Model Checking of Stream Processing Pipelines

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

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

Abstract

Event stream processing (ESP) is the application of a computation to a set of input sequences of arbitrary data objects, called "events", in order to produce other sequences of data objects. In recent years, a large number of ESP systems have been developed; however, none of them is easily amenable to a formal verification of properties on their execution. In this paper, we show how stream processing pipelines built with an existing ESP library called BeepBeep 3 can be exported as a Kripke structure for the NuXmv model checker. This makes it possible to formally verify properties on these pipelines, and opens the way to the use of such pipelines directly within a model checker as an extension of its specification language.

2012 ACM Subject Classification Theory of computation → Streaming models

Keywords and phrases stream processing, model checking

Digital Object Identifier 10.4230/LIPIcs.TIME.2021.5

Supplementary Material *Software (Source Code)*: `https://doi.org/10.5281/zenodo.5386462`

Funding Canada Research Chair on Software Specification, Testing and Verification

Acknowledgements The open access publication of this article was supported by the Alpen-Adria-Universität Klagenfurt, Austria.

1 Introduction

Information systems generate logs from a variety of sources: business process management engines, web servers and instrumented programs alike produce sequences of data elements called *event streams*. The analysis of these logs, either offline or in real-time, can be put to numerous uses: computation of various metrics, evaluation of compliance with respect to a policy [34], detection of anomalous patterns or presence of bugs [36]. The field of Complex Event Processing (CEP) concentrates on the real-time evaluation of expressive queries over streams of events [28]. Typically, CEP scenarios involve not only the expression of temporal patterns, but also arithmetical (counts, sums) or even statistical computations over event data. The development of tools and libraries for the processing of event streams has seen a rapid growth in the past decade, with popular products such as Siddhi [33], Esper [2] or Apache Flink [1].

As we shall see, none of these stream processing frameworks is directly amenable to the formal verification of properties on their execution. This is an important gap, as multiple tasks related to the management of event pipelines, which would require the establishment of invariants, are currently impossible. The first obvious example is correctness, which is the guarantee that a processing pipeline produces the expected result for all possible input streams. The static verification of a processing pipeline can also be put to other uses: identifying "dead paths" (parts of a pipeline that never contribute to the output and are therefore useless), verifying the equivalence of two implementations of the same computation, or making sure that I/O buffers allocated to each part of the chain are of sufficient size.

© Alexis Bédard and Sylvain Hallé;
licensed under Creative Commons License CC-BY 4.0

This paper addresses this issue, by describing a formalization of event stream processing pipelines as Kripke structures that can be handled by a model checker. Section 2 briefly describes an existing stream processing library called BeepBeep [24]. Section 3 introduces an extension to this library that makes it possible to export a BeepBeep pipeline into an input file for the NuXmv model checker [15]. Section 4 illustrates the use of this extension by presenting a few scenarios that require the model checking of stream processing programs; an overview of the performance, in terms of execution time, is provided by experiments discussed in Section 5. Section 6 gives a broad portrait of the state of the art in event stream processing related to this question, while Section 7 touches upon future research opportunities.

2 Stream Processing with BeepBeep

Taken in its broadest sense, event stream processing can be defined as the application of a computation over a sequence of data elements called *events*. This computation is typically executed in a "streaming" fashion: its output (generally another event sequence) is progressively produced as input events of the sequence are consumed. Classical applications of this model include the realtime calculation of descriptive statistics (such as the average of values over a sliding window) and the detection of sequential patterns of events in the sequence.

We now describe BeepBeep, an open source event stream processing engine implemented in Java [20]. Over the years, BeepBeep has been involved in multiple case studies [10,11,21,25,36]. We briefly provide here a formal description of the operation of the system's core elements. For further details, the reader is referred to a complete textbook describing the system [24].

Let Σ be a set of arbitrary data elements $\{\sigma_1, \sigma_2 \dots\}$ called *events*; an event stream, noted $\overline{\sigma}$, is an element of the set Σ^*. The notation $\overline{\sigma}[i]$ is used to denote the i-th event of $\overline{\sigma}$. We denote by $\overline{\sigma} \preceq \overline{\sigma}'$ the fact that $\overline{\sigma}$ is a prefix of $\overline{\sigma}'$. If $\Sigma_1, \dots, \Sigma_n$ are event alphabets, a stream vector $\vec{v} = \langle \overline{\sigma}_1, \dots, \overline{\sigma}_n \rangle$ is an element of $(\Sigma_1 \times \cdots \times \Sigma_m)^*$; note that this imposes that each stream within the vector is of the same length. A stream vector $\langle \overline{\sigma}_1, \dots, \overline{\sigma}_n \rangle$ is a prefix of another vector $\langle \overline{\sigma}'_1, \dots, \overline{\sigma}'_n \rangle$ if each $\overline{\sigma}_i$ is a prefix of $\overline{\sigma}'_i$. Given a n-stream vector $\vec{v} = \langle \overline{\sigma}_1, \dots, \overline{\sigma}_n \rangle$ and an n-uple $v = (\sigma_1, \dots, \sigma_n)$, the concatenation $\vec{v} \cdot v$ is the n-uple $\langle \overline{\sigma}_1 \cdot \sigma_1, \dots, \overline{\sigma}_n \cdot \sigma_n \rangle$; this notion can then easily be extended to the concatenation of n-stream vectors.

What in BeepBeep are called "processors" are functions $\pi : (\Sigma_1 \times \cdots \times \Sigma_m)^* \to (\Sigma'_1 \times \cdots \times \Sigma'_n)^*$, with the condition that $\vec{v} \preceq \vec{v}'$ implies $\pi(\vec{v}) \preceq \pi(\vec{v}')$. The values of m and n are the input and output *arity* of the processor, which is often represented with the notation m:n. The core of the BeepBeep engine is made of a handful of basic processors performing elementary operations on streams. Each of these processors is represented graphically as a box with input and output "pipes", and its operation is symbolized by a standardized pictogram. The most common of these processors are shown in Figure 1.

