Dissertation Proposal:
Reliable Programming for Stream Processing Systems

Caleb Stanford

May 2021

Abstract

The sheer scale of today's data processing needs, the emergence of the internet of things, and the increasing transience of data have led to the popularity of specialized software systems centered around requirements for high-throughput, distributed, low-latency computation. Despite their widespread adoption, *distributed stream processing systems (DSPS)*, such as Apache Flink, Timely Dataflow, and Apache Spark Streaming, remain challenging for end users due to unexpected behavior at runtime. In particular, formal guarantees are absent, resulting in correctness and performance bugs.

We propose new programming abstractions towards *reliable* programming for DSPS. To achieve reliability, our techniques span *semantics, testing, safe distribution,* and *monitoring* for distributed stream processing programs. To bring these ideas to end users, we report on the development of the DiffStream and DGSStream tools. Finally, as future work, we envision a library providing *verification* of programs built using our abstractions.

Contents

1 Introduction

Today, data is produced at an overwhelming rate that cannot be processed by traditional methods. For example, Cisco has estimated in its annual white paper that data produced by people, machines, and things is around 500 zettabytes, in contrast to a much smaller volume of data that can be feasibly stored [40]. At the same time, an increasing number of IoT devices [61, 20] and cloud computing applications are joining the internet. Industrial practice has accordingly recognized the demand for a new approach to computing where data is transient, distributed, and temporally structured. *Distributed stream processing systems (DSPS)* have emerged as a popular programming solution. Popular DSPS include Apache Flink [30, 22], Timely Dataflow [27, 53], and Apache Spark Streaming [32, 66]; a selection of these and other systems is included in Table 1.

To understand the emergence of DSPS and why traditional software infrastructure is not sufficient, note that traditional software is often based on the assumption that critical data can be stored and then processed later. For example, much of large-scale data analytics relies on processing data in large *batches* (e.g. training a machine learning model daily), such as via MapReduce jobs [25]. While batch processing does take advantage of distributed computing resources, it does not take advantage of the transient and temporal structure of data. Data is not transient because it must be stored first, which introduces costs due to batch sizes and data movement. In practice due to these costs, most data traveling over the internet and processed by cloud services is either not stored or not harvested to its full potential. Additionally, the temporal structure of data is lost in batches, which divide data at arbitrary boundaries in time. For example, a machine learning model that might benefit from continuous updates is instead trained only on the data from yesterday. The clear solution to these challenges is to write programs that compute over data in real time. DSPS offer a software framework for writing such programs, where data processing logic is defined in a platform-independent manner, then deployed as a distributed application over many nodes. DSPS performance is measured in terms of *latency* and *throughput*; concretely, stream processing platforms aim for latency in the milliseconds and throughput in tens of thousands of events per process per node.

Despite their popularity and high performance, programming in DSPS remains difficult for end users. This has been investigated by software engineering researchers and surveys [36, 29, 64], and many of the identified challenges are specific to the big data setting. For instance, the scale of data makes it difficult to identify faults due to behavior on a single input tuple. Another source of bugs is that automatic distribution of stream processing applications is unfortunately not semantics-preserving [65, 60, 38], which results in nondeterminism due to ordering of distributed events. Such nondeterminism can lead to bugs that are difficult to identify and reproduce. In practice, these challenges are partially addressed by rigorous testing, data validation, and offloading of key functionality such as reliable storage to external services, but it would be preferable for DSPS systems to provide more robust behavior in the first place. In more detail, the following are some of the biggest programming concerns in today's DSPS applications:

- *Lack of semantics.* There exists no widely accepted common semantics for distributed stream processing. DSPS applications are always written as dataflow graphs, and there are other common elements, but beyond this different APIs make different choices. For example, Flink's API assumes that data arrives in-order per-key, whereas Timely's API does not offer this guarantee. Specific constructs then come with other differences, for example: whether watermarks (indicating stream progress) are explicitly available or implicit; and whether side effects are allowed in an operator or whether the system makes no guarantees in the presence of side effects.

- *Nondeterminism due to event ordering.* DSPS applications achieve parallelism through *sharding*, which means that operators in the dataflow graph are replicated, and each replica (or shard) processes a subset of the input stream. However, partitioning of the input stream and collecting of results introduces the possibility of data reordering. In practice, many operators are not affected by such reordering: e.g., commutative, associative reduce operations or stateless maps and filters. However in cases where ordering matters, subtle bugs arise that are not visible at the application level.

