event at a specific position in an output stream computed by a processor chain, the event tracker must point to the events of the chain's inputs that contribute to (or "explain") the fact that this particular output event contains this particular value. Since each processor instance in BeepBeep is given a numerical identifier that is unique across a given program, the associations for each processor of a chain can be recorded and distinguished.

Equipped with this basic setup, supporting lineage in processors amounts to the insertion, in each class descending from the top-level `Processor`, of appropriate calls to a tracker's `associate()` methods. Since processors have a streaming mode of operation, these calls should also be made in a streaming fashion. This means that associations are recorded progressively as the input events are ingested, as soon as such associations can be determined.

However, processors must be aware of the existence of such an event tracker so that they can call it. This is why the `Processor` class is modified in such a way that each of these objects can now store a reference to an event tracker. By default, lineage is turned off: processors are instantiated with a null reference as their default event tracker, indicating that no call to `associate()` needs to be made. This default can be changed by passing a non-null implementation of `EventTracker` to a processor object after its creation.

For the end user, enabling explainability amounts to a single modification to the code for an existing query. The user must create an instance of an `EventTracker`, and pass it to the BeepBeep `Connector` object used to pipe processors together. This mechanism entails that explainability can be switched on or off, and also that, through the use of different tracker instances, different explanations can be obtained for the same stream query.

B. Experimental Setup

In order to assess the viability of such a system in practical situations, we performed an empirical evaluation of BeepBeep's lineage functionalities through an experimental benchmark. In this section, we report on these results, which have been obtained by running BeepBeep on various processor chains. They are aimed at measuring the impact, both in terms of computation time and memory, of the introduction of lineage functionalities inside the system. As we have seen, this is possible thanks to a switch provided by BeepBeep, and which allows users to completely disable lineage tracking if desired.

The experiments were implemented using the LabPal testing framework [12], which makes it possible to bundle all the necessary code, libraries and input data within a single self-contained executable file, such that anyone can download and independently reproduce the experiments. A downloadable lab instance containing all the experiments of this paper can be obtained online from Zenodo, a research data sharing platform [15]. All the experiments were run on a Intel CORE i5-7200U 2.5 GHz running Ubuntu 18.04, inside a Java 8 virtual machine with 1746 MB of memory.

The experiments comprise a number of "scenarios", each made of an event source and a computation to be executed on this event stream. These include:

- *Window product*: a randomly generated source of numerical values, on which the calculation of Figure 3 is applied.
- *Process lifecycle*: a source of interleaved instances of events, on which the chain of Figure 5 is computed.
- *LTL property*: a source of randomly generated tuples corresponding to the chain of Figure 6.
- *CVC procedure*: a log taken from the Compliance Checking Challenge 2019 [7], that contains operations recorded from multiple actual instances of a medical procedure. The property evaluated on this log checks compliance of the sequence of steps to a BPMN model of the procedure.

Query	No tracker	With tracker
CVC Procedure	2000200.0	370407.4
LTL property	23421.545	3488.3154
Payment	7525.207	2886.2915
Process lifecycle	4456.7734	3000.6
Window product	200020.0	31850.318

Table I: Relative throughput overhead.

Query	No tracker	With tracker
CVC Procedure	359317	3226525
LTL property	12341	53028281
Payment	10259056	13404048
Process lifecycle	24039551	41147231
Window product	5294	7404930

Table II: Relative memory overhead.

- *Payment*: a dataset that contains events pertaining to travel expense claims from the Eindhoven University of Technology [23]. On this log, it is checked that the total duration of each claim process never exceeds some threshold.

The first three scenarios involve abstract traces and queries, and act as a form of "stress test", while the latter two scenarios involve real-world logs.

1) Impact on Throughput: The first element we measured is the impact on processing speed, or *throughput*. Table I shows the results for various types of stream queries. Each line represents a pair of experiments, corresponding to the evaluation of a stream query both with and without the use of a tracker. The measured value in each case is the average throughput, in number of input events processed per second.

Unsurprisingly, turning lineage on incurs a non-negligible slowdown, by as much as 6.71× for the queries we considered. This is caused by the fact that, on each new event, a processor now calls the event tracker possibly multiple times, in order to register associations between inputs and outputs. These results should be put in context with respect to existing works that include a form of lineage. The Mondrian system reports an average slowdown of 3× [9]; pSQL ranges between 10× and 1,000× [1]; the remaining tools do not report CPU overhead. Time overhead for Spline [21] is close to zero, but as we have discussed, it provides lineage information at a much coarser level of granularity. Of course, these various systems compute different types of lineage information, but these figures give an outlook of the order of magnitude one should expect from such systems.