First, the *ApplyFunction* processor lifts any function $f : \Sigma_1 \times \cdots \times \Sigma_m \to \Sigma'_1 \times \cdots \times \Sigma'_n$ into a processor $\pi : (\Sigma_1 \times \cdots \times \Sigma_m)^* \to (\Sigma'_1 \times \cdots \times \Sigma'_n)^*$ defined as $\pi(\vec{v} \cdot (\sigma_1, \dots, \sigma_m)) \triangleq \pi(\vec{v}) \cdot f(\sigma_1, \dots, \sigma_m)$. *CountDecimate* is a 1:1 processor that keeps one event every k, and is defined as $\pi(\vec{v}) \triangleq \langle \vec{v}[0], \vec{v}[k], \vec{v}[2k], \dots \rangle$. *Trim* removes the first k events of the stream and is defined as $\pi(\vec{v}) \triangleq \langle \vec{v}[k], \vec{v}[k+1], \vec{v}[k+2], \dots \rangle$; *Fork* is a 1:$n$ processor that simply replicates its input on n independent outputs: $\pi(\vec{v}) \triangleq \langle \vec{v}, \dots, \vec{v} \rangle$. *Filter* is a processor $\pi : (\Sigma \times \{\top, \bot\})^* \to \Sigma^*$ that discards events based on a stream of Boolean values. The event at position n in the first stream is sent to the output, if and only if the event at the same position in the second stream is the Boolean value true; formally: $\pi(\vec{v} \cdot (\sigma, b)) \triangleq \pi(\vec{v}) \cdot \sigma$ if $b = \top$, and $\pi(\vec{v})$ otherwise.

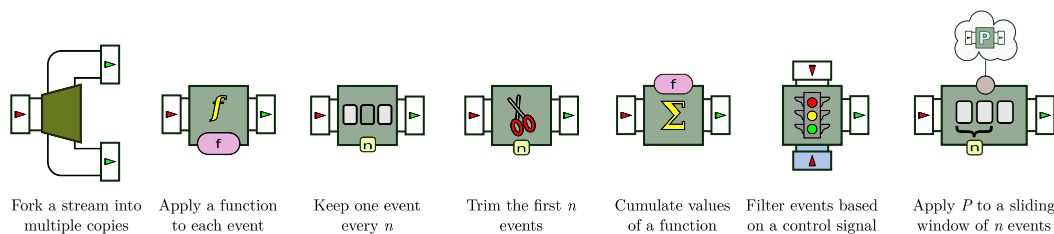
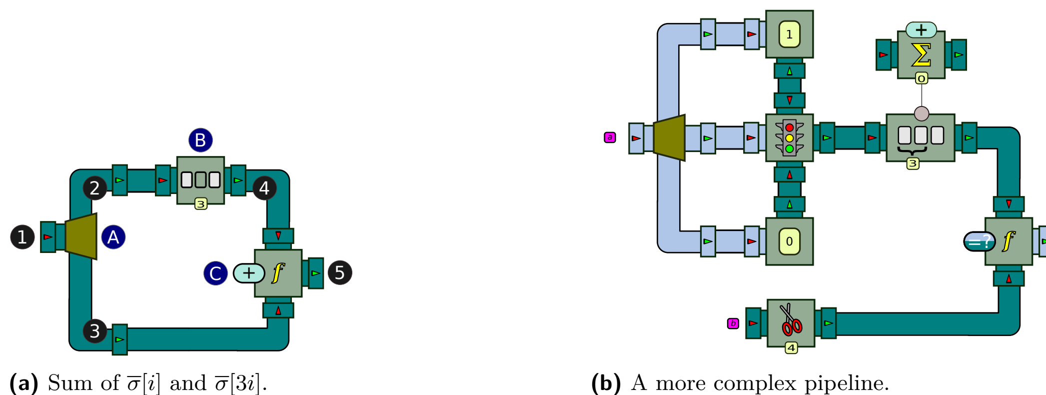


Figure 1 Pictorial representation of BeepBeep's core processors.



(a) Sum of $\overline{\sigma}[i]$ and $\overline{\sigma}[3i]$. **(b)** A more complex pipeline.

Figure 2 Two examples of processor pipelines discussed in the text.

As its name implies, the *Cumulate* processor is designed to "accumulate" the successive values of a binary function. Given a function $f : \Sigma^2 \rightarrow \Sigma$ and an implicit initial value $\sigma_0 \in \Sigma$, the processor is defined recursively as: $\pi(\langle\sigma\rangle) \triangleq \langle f(\sigma_0, \sigma)\rangle$, and $\pi(\langle\overline{\sigma} \cdot \sigma\rangle) = \pi(\langle\overline{\sigma}\rangle) \cdot \langle f(\pi(\langle\overline{\sigma}\rangle)[-1], \sigma)\rangle$, where $\pi(\langle\overline{\sigma}\rangle)[-1]$ stands for the last event produced by π on the input stream $\overline{\sigma}$. This generic construction can represent various types of computations depending on the function used. For example, if f is addition and $\sigma_0 = 0$ is used as the start value, π produces an output stream where the i-th event is the sum of all input events up to the i-th. If f is Boolean conjunction and $\sigma_0 = \top$, π produces an output stream where the i-the event is the conjunction of all input events up to the i-th.

Finally, the *Window* processor is probably the most complex included in BeepBeep's core. It is parameterized by another processor $\pi' : \Sigma^* \rightarrow \Sigma'^*$, which is used to evaluate a sub-stream of k successive events in the global input stream:

$$\pi(\langle\sigma_0 \cdots \sigma_n\rangle) \triangleq \langle\pi'(\langle\sigma_0 \dots \sigma_{k-1}\rangle)[-1], \pi'(\langle\sigma_1 \dots \sigma_k\rangle)[-1], \dots, \pi'(\langle\sigma_{n-k} \dots \sigma_n\rangle)[-1]\rangle$$

Intuitively, the first output event of π is the last event produced by π' on the window of width k running from input events 0 to $k-1$; The second output event of π is the last event produced by π' on the window of width k running from input events 1 to k, and so on. Note that this processor produces no output event until it receives its first k input events. This construction is very generic, and distinguishes BeepBeep from other stream processing engines, as typically, sliding windows only apply to aggregations over numerical values; in BeepBeep any computation can be put in a sliding window, as π' is an arbitrary *processor*.

Not represented here is the *Group* processor, which makes it possible to encapsulate a complete pipeline and give it as an argument to another processor, as if it were a single "black box".

In order to perform more complex computations, processors can be composed (or "piped") together, by letting the output of one processor be the input of another. This piping is possible as long as the type of the first processor's output matches the second processor's input type. Figure 2a shows a graphical rendition a possible pipeline, where events flow from left to right. It represents a calculation made of three processors (A–C) connected by five pipes (1–5), and where the i-th output event is the sum of input events at positions $3i$ and i.