- *Low-level state management.* DSPS applications are built under the assumption that users should not have to write state management logic on their own, intsead relying on predefined dataflow operators (e.g. maps, filters, aggregation, windowing, and SQL query libraries). In practice, however, some streaming operations require custom logic. Examples of more complex logic include linear interpolation (fill in missing input data in a temporally dependent manner), machine learning operators (aggregate and update a statistical model), and event-dependent windows (form a window with data-dependent start and closing times). Because of the ubiquity of such manual state management tasks, popular APIs (including Storm, Flink, and Timely) allow users to program operators manually, e.g. providing a state type, an initial state, and an update function for each input tuple. However, such operators are difficult to program because they must work under the sharding mentioned above. Additionally, if the operator requires interaction with an external service or has side effects (e.g. querying a database), state update logic has to be tolerant to unexpected behavior in case of node failures or network communication; yet in practice, users simply ignore these concerns and this results in applications that may crash unexpectedly on a failure. As a result of these concerns, DSPS developers and researchers generally agree that better high-level query constructs are needed that alleviate the need for low-level programming.

- *Manual parallel programming.* Related to low-level state management, DSPS also promise a programming framework where users should not have to parallelize their application themselves. However, in practice, many applications are highly difficult to parallelize and require low-level constructs: for example, a *broadcast* construct is used to broadcast important shared or global state to other nodes. Such parallel programming is highly prone to correctness bugs.

- *Unpredictable performance.* Because operators and input data are partitioned by the system, users do not describe explicitly how to partition them. However, DSPS do not offer any concrete guarantees about the throughput or latency of the runtime. In particular, unexpected performance bugs arise in the case that partitioning is not efficient. For instance, in an example application processing an input stream of webpage views, if most of the views come from the same website, partitioning by key fails and results in a performance bottleneck. To address these bugs requires that we have more reliable performance guarantees. Ideally, application performance could be estimated statically or in conjunction with runtime profiling.

In this thesis, we address these limitations by providing abstractions and tools to make programming DSPS more reliable. To put DSPS on formal foundations, we first propose *partial-order semantics* for data streams. We argue that events in the system should be viewed as a partially ordered set, where events at different nodes should be viewed as unordered if their order is not guaranteed by the system. The partial order viewpoint, called *dependence relations* and inspired by foundational work in concurrency theory, forms a common semantic framework for much of the work in this thesis. We define two typing disciplines for partially ordered streams: data-trace types [19, 50], and synchronization schemas [17]. In addition to static typing guarantees [50], we use the partial order viewpoint for differential testing [42] and for to enable *safe* distribution, i.e. distribution with correctness guarantees [43]. The motivation for the former is that while static types are useful for new applications, existing applications are not developed with the new framework and so can benefit from runtime testing to identify bugs due to output even reordering. The motivation for the latter work is to define a parallel programming framework appropriate to DSPS where distribution is semantics-preserving rather than potentially semantics-breaking.

In the second part of the thesis, we aim to offer provable guarantees about performance: in particular considering finite-state models of stream processing systems [3, 18, 1]. In particular, we target *runtime monitoring* applications over data streams, and we show how to define and compile runtime monitors with provable performance. These can be used either to monitor DSPS applications on the side, or to define DSPS operators themselves. While the existing work has been used for compilation of high-level query languages on a single machine with space and time bounds [18, 15, 48], the primary direction for future work here is to adapt to the *distributed setting* and show similar performance guarantees there.

For the remaining future work, we target an implementation of the synchronization schemas [17] and data transducer [18] abstractions on top of Timely Dataflow [27, 53] in Rust [57]. Timely is a good choice because it offers a semantically sound low-level dataflow representation, and we aim to leverage Rust's type system for compile-time guarantees, while generating external verification conditions to prove user programs correct.

1.1 Contributions

In summary, we make the following (completed and planned) contributions:

- *Semantics:* We propose *partial order semantics* for DSPS, through the data-trace types [50] and synchronization schemas [17] typing disciplines. We show how viewing streams as partially ordered allows for deterministic semantics, and show how well-typed programming disciplines can be developed, leading to well-defined DSPS programming libraries. (Section 2)

- *Testing:* To detect bugs due to nondeterminism in existing DSPS applications, we propose DiffStream, a differential testing framework [42]. In particular, we leverage the partial order viewpoint to specify ordering requirements, and we test for violations at runtime. (Section 3)

- *Safe distribution:* For more complex application logic where currently error-prone low-level state management is used, we propose *dependency-guided synchronization*, a programming framework for *semantics-preserving* distribution [43]. A key aspect of this work is the ability to program synchronization between distributed nodes, which is not supported by most existing systems, while still ensuring safety and allowing automatic distribution of streaming applications. (Section 4)

- *Monitoring:* Towards streaming applications with predictable performance, we propose data transducers, a monitoring formalism and state-machine based intermediate representation [18]. Our formalism is compositional, enabling compilation of high-level monitoring queries with provable performance bounds. (Section 5)

- *Verification:* As future work, we propose a verified streaming library over partially ordered streams, incorporating ideas from synchronization schemas [17] and data transducers [18]. The library enables a high-level query language which is well-typed, where