2) Impact on Memory: A second part of the experiment consisted in measuring the amount of additional memory required by the use of an event tracker. Memory was computed using the `SizePrinter` object from the Azrael serialization library[6]. This tool performs a recursive traversal of the member fields of a Java object, down to primitive types, and computes the sum of their reported sizes. The end result is a much more accurate indication of the memory actually consumed by an object, than would be a measurement of the JVM's memory footprint.

The results are summarized in Table II. We can see that the relative impact on memory is larger than the impact of lineage on computation time. This is consistent with the intuition that lineage tracking requires one to "remember" more things, much more than to "compute" more things. This consumption is still relatively reasonable in the absolute: for example, with the *Window product* processor chain, it would take an input file of 86 million lines before filling up the available RAM in a 64 GB machine with lineage data.

[6]https://github.com/sylvainhalle/Azrael

Query	Memory per event
CVC Procedure	286
LTL property	5301
Payment	314
Process lifecycle	1710
Window product	739

Table III: Average memory overhead (in bytes) per input event incurred by the use of an event tracker.

The large relative blow-up is mostly caused by the fact that, for many processor chains, evaluating a query without lineage requires a constant amount of space. In contrast, we observed that, for all the functions considered in this paper, each element of the output contributes for a constant amount of lineage data; hence the tracking-enabled pipeline uses a linear amount of space. Table III gives, for each query we considered, the memory overhead per input event incurred by the use of an event tracker. These figures should be put in context by comparing the overhead incurred by other lineage tracking tools. Notably, related systems for provenance in databases (namely Polygen [26], Mondrian [9], MXQL [25], DBNotes [5] pSQL [1] and Orchestra [16]) do not report their storage overhead for provenance data.

VI. Conclusion and Future Work

In this paper, we presented a framework to provide explainable results on event stream queries. Using the notion of a derivation operator, we have shown how input/output relations on elementary computation units can be formally defined in such a way that specific output events in a stream can be retraced to the input events that "explain" their presence. Thanks to the principle of composition, these elementary units can be piped together to form complex computation chains, and the explainability functionalities can then easily be lifted to these chains.

Through a few examples, it has been shown how such explainability capabilities can provide articulate and intuitive explanations for a result. These concepts have been implemented into the BeepBeep event stream processing engine, and incur very minimal modification to existing queries in order to be used: a single line of code suffices to switch the mechanism on or off. To the best of our knowledge, BeepBeep is the first event stream processing engine that provides such a simple,

yet all-encompassing explanation system. The implementation has been tested, both on abstract and real-world instances of logs for various types of stream queries; these tests indicate that the use of explainability incurs both a time and a memory overhead that is linear in the size of the input log.

These promising results open the way to multiple research questions and improvements over this first solution. Extensions to BeepBeep have been developed to perform trend deviation detection and predictive analytics [20], among other uses; it is planned to expand the basic explanation capabilities to these extensions in the near future. It shall also be noted that the system in its current state can only record associations between whole events. However, there exist situations where a finer granularity in the relationships between inputs and outputs would be required, such as when events are extracted from parts of a larger "document" such as an XML event. In addition, our notion of derivation operator currently only allows a single explanation for any given output event (although this explanation may itself involve multiple input events). However, there exist situations where multiple alternative explanations could be produced; for example, it is possible that removing any one of a set of actions would restore compliance of a trace to a specification, resulting in as many possible "explanations" for the violation.

The implementation of the explanation mechanism could also be optimized in a few ways. First, we can observe that some processors always record the same association for each input/output event pair. Instead of recording this fact for every event, considerable savings, both in terms of time and space, could be achieved by making the tracker replace these individual associations with a single generic rule. The existence of a lineage tracking system inside BeepBeep also opens the way to a myriad of exciting research questions. For example, given that a part of the input is considered corrupted, are there parts of the output that are not affected by this corruption? Such questions could be studied both concretely (by studying a particular input-output pair), but more interestingly by reasoning over all the possible input-output pairs of a given stream query.

References

[1] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *VLDB J.*, 14(4):373–396, 2005.

[2] R. P. J. C. Bose and W. M. P. van der Aalst. Trace alignment in process mining: Opportunities for process diagnostics. In R. Hull, J. Mendling, and S. Tai, editors, *Business Process Management - 8th International Conference, BPM 2010, Hoboken, NJ, USA, September 13-16, 2010. Proceedings*, volume 6336 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 2010.

[3] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In J. V. den Bussche and V. Vianu, editors, *Proc. ICDT 2001*, volume 1973 of *LNCS*, pages 316–330. Springer, 2001.