It is important to note that BeepBeep processors are not defined on tuples of streams, but are rather in terms of streams of tuples. For processors with an input arity of 2 or more, this entails that the processing of their input is done *synchronously*: a computation step is performed if and only if an event can be consumed from each input stream. This is a strong assumption; many other CEP engines allow events to be processed asynchronously, meaning that the output of a query may depend on what input stream produced an event first. As a consequence, processors must implicitly manage buffers to store input events until a result can be computed. This buffering is implicit: it is absent from both the formal definition of processors and any graphical representation of their piping. Consider for example the input stream $0, 1, 2, \ldots$ fed to the previous pipeline. The fork A sends the event to the input of B and the lower pipe of C, and processor B lets the event through to the upper pipe of C; therefore, the sum $0+0$ is computed. However, feeding the next event (1) to the chain results in no output. Fork A sends the event through pipes 2 and 3, but processor B discards this event. As a result, processor C cannot consume an event from both its inputs, and event 1 is stored in the input queue associated to its second input pipe.

3 A Formal Modeling of BeepBeep Processors

This section presents a framework that enables the model checking of stream processing models, based on the synchronous formal semantics of BeepBeep processors described in Section 2. More precisely, it shows how a processor chain can be turned into a Kripke structure specified in the form of an input model for the NuXmv model checker [15].

3.1 Design Hypotheses

The translation of a processor pipeline into a finite-state model rests on a number of hypotheses, which are necessary in order to address some of the limitations incurred by the use of a model checker.

The first obvious constraint is related to the use of event queues by processors. These queues are theoretically unbounded, as is exemplified in the pipeline of Figure 2a; in this scenario, the size of the input queue for the bottom pipe of processor C grows indefinitely. The translation to a NuXmv model must therefore impose a fixed maximum size Q to the input queues of each processor. Event types are restricted to the scalars supported by the target model checker, namely Booleans, integers and symbolic constants. Integers themselves are bounded to the range $[0, N-1]$, so all arithmetic operations are implicitly assumed to be performed modulo N.

BeepBeep allows events to be generated in two modes. In *pull* mode, the handling of events in the processor pipe is triggered by requesting for a new output event, which propagates upstream. In order to produce this output event, the processor may require itself to fetch new events from its input(s), which in turn may ultimately lead to fetching events from the original event stream. On the contrary, in *push* mode, output events are produced by signaling the arrival of new events at the input side of the processor pipe. This

may trigger the computation of an output event by the processor, propagating push signals downstream. Our proposed modeling currently simulates a processor chain that operates in push mode only.

Computations in the target model are performed in a lockstep fashion. For example, in the pipeline of Figure 2a, we have seen that pushing an event in pipe 1 results in this event being pushed through pipes 2 and 3; this, in turn, triggers the evaluation of this event through the *CountDecimate* processor, which may result in the event being pushed in pipe 4, and so on. In the generated model, all computations in a chain resulting from a single upstream push operation occur in a single transition. This design choice reduces the potential state space of the resulting system. Were it defined in such a way that each processor along the chain required its own transition, an event pushed through a straight chain of n processors would result in $n+1$ successive states of the model; our definition rather conflates them into a single transition between two states.

However, this also brings challenges of its own. First, the definition of some processors becomes more intricate, as some operations that are easily described through loops must somehow be "flattened" into a single operation (we shall see that the *Window* processor presents a particular challenge in this respect). Second, this design only handles processors that produce 0 or 1 output event for a given input event. Fortunately, most processors satisfy this condition, including all those presented in this paper.

3.2 Pipeline Modeling

Based on these hypotheses, it is now possible to define the translation of a BeepBeep pipeline into a corresponding NuXmv model. The first element to simulate is that of a pipe connecting two processors. Each pipe is represented by a triplet of variables in the model, called p_c, p_b and p_ℓ. First, p_c is the variable that carries the events themselves: its value at a given computation step indicates the event that this pipe carries from upstream to downstream at that moment. Variable p_b is a Boolean flag set to value $\top$ when the pipe does contain an event, and to $\bot$ otherwise. Finally, variable p_ℓ is also a Boolean flag that indicates whether the event being pushed in the pipe is the last of the chain. Some processors use this signal to perform a different action when receiving what is announced as the last event of a stream.

The modeling of processors takes advantage of NuXmv's concept of *modules*, which makes it possible to encapsulate a set of internal variables and transitions into a self-contained computation unit. Each module instance can accept a number of parameters, which it can read from or write to, in addition to its own internal variables. Each module corresponding to a m:n processor has the same signature, made of $3m+3n+1$ parameters. Its first $3m$ parameters are the triplets that define its m input pipes; its next $3n$ parameters define its n output pipes; a final parameter is a Boolean *reset* flag used to signal the processor that it should revert its internal state to a predefined initial state.

The composition of processors in a pipeline is achieved by creating as many triplets of variables as there are pipes in the chain, and passing to each module instance the appropriate pipe variables either as inputs or outputs. An example is shown in Figure 3, where a fragment of the NuXmv code for the pipeline of Figure 2a is presented. Variables $p_{c,i}$, $p_{b,i}$ and $p_{\ell,i}$, for $i \in [1,5]$, and a global reset flag *prst* are first declared. Then, the three processors are represented by variables π_i, whose type is a module corresponding to the appropriate processor. Finally, one can observe by the input and output parameters of each processor that their connections are made in accordance with the illustration.

```
MODULE main
  VAR
    pc_1: 0..N;
    pb_1: boolean;
    pl_1: boolean;
    prst: boolean;
    ...
    pi_1: Fork_2(pc_1, pb_1, pl_1, pc_2, pb_2, pl_2, pc_3, pb_3, pl_3, prst);
    pi_2: Decimate_3(pc2, pb_2, pl_2, pc_4, pb_4, pl_4, prst);
    pi_3: ApplyFuction(pc_4, pb_4, pl_4, pc_3, pb_3, pl_3, pc_5, pb_5, pl_5, prst);
```

Figure 3 Creating the pipeline of Figure 2a in NuXmv.

Concretely, this process has been implemented by additions to a fork of the original BeepBeep library.[1] BeepBeep already provides a class that allows one to navigate through a processor pipeline using the *Visitor* design pattern [19]. Our implementation adds a new object, called `SmvCrawler`, which traverses a pipeline and makes an inventory of all the processors encountered, along with their respective connections. Each connection results in a unique triplet of model variables, and the processor variables are instantiated according to their type and their input/output relationships. The resulting NuXmv code is then printed to a file.

3.3 Processor Modeling

Any processor that can be exported as a NuXmv module must implement a Java interface called `SmvPrintable`, which consists of a single method called `printSmv()`. This method receives as arguments the user-specified values of Q and N described in Section 3.1. It prints to the output the NuXmv code snippet defining its corresponding module. For some processors, such as *Trim* and *CountDecimate*, the translation is straightforward and does not depend on Q and N. We describe in the following the details of the translation for processors presenting particular challenges.