System	Year	Stable Release	Active?	Questions on StackOverflow (as of 2021-04-29)
Aurora	2003 [11]	2003 [55]	No	–
Borealis	2005 [10]	2008 [56]	No	–
APACHE STORM™ Distributed · Resilient · Real-time	2011 [58]	2020 [33]	Yes	2548
Apache Flink [30, 22]	2011	2021	Yes	5345
Google MillWheel	2013 [12]	–	No	–
Spark (Apache Spark Streaming)	2013 [66]	2021 [32]	Yes	5222
samza [31, 54]	2013	2021	Yes	85
Timely Dataflow	2013 [53]	2021 [27]	Yes	–
Apache HERON [63, 44]	2015	2021	Yes	41

Table 1: A selection of major distributed stream processing systems.

typing properties are checked partly statically and partly by generating external verification conditions. (Section 6)

2 Semantics

In this section we introduce partially ordered semantics for DSPS applications. DSPS applications suffer from unexpected nondeterminism due to out-of-order data, where some operators are not written in an order-independent (or "commutative") fashion. We argue that streams should be viewed as *partially ordered*, and ordering requirements should be defined by using *types* to describe these partial orders on concrete streams. We propose *data-trace types* [19, 50] and *synchronization schemas* [17], to solve the problem of encoding ordering requirements as types. Synchronization schemas generalize the data-trace types framework to the symbolic setting with key-based partitionining and focuses on series-parallel streams [17].

Concretely, a type describes a set of possible events that could arrive in a stream, together with a dependence relation which induces a partial order on the set of events. Dependence relations date back to Mazurkiewicz traces, studied in concurrency theory to model distributed sets of events [52, 26]. These give a kind of semantics to stream processing programs (what it means for them to be correct) in the presence of out-of-order data.

2.1 Data-Trace Types

A ***data type*** $A = (\Sigma, (T_\sigma)_{\sigma \in \Sigma})$ consists of a potentially infinite *tag alphabet* Σ and a value type T_σ for every tag $\sigma \in \Sigma$. The set of *elements* of type A, or ***data items***, is equal to $\{(\sigma, d) \mid \sigma \in \Sigma \text{ and } d \in T_\sigma\}$, which we will also denote by A. The set of *sequences* over A is denoted as A^*. A ***dependence relation*** on a tag alphabet Σ is a symmetric binary relation on Σ. We say that the tags σ, τ are *independent* (w.r.t. a dependence relation D) if $(\sigma, \tau) \notin D$. For a data type $A = (\Sigma, (T_\sigma)_{\sigma \in \Sigma})$ and a dependence relation D on Σ, we define the dependence relation that is induced on A by D as $\{((\sigma, d), (\sigma', d')) \in A \times A \mid (\sigma, \sigma') \in D\}$, which we will also denote by D. Define $\equiv_D$ to be the smallest congruence (w.r.t. sequence concatenation) on A^* containing $\{(ab, ba) \in A^* \times A^* \mid (a, b) \notin D\}$. Informally, two sequences are equivalent w.r.t. $\equiv_D$ if one can be obtained from the other by repeatedly commuting adjacent items with independent tags.

Example 2.1. Suppose we want to process a stream that consists of sensor measurements and special symbols that indicate the end of a one-second interval. The data type for this input stream involves the tags $\Sigma = \{\mathtt{M}, \mathtt{\#}\}$, where `M` indicates a sensor measurement and `#` is an end-of-second marker. The value sets for these tags are $T_{\mathtt{M}} = \mathbb{N}$ (the natural numbers), and $T_{\mathtt{\#}} = \mathtt{Ut}$ is the unit type (singleton). So, the data type $A = (\Sigma, T_{\mathtt{M}}, T_{\mathtt{\#}})$ contains measurements $(\mathtt{M}, d)$, where d is a natural number, and the end-of-second symbol `#`.

The dependence relation $D = \{(\mathtt{M}, \mathtt{\#}), (\mathtt{\#}, \mathtt{M}), (\mathtt{\#}, \mathtt{\#})\}$ says that the tag `M` is independent of itself, and therefore consecutive `M`-tagged items are considered unordered. For example, $(\mathtt{M}, 5)\ (\mathtt{M}, 5)\ (\mathtt{M}, 8)\ \mathtt{\#}\ (\mathtt{M}, 9)$ and $(\mathtt{M}, 8)\ (\mathtt{M}, 5)\ (\mathtt{M}, 5)\ \mathtt{\#}\ (\mathtt{M}, 9)$ are equivalent w.r.t. $\equiv_D$.