[4] L. Chiticariu and W. C. Tan. Debugging schema mappings with routes. In U. Dayal, K. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y. Kim, editors, *Proc. VLDB 2006*, pages 79–90. ACM, 2006.

[5] L. Chiticariu, W. C. Tan, and G. Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. In F. Özcan, editor, *Proc. SIGMOD 2005*, pages 942–944. ACM, 2005.

[6] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

[7] R. de la Fuente, M. Sepúlveda, and R. Fuentes. Central veinous catheter, compliance checking challenge 2019, 2019. https://data.4tu.nl/repository/uuid:c923af09-ce93-44c3-ace0-c5508cf103ad.

[8] E. G. L. de Murillas, H. A. Reijers, and W. M. P. van der Aalst. Everything you always wanted to know about your process, but did not know how to ask. In M. Dumas and M. Fantinato, editors, *BPM Workshops*, volume 281 of *Lecture Notes in Business Information Processing*, pages 296–309, 2016.

[9] F. Geerts, A. Kementsietsidis, and D. Milano. MONDRIAN: annotating and querying databases through colors and blocks. In L. Liu, A. Reuter, K. Whang, and J. Zhang, editors, *Proc. ICDE 2006*, page 82. IEEE Computer Society, 2006.

[10] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In L. Libkin, editor, *Proc. PODS 2007*, pages 31–40. ACM, 2007.

[11] P. Groth and L. Moreau, 2013. http://www.w3.org/TR/prov-overview/, Accessed November 12th, 2019.

[12] S. Hallé, R. Khoury, and M. Awesso. Streamlining the inclusion of computer experiments in a research paper. *IEEE Computer*, 51(11):78–89, 2018.

[13] S. Hallé and S. Varvaressos. A formalization of complex event stream processing. In M. Reichert, S. Rinderle-Ma, and G. Grossmann, editors, *18th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2014, Ulm, Germany, September 1-5, 2014*, pages 2–11. IEEE Computer Society, 2014.

[14] S. Hallé. *Event Stream Processing with BeepBeep 3: Log Crunching and Analysis Made Easy*. Presses de l'Université du Québec, 2018.

[15] S. Hallé. Benchmark for BeepBeep data lineage (LabPal instance), 2020. DOI: 10.5281/zenodo.4027100.

[16] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In A. K. Elmagarmid and D. Agrawal, editors, *Proc. SIGMOD 2010*, pages 951–962. ACM, 2010.

[17] L. Moreau, B. V. Batlajery, T. D. Huynh, D. Michaelides, and H. Packer. A Templating System to Generate Provenance. *IEEE Transactions on Software Engineering*, 44(2):103–121, Feb. 2018.

[18] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

[19] A. Polyvyanyy, C. Ouyang, A. Barros, and W. M. P. van der Aalst. Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.*, 100:41–56, 2017.

[20] M. Roudjane, D. Rebaine, R. Khoury, and S. Hallé. Predictive analytics for event stream processing. In *Proc. EDOC 2019*, pages 171–182. IEEE, 2019.

[21] J. Scherbaum, M. Novotny, and O. Vayda. Spline: Spark lineage, not only for the banking industry. In *BigComp 2018*, pages 495–498. IEEE Computer Society, 2018.

[22] S. Suhothayan, K. Gajasinghe, I. L. Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara. Siddhi: a second look at complex event processing architectures. In R. Dooley, S. Fiore, M. L. Green, C. Kiddle, S. Marru, M. E. Pierce, M. Thomas, and N. Wilkins-Diehr, editors, *Proc. GCE 2011*, pages 43–50. ACM, 2011.

[23] B. F. van Dongen. BPI challenge 2020: Request for payment. 4TU. centre for research data. dataset., 2020. https://doi.org/10.4121/uuid:895b26fb-6f25-46eb-9e48-0dca26fcd030.

[24] S. J. van Zelst, A. Bolt, M. Hassani, B. F. van Dongen, and W. M. P. van der Aalst. Online conformance checking: relating event streams to process models using prefix-alignments. *Int. J. Data Sci. Anal.*, 8(3):269–284, 2019.

[25] Y. Velegrakis, R. J. Miller, and J. Mylopoulos. Representing and querying data transformations. In K. Aberer, M. J. Franklin, and S. Nishio, editors, *Proc. ICDE 2005*, pages 81–92. IEEE Computer Society, 2005.

[26] Y. R. Wang and S. E. Madnick. A polygen model for heterogeneous database systems: The source tagging perspective. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proc. VLDB 1990*, pages 519–538. Morgan Kaufmann, 1990.