The first such processor is *ApplyFunction*, for functions of input arity $m > 1$. As explained earlier, this processor must handle input queues, whose behavior needs to be simulated through arrays. Concretely, the queue associated to an input pipe is represented by two array variables, q_c and q_b. The role of these two variables is similar to p_c and p_b: the first contains the actual events inside the queue, while the second is used to record whether an event is contained in the queue at a given position. Depending on the combination of input pipes where an event is pushed at a given computation step, the transition relation of the processor must take care of shifting queue contents forwards or backwards into the arrays. The behavior of this processor when an event must be added to a full queue is left undefined (this shall be discussed later).

The *ApplyFunction* processor is also special in that it is parameterized by another object (the function f to apply). Therefore, BeepBeep functions themselves are also represented as stateless NuXmv modules: a $m{:}n$ function becomes a module with $m + n$ arguments, and whose transition relation in each state sets to the n outputs the result of applying f on the m inputs. Consequently, each instance of *ApplyFunction* module needs only to be passed the module created for its function f; however, this is done in a modular fashion, as its module merely sets inputs and reads outputs from another abstract module corresponding to the function to apply. A similar process is done for *Cumulate* which, in addition to the module corresponding to the function f to accumulate values from, also has an internal variable *last* to store the value produced by f on the last push of an event.

[1] https://doi.org/10.5281/zenodo.5386462

Finally, the *Window* processor deserves some discussion. We recall from Section 2 that sliding windows in BeepBeep can be applied on any processor π –including stateful processors. However, the design hypotheses elicited in Section 3.1 stipulate that each push action on a pipeline must be accounted for in a single transition of the model. This rules out the "naïve" way of handling windows, which would be to wait for k events to be received, push them into a fresh instance of π, collect the last event it produces, and push it downstream.

The solution we propose works differently, and makes use of the *reset* flag that each processor receives. For a window of width k, the *Window* module maintains as internal variables k instances $P_1, \dots, P_k$ of the processor π to be applied on a window. Each successive processor receives the same input stream, with an increasing number of events trimmed from the beginning. Hence, P_1 receives the whole stream, P_2 receives all events starting from the second, and so on. An internal variable c determines which of these instances is due to be considered for the next output event. Once an instance P_c has received k events, its output is collected and set as the output of the *Window* processor, and that instance is then reset to its initial state by setting its *reset* flag to $\top$. The value of c then shifts to the next processor instance. In such a way, any processor can be applied to a sliding window, yet still use a fixed number of variables and be processed in a single transition of the model. To the best of our knowledge, this implementation is the first to consider stream processing pipelines with cumulative functions and arbitrary sliding windows, which makes it possible to represent complex operations on streams, such as "windows on windows", and the like.

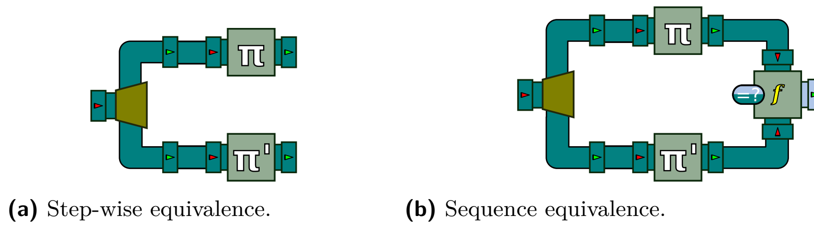
Finally, the *Group* processor can be handled easily. Each group is exported as an independent NuXmv module, where a fresh instance of `SmvCrawler` is instructed to generate the contents of the group (including any other module it may contain). Its piping is then taken care of in a similar way as the `main` module, with the exception that the group is exported as a module with a unique name, and with parameters corresponding to its open input and output pipes.

4 Applications

Equipped with the capability of performing model checking on stream processing pipelines, a number of applications become possible. A first obvious case is the verification of an arbitrary invariant on the execution of a processor chain, expressed as a temporal logic formula involving the state variables that represent the inputs and the outputs of the chain. For example, liveness is a property stating that a pipeline may always output one more event in the future; if p_b is the Boolean variable associated to the chain's output pipe, this can be expressed in CTL as $\mathbf{AG}\,\mathbf{EF}\,p_b$. Bounded liveness is a variant of liveness stipulating that the processor pipeline never remains silent for more than k successive computation steps. This stronger condition is written as an LTL property $(\mathbf{G}\,(p_b \vee \mathbf{X}\,(p_b \vee \dots)))$.

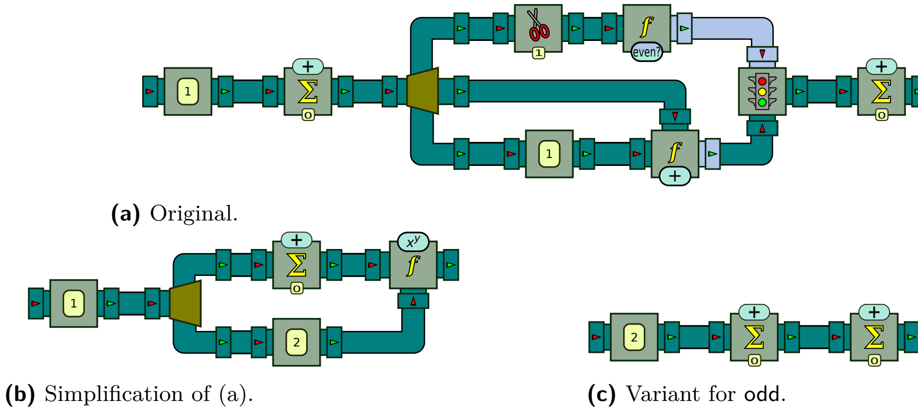
4.1 Implementation Comparison

We shall now mention a few other, less trivial applications that rest on model checking. A first possibility is to check that two pipelines perform the same computation. The basic setup for doing so is illustrated in Figure 4a. If p_c and p_b represent the pair of variables corresponding to the output pipe of π, and p'_c and p'_b represent the pair of variables corresponding to the output pipe of π', the equivalence between the two implementations can be expressed as the LTL formula $\mathbf{G}\,(p_b = p'_b \wedge (p_b \rightarrow (p_c = p'_c)))$. This is what we call *stepwise equivalence*: $\pi(\vec{v}) = \pi'(\vec{v})$ for every stream vector $\vec{v}$; that is, π and π' produce events at the same time, and the same event is produced in such a case. In case the equivalence is not verified, the model



(a) Step-wise equivalence. **(b)** Sequence equivalence.

Figure 4 Basic setup for comparing two implementations π and π' of the same pipeline.



(a) Original.

(b) Simplification of (a). **(c)** Variant for odd.