A ***data-trace type*** is a pair $X = (A, D)$, where A is a data type and D is a dependence relation on the tag alphabet of A. A ***data trace*** of type X is a congruence class of the relation $\equiv_D$. We also write X to denote the set of data traces of type X. Since the equivalence $\equiv_D$ is a congruence w.r.t. sequence concatenation, the operation of concatenation is also

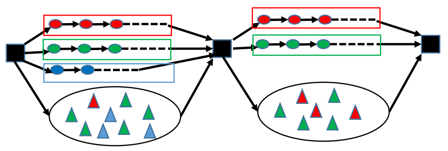


Figure 1: Illustrative series-parallel stream.

well-defined on data traces: $[u] \cdot [v] = [uv]$ for sequences u and v, where $[u]$ is the congruence class of u. We define the relation $\leq$ on the data traces of X as a generalization of the prefix partial order on sequences: for data traces $\mathbf{u}$ and $\mathbf{v}$ of type X, $\mathbf{u} \leq \mathbf{v}$ iff there are $u \in \mathbf{u}$ and $v \in \mathbf{v}$ s.t. $u \leq v$ (i.e., u is a prefix of v). The relation $\leq$ on data traces of a fixed type is a partial order. Since it generalizes the prefix order on sequences (when the congruence classes of $\equiv_D$ are singleton sets), we will call $\leq$ the *prefix order* on data traces.

Example 2.2 (Data Traces)**.** Consider the data-trace type $X = (A, D)$, where A and D are given in Example 2.1. A data trace of X can be represented as a sequence of multisets (bags) of natural numbers and visualized as a partial order on that multiset. The trace corresponding to the sequence of data items $(\mathtt{M}, 5)$ $(\mathtt{M}, 7)$ # $(\mathtt{M}, 9)$ $(\mathtt{M}, 8)$ $(\mathtt{M}, 9)$ # $(\mathtt{M}, 6)$ is visualized as:

$$\begin{tikzcd}
(\mathtt{M},5) \arrow[dr, dash] & & (\mathtt{M},9) \arrow[dr, dash] & & \\
& \# \arrow[ur, dash] \arrow[r, dash] \arrow[dr, dash] & (\mathtt{M},8) \arrow[r, dash] & \# \arrow[r, dash] & (\mathtt{M},6) \\
(\mathtt{M},7) \arrow[ur, dash] & & (\mathtt{M},9) \arrow[ur, dash] & &
\end{tikzcd}$$

where a line from left to right indicates that the item on the right must occur after the item on the left. The end-of-second markers # separate multisets of natural numbers. So, the set of data traces of X has an isomorphic representation as the set $\mathsf{Bag}(\mathbb{N})^+$ of nonempty sequences of multisets of natural numbers. In particular, the empty sequence ϵ is represented as $\emptyset$ and the single-element sequence # is represented as $\emptyset$ $\emptyset$.

A singleton tag alphabet can be used to model sequences or multisets over a basic type of values. For the data type given by $\Sigma = \{\sigma\}$ and $T_\sigma = T$ there are two possible dependence relations for Σ, namely $\emptyset$ and $\{(\sigma, \sigma)\}$. The data traces of $(\Sigma, T, \emptyset)$ are multisets over T, which we denote as $\mathsf{Bag}(T)$, and the data traces of $(\Sigma, T, \{(\sigma, \sigma)\})$ are sequences over T.

Example 2.3 (Multiple Input/Output Channels)**.** Suppose we want to model a streaming system with multiple independent input and output channels, where the items within each channel are linearly ordered but the channels are completely independent. This is the setting of (acyclic) *Kahn Process Networks* [35] and the more restricted synchronous dataflow models [46, 21]. We introduce tags $\Sigma_{\mathtt{I}} = \{\mathtt{I}_1, \ldots, \mathtt{I}_m\}$ for m input channels, and tags $\Sigma_{\mathtt{O}} = \{\mathtt{O}_1, \ldots, \mathtt{O}_n\}$ for n output channels. The dependence relation for the input consists of all pairs $(\mathtt{I}_i, \mathtt{I}_i)$ with $i = 1, \ldots, m$. This means that for all indexes $i \neq j$ the tags $\mathtt{I}_i$ and $\mathtt{I}_j$ are independent. Similarly, the dependence relation for the output consists of all pairs $(\mathtt{O}_i, \mathtt{O}_i)$ with $i = 1, \ldots, n$. Assume that the value types associated with the input tags are $T_1, \ldots, T_m$, and the value types associated with the output tags are $U_1, \ldots, U_n$. The sets of input and output data traces are (up to a bijection) $T_1^* \times \cdots \times T_m^*$ and $U_1^* \times \cdots \times U_m^*$ respectively.

2.2 Synchronization Schemas

We start by defining some preliminary notions from relational query processing: headers, tuples, and header keys. We assume that tuples, i.e. stream elements, are of