Figure 5 A processing pipeline and two simplified versions.

checker produces a counter-example which shows a possible input violating the condition. This can be used by the developer to help pinpoint the cause of the discrepancy and fix the issue. However, this setup is not restricted to strict equality. For example, if π and π' represent pipelines evaluating a Boolean condition on an input stream, one could verify that π' is a (stepwise) conservative approximation of π by checking that $\mathbf{G}\,(p_b = p'_b \wedge (p_b \rightarrow (p'_c \rightarrow p_c)))$.

There are a few situations where comparing two pipelines can be useful. First, one can check that a simplification applied to an original pipeline does not disturb its operation. For instance, Figure 5a shows a relatively convoluted pipeline, formed of 8 processors and 12 pipes; examining its operation, one can discover that this computation actually amounts to producing sequence of square numbers $(1, 4, 9, 16, \dots)$.[2] In contrast, Figure 5b shows a much simpler chain of processors performing this computation formed of 4 processors and 7 pipes. In order to make sure that the two are actually equivalent, they could be plugged in place of π and π' in Figure 4a.

Step-wise equivalence imposes that π and π' produce output events at the same time: at each computation step, they either both remain silent, or they both produce the same output. A looser condition, called *sequence equivalence*, asserts that π and π' produce the same sequence of events, discarding in each any computation step where no event is produced. This condition can be stated formally as the fact that for any stream vector $\vec{v}$, either $\pi(\vec{v}) \preceq \pi'(\vec{v})$ or $\pi'(\vec{v}) \preceq \pi(\vec{v})$. Such a property is hard to express directly in temporal logic: one must take into account the fact that either π or π' may not produce an output at a given computation step, and then keep track of the relative offset between the output values of π and π'. However, it can be easily verified (within the bounds for Q) by modifying

[2] The rightmost processor is given the sequence of odd numbers, whose sum for n terms is n^2.

the original setup to the one shown in Figure 4b. This time, the outputs of π and π' are piped into a binary processor that evaluates equality between its inputs. By virtue of the operation of processors, the events produced by π and π' will be buffered until two values can be compared on each input. This has for effect that events at matching positions in both output streams will be compared for equality, regardless of the number of computation steps required to produce them.

Using this setup, it is possible to discover that the pipelines of Figures 5a and 5b are sequence equivalent, but are *not* step-wise equivalent. Indeed, although both produce the same stream of output values, in the case of the first pipeline, one must push *two* input events in order to receive the first output event (1). In contrast, the pipeline of Figure 5b produces the first output event immediately after the first push. In other words, the first output stream is $\epsilon, 1, 4, 9, \dots$, while the second one is $1, 4, 9, \dots$ The problem of finding simpler equivalents to existing pipelines is far from trivial. Case in point, if one simply replaces the `even` condition by `odd` in the pipeline of Figure 5a, the simplified pipeline becomes that of Figure 5c. Indeed, the pipeline now computes the sum of successive even numbers, which can be accomplished with only 3 processors and 4 pipes. One can see that both simplifications are largely unrelated, in terms of structure, to the original pipeline, and that even a slight change in the original layout can result in drastically different simplifications. The discovery of "dead paths" mentioned in the introduction is simply the special case where a pipeline is compared with a truncated version of itself –the truncation representing the elements of the pipeline that can be deleted.

4.2 Reasoning on Buffers

Input buffers in BeepBeep are unbounded: they are simply instances of the `Queue` interface that can be dynamically resized on demand. However, embedded systems and many resource-critical systems disallow the use of dynamically allocated memory, which entails that all data structures must be of a fixed size. Therefore, it may be desirable to verify that the memory required to evaluate a pipeline never exceeds some given threshold, which can then be used to bound the underlying data structures.

We recall that queues within a BeepBeep processor are represented by module variables q_b, which are arrays of size Q. A full queue therefore corresponds to a state where $q_b[Q] = \top$. If $\{q_{b,1}, \dots, q_{b,n}\}$ is the set of all module variables representing queue state inside a pipeline, the full queue condition becomes straightforward to express as: $\mathbf{G}\ (\bigwedge_{i=1}^{n} \neg q_{b,i}[Q])$. This property is false exactly when there exists a sequence of input events given to the pipeline that is such that a processor has one of its buffers reaching its maximum size, Q. On the contrary, if the condition is true, the buffers can safely be bounded to a size of $Q - 1$. One can even adopt an iterative approach, where the value of Q is progressively increased or decreased in order to discover the existence of a minimal value for buffer size.

There exist pipelines whose queue size cannot be bounded; one example is Figure 2a. As we have seen, its i-th output is the sum of the input events at positions $3i$ and i; therefore, when the i-th output event is produced, $3i - 1$ events are buffered in the processor's second input queue. Memory consumption grows linearly in the length of the input stream, and therefore no upper bound on buffer size will ever satisfy the previous condition. However, performing this verification with the pipeline of Figure 5a will reveal that a value of $Q \geq 1$ satisfies the condition, while $Q \geq 0$ is sufficient for the pipeline of 5b.

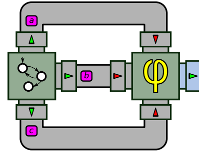


Figure 6 Model checking a system observed by a monitor.

4.3 Stream-Based Formal Verification

So far, the applications we introduced are focused on the model checking of stream pipelines themselves. However, these pipelines can also be fed as input the sequence of states of another system that executes within the model checker environment. This is illustrated in Figure 6. The leftmost box represents a standard Kripke structure K, defined with three state variables x, y and z. The values of these variables at each execution step are taken as streams of values that are fed to a BeepBeep processor pipeline φ, which in turn produces a stream of Boolean values from these input streams.

Intuitively, φ represents a *monitor* that observes a run of K and evaluates a condition on this run; the monitor emits $\perp$ at the moment where the observed run is considered to violate the property, if ever. This corresponds to a classical setup of runtime verification [8], but where both the system and the monitor observing it are simulated within a model checker. Therefore, while a monitor can only provide a verdict for a single execution at a time, its presence within the model checker makes it possible to obtain guarantees for all executions of K. If p_b and p_c are the variables modeling the output of the pipeline, asserting that the property monitored by φ is true for all runs of K simply amounts to model checking $\mathbf{G}\,(p_b \rightarrow p_c)$.

It turns out that such a setup can be useful to perform model checking of properties that would be very inconvenient to express directly as temporal logic formulas on the state variables of K. Consider the following property: at any moment, b contains the number of times a has been true in the k previous states. Expressing this as an LTL formula is possible, but results in a clumsy expression of the form: $\mathbf{G}\,(a \leftrightarrow \mathbf{X}\,(a \leftrightarrow \mathbf{X}\,(a \leftrightarrow \mathbf{X}\,b = k \vee \neg a \leftrightarrow \mathbf{X}\,b = k-1)\cdots))$. In comparison, the corresponding pipeline that monitors this property is shown in Figure 2b. In addition to being arguably simpler to express, it is also easier to modify: changing the value of k only requires a change on window width, instead of a complete rewrite of the corresponding LTL specification.

5 Experimental Results

We shall now briefly discuss an experimental evaluation of the proposed implementation, by measuring the execution time of NuXmv on a number of BeepBeep pipelines and for a sample of generic properties, with a special focus on the impact of parameters Q and N. Experiments are bundled into a LabPal [23] experimental package that is publicly available online.[3] Overall, the combinations of queue sizes, domain sizes and values of parameter k represent 122 NuSMV input models, which, combined with the set of properties to model check, correspond to 162 distinct model checking problems. Processor pipelines contain between 1 and 22 processors, resulting in Kripke structures with up to 21 distinct modules and 91 variables; for the sake of completeness, they are listed in the appendix. Experiments were run on a Intel Xeon 12-core 3.6 GHz running Ubuntu 18.04, using NuXmv version 2.0.0.

[3] `https://github.com/alexisBedard/nusmv-beepbeep-lab`

Query	No full queues	Liveness	Bounded liveness
Output if smaller than k	53	56	88
Passthrough	24	25	26
Product of 1 and k-th	387	342	1139
Sum of 1s on window	1317	1330	5231
Sum of doubles	1269	1116	11220
Sum of odds	612	526	2075
Sum of window of width 3	4842	4948	15641

(a) Verification time for various properties.

Queue size	Sum of odds	Sum of 1s on window	Pass-through	Sum of doubles	Product of 1 and k-th	Sum of window of width 3	Output if smaller than k
1	719	4761	23	175	78	23216	49
2	2992	4525	23	1694	206	23385	58
3	2249	4513	24	17012	823	22901	67
4	8207	4458	23	95619	4158	23018	95

(b) Impact of queue size on verification.

Query	Stepwise equivalence	Sequence equivalence
Passthrough	26	34
Passthrough vs delay comparison	31	44
Product of 1 and k-th	176	450
Sum of 1s on window	2085	10868
Sum of odds	1259	1423
Sum of window of width 3	468	10966
Window sum of 2 comparison	488	963
Window sum of 3 comparison	16107	116959

(c) Implementation comparison.

Figure 7 A summary of experimental results. All table entries are in milliseconds.

A first experiment measures the relative time taken to evaluate the properties discussed in Section 4 on the same processor pipeline and the same values of Q and N. These results are plotted in Table 7a. These results show that formal guarantees on pipelines can indeed be checked in reasonable time; one can see that for all processor chains, bounded liveness checking dominates verification time.

Table 7b shows the impact of queue size Q for all pipelines, on the property *No full queues*. It shows, unsurprisingly, that verification time grows exponentially for pipelines having queues, while it remains constant for processor pipelines where no queuing of events ever happens. In the case where the property is false, we observed that NuXmv indeed provides a counter-example input stream that results in one of the processors filling one of its queues; moreover, for all counter-examples we examined, this stream is of the shortest possible length for this to happen. Although not shown, a similar behavior has been observed for the impact of parameter N. Over all models, maximum verification time is 275057 ms, for the pipeline *Sum of window of width 3* on the property *Liveness*.

Finally, Table 7c shows the time taken to verify implementation comparison. This is done here by putting two copies of the same pipeline side by side; in each case, the model checker is asked to verify that they are either step-wise or sequence equivalent. Two other pipelines have also been added, which compare two different implementations of the same pipeline (a sum over a sliding window). These experimental results reveal that step-wise equivalence, although a stronger condition than sequence equivalence, is actually easier to verify.

6 Related Work

Stream processing has been the subject of various formal frameworks and implementations in the recent past. We end this paper by providing a broad overview of existing works related to the verification of properties on streams.

6.1 Stream Processing Engines

A variety of stream processing software and theoretical frameworks have been developed over the years, which all differ in a number of dimensions. For example, TelegraphCQ [13] was built to fix the problem of continuous stream of data coming from networked environments; it shares similarities with the earlier STREAM system [7]. SASE [37] was brought as a solution to meet the needs of a range of RFID-enabled monitoring applications. On its side, Siddhi [33] focuses

on the multi-threading aspect of evaluating CEP queries. Among other popular software, we shall also mention Borealis [5], Cayuga [12], StreamBase SQL [3], StreamInsight [26], and VoltDB [4]. Recently, stream processing engines have started adopting an SQL-like declarative language called the Event Processing Language (EPL) [9]; such systems include Esper [2] and Apache Flink [1]. Despite this large number of concrete implementations, few of them have a formally-defined semantics, or have been studied under the angle of static verification. This absence of formal grounds has for consequence that establishing properties of computations made using these systems is difficult. The use of a model checker has been suggested to solve the scheduling problem in a Synchronous Data Flow (SDF) pipeline [31]. However, to the best of our knowledge, we are not aware of the use of model checking to establish formal properties of the pipeline itself, down at the individual event and buffer level.

We finally mention Lola [16], a formal stream processing language where new event streams are created by combining existing streams through stream operators, and where complex processing is achieved by systems of equations on stream variables. In this context, a system of equations is said to be *efficiently monitorable* when the worst-case memory requirement for the online evaluation algorithm is constant in the length of the input stream; a sufficient condition is demonstrated, by calculating a dependency graph between streams and showing that it contains no cycle of positive weights. It has been shown that features of this language can be reproduced by composing appropriate BeepBeep processor pipelines [22]; therefore, our proposed approach can be seen as a means to indirectly verify properties on Lola models. Section 4.2 has also shown that a constant-memory requirement can be established through model checking on BeepBeep pipelines.

6.2 Formal Models

Since some of these tools are designed as independent computation units passing events downstream in a pipeline, it is somewhat sensible to compare them to formal models of finite-state systems that communicate with each other. Input-output automata [29] are a model of computation in asynchronous distributed networks, composed of a disjoint set of inputs, outputs, and internal actions. The theory of communicating automata (CA) considers networks of such units exchanging messages between each other. In this context, the reachability problem consists of determining if there exists a sequence of interactions in a given network such that each automaton can reaches an internal final state from an initial state [32].

Compared to the BeepBeep event stream processing engine that is the focus of our work, CAs present some important differences. First, CAs are asynchronous models, while BeepBeep is strictly synchronous. Second, CAs define concurrent systems. Finally, CAs use unbounded channels, while BeepBeep's computation units may only send and receive one and only one message at each computation step. A further line of research concentrates on the impact of *lossy channels*, where a message sent by a CA may or may not reach its destination point [6]. Although this problem is relevant in situations involving actual point-to-point (e.g. network) communications, the message exchanges in ESP systems is typically internal through shared memory, where such an issue is much less important.

Modular Petri nets is another model in which individual modules interact via shared places and transitions [14]. Previous work combined Linear Time Logic with modular Petri nets to show how LTL-X model checking can be done on the synchronization graph [27]: the goal is to find an illegal execution of the modular net by synchronizing the Büchi automaton with the visible transitions. Similarly, some works described how model checking can be

using the net unfoldings approach [17, 18, 30], which use partial order techniques to verify concurrent and distributed system. Communicating Transaction Processes (CTP), based on Message Sequence Charts (MSCs), is another model of computation describing a network of communicating processes interacting via common action labels [35]. Essentially, a CTP is a Petri net where the places are the states inside each process. This transition system models non-atomic inter-process communications (event structure) and describes intra-process control flow (transaction scheme); in other words, a process interacts with another before performing some internal computation. The problem of whether a CTP satisfies a specification expressed as a Live Sequence Chart is then discussed.

In comparison to these approaches, BeepBeep processor pipelines are simpler, as communications are unidirectional, and are always performed in global transitions for the whole pipeline at once. However, they allow for more complex forms of computation, such as sliding windows and aggregation, which cannot easily be modeled as PNs. In addition, while the previous works focus on the verification of reachability properties or compliance with a predefined pattern of message exchanges between computational units, our approach is more interested in the verification of input/output relationships of the global pipeline, and not on the intermediate events that are passed within it.

7 Conclusion and Discussion

This paper has shown how a stream processing pipeline can be turned into a finite-state model that simulates its execution, which makes it amenable to the model checking of properties on the original chain. Although the proposed work focuses on the BeepBeep event stream engine, the set of basic operators it handles are common to a large variety of other stream processors. Worthy of mention is that this set includes aggregation and sliding windows; to the best of our knowledge, the formal verification of such expressive types of computations is being studied here for the first time. The specifics of this translation, however, remains system-dependent.

This modeling presents a number of limitations that should be addressed in future work. First are constraints inherent to the underling model checker: only scalar data types are currently handled, whereas a few BeepBeep processors operate on richer types (character strings and variable-sized associative maps) and had to be left out. Our model also supports processors that produce at most one output event for each input event tuple received, which excludes yet a few more processors. Input queues must be given the same (finite) size across all processors of a chain, which makes it impossible to reason about queue size for individual processors.

In counterpart, the paper has shown through a few examples the potential applications that are opened by the model checking of stream pipelines. In addition to straightforward invariants, we shall mention the simplification of processor chains, the calculation of minimum buffer size, and the verification of equivalence between two implementations. The "monitor within a model checker" concept introduced in Section 4.3 especially warrants further scrutiny, as it presents the potential to easily verify properties of a system that would be cumbersome to write directly using temporal logic.

Since the NuXmv model is based on the formal semantics of each processor, it also becomes possible to use a combination of testing and model checking to assess whether the actual Java code implementing each processor is faithful to that semantics. Finally, although the lossy channel problem has not been addressed, it would be easy to include it in our model by having pipes between processors non-deterministically "leak" events, and track the consequences of such an action.

References

1 Apache Flink. URL: https://flink.apache.org, Accessed April 26th, 2021.

2 Esper. URL: http://espertech.com.

3 StreamBase SQL. URL: http://streambase.com.

4 VoltDB. URL: http://voltdb.com.

5 Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, pages 277–289, 2005. URL: http://www.cidrdb.org/cidr2005/papers/P23.pdf.

6 Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Inf. Comput.*, 127(2):91–101, 1996. doi:10.1006/inco.1996.0053.

7 A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004. URL: http://ilpubs.stanford.edu:8090/641/.

8 Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2018. doi:10.1007/978-3-319-75632-5_1.

9 Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. One SQL to rule them all - an efficient and syntactically idiomatic approach to management of streams and tables. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1757–1772. ACM, 2019. doi:10.1145/3299869.3314040.

10 Quentin Betti, Raphaël Khoury, Sylvain Hallé, and Benoît Montreuil. Improving hyper-connected logistics with blockchains and smart contracts. *IT Prof.*, 21(4):25–32, 2019. doi:10.1109/MITP.2019.2912135.

11 Mohamed Recem Boussaha, Raphaël Khoury, and Sylvain Hallé. Monitoring of security properties using BeepBeep. In Abdessamad Imine, José M. Fernandez, Jean-Yves Marion, Luigi Logrippo, and Joaquín García-Alfaro, editors, *FPS*, volume 10723 of *Lecture Notes in Computer Science*, pages 160–169. Springer, 2017. doi:10.1007/978-3-319-75650-9_11.

12 Lars Brenna, Johannes Gehrke, Mingsheng Hong, and Dag Johansen. Distributed event stream processing with non-deterministic finite automata. In Aniruddha S. Gokhale and Douglas C. Schmidt, editors, *DEBS*. ACM, 2009. doi:10.1145/1619258.1619263.

13 Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003. URL: http://www-db.cs.wisc.edu/cidr/cidr2003/program/p24.pdf.

14 Søren Christensen and Laure Petrucci. Modular analysis of petri nets. *Comput. J.*, 43(3):224–242, 2000. doi:10.1093/comjnl/43.3.224.

15 Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model checker. *Int. J. Softw. Tools Technol. Transf.*, 2(4):410–425, 2000. doi:10.1007/s100090050046.

16 Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*, pages 166–174. IEEE Computer Society, 2005. doi:10.1109/TIME.2005.26.

17 Javier Esparza and Keijo Heljanko. A new unfolding approach to LTL model checking. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *ICALP*, volume 1853 of *Lecture Notes in Computer Science*, pages 475–486. Springer, 2000. doi:10.1007/3-540-45022-X_40.

18 Javier Esparza and Keijo Heljanko. Implementing LTL model checking with net unfoldings. In Matthew B. Dwyer, editor, *SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2001. doi:10.1007/3-540-45139-0_4.

19 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

20 Sylvain Hallé. When RV meets CEP. In Yliès Falcone and César Sánchez, editors, *RV*, volume 10012 of *Lecture Notes in Computer Science*, pages 68–91. Springer, 2016. doi:10.1007/978-3-319-46982-9_6.

21 Sylvain Hallé, Sébastien Gaboury, and Bruno Bouchard. Activity recognition through complex event processing: First findings. In Bruno Bouchard, Sylvain Giroux, Abdenour Bouzouane, and Sébastien Gaboury, editors, *Artificial Intelligence Applied to Assistive Technologies and Smart Environments, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016*, volume WS-16-01 of *AAAI Workshops*. AAAI Press, 2016. URL: http://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12561.

22 Sylvain Hallé and Raphaël Khoury. Writing domain-specific languages for BeepBeep. In Christian Colombo and Martin Leucker, editors, *RV*, volume 11237 of *Lecture Notes in Computer Science*, pages 447–457. Springer, 2018. doi:10.1007/978-3-030-03769-7_27.

23 Sylvain Hallé, Raphaël Khoury, and Mewena Awesso. Streamlining the inclusion of computer experiments in a research paper. *Computer*, 51(11):78–89, 2018. doi:10.1109/MC.2018.2876075.

24 Sylvain Hallé. *Event Stream Processing With BeepBeep 3: Log Crunching and Analysis Made Easy*. Presses de l'Université du Québec, 2018.

25 Raphaël Khoury, Sylvain Hallé, and Omar Waldmann. Execution trace analysis using LTL-FO+. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA*, volume 9953 of *Lecture Notes in Computer Science*, pages 356–362, 2016. doi:10.1007/978-3-319-47169-3_26.

26 Ramkumar Krishnan, Jonathan Goldstein, and Alex Raizman. A hitchhiker's guide to StreamInsight queries, version 2.1, 2012. URL: http://support.sas.com/documentation/onlinedoc/dfdmstudio/2.4/dfU_ELRG.pdf.

27 Timo Latvala and Marko Mäkelä. LTL model checking for modular petri nets. In Jordi Cortadella and Wolfgang Reisig, editors, *ICATPN*, volume 3099 of *Lecture Notes in Computer Science*, pages 298–311. Springer, 2004. doi:10.1007/978-3-540-27793-4_17.

28 David C. Luckham. *The power of events – An introduction to complex event processing in distributed enterprise systems*. ACM, 2005.

29 Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In Fred B. Schneider, editor, *PODC*, pages 137–151. ACM, 1987. doi:10.1145/41840.41852.

30 Agnes Madalinski and Victor Khomenko. Predictability verification with parallel ltl-x model checking based on petri net unfoldings. *IFAC Proceedings Volumes*, 45(20):1232–1237, 2012.

31 Avinash Malik and David Gregg. Orchestrating stream graphs using model checking. *ACM Trans. Archit. Code Optim.*, 10(3):19:1–19:25, 2013. doi:10.1145/2512435.

32 Anca Muscholl. Analysis of communicating automata. In Adrian-Horia Dediu, Henning Fernau, and Carlos Martín-Vide, editors, *LATA*, volume 6031 of *Lecture Notes in Computer Science*, pages 50–57. Springer, 2010. doi:10.1007/978-3-642-13089-2_4.

33 Srinath Perera, Sriskandarajah Suhothayan, Mohanadarshan Vivekanandalingam, Paul Fremantle, and Sanjiva Weerawarana. Solving the grand challenge using an opensource CEP engine. In Umesh Bellur and Ravi Kothari, editors, *DEBS*, pages 288–293. ACM, 2014. doi:10.1145/2611286.2611331.

34 Stefanie Rinderle-Ma and Sonja Kabicher-Fuchs. An indexing technique for compliance checking and maintenance in large process and rule repositories. *Enterp. Model. Inf. Syst. Archit. Int. J. Concept. Model.*, 11:2:1–2:24, 2016. doi:10.18417/emisa.11.2.

35 Abhik Roychoudhury and P. S. Thiagarajan. Communicating transaction processes: An MSC-based model of computation for reactive embedded systems. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 789–818. Springer, 2003. doi:10.1007/978-3-540-27755-2_22.

36 Simon Varvaressos, Kim Lavoie, Sébastien Gaboury, and Sylvain Hallé. Automated bug finding in video games: A case study for runtime monitoring. *Computers in Entertainment*, 15(1):1:1–1:28, 2017. doi:10.1145/2700529.

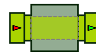
37 Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *SIGMOD Conference*, pages 407–418. ACM, 2006. doi:10.1145/1142473.1142520.

A Appendix: Pipelines

Here are illustrated the pipelines included in the experiments.

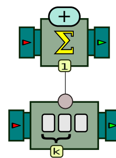
Passthrough

The passthrough is a processor that simply outputs whatever it receives as its input.



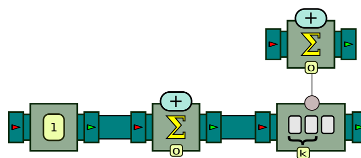
Sum on window of width 3

The pipeline sums three successive input events.



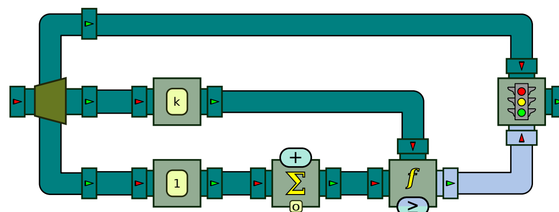
Sum of 1s on window

This pipeline turns every event into a 1, and then sums these values over a window of width k. Therefore, it always returns k as its output events.



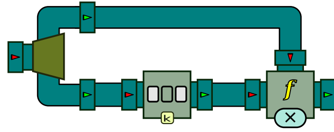
Output if smaller than k

This pipeline turns every event into a 1, and then outputs these values only if they are smaller than some constant k. As a result, it only outputs k events, and remains silent after that.



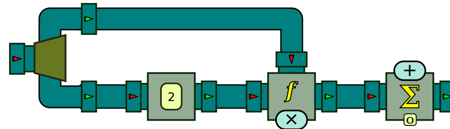
Product of 1 and k-th

This pipeline corresponds to a variant of Figure 2a, with multiplication replacing addition.



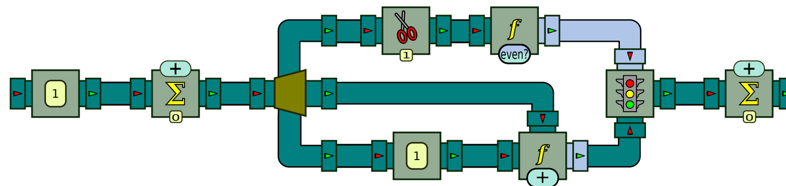
Sum of doubles

This chain multiplies an input event by two, and sums the resulting stream.



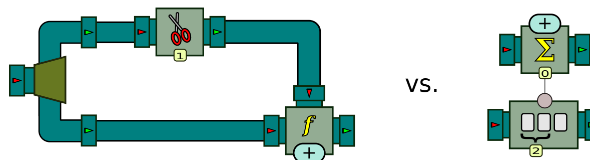
Sum of odds

This example was discussed as Figure 5a.



Compare window sum of 2

This model compares two implementations of the sum over a sliding window of width 2.



Compare window sum of 3

This model compares two implementations of the sum over a sliding window of width 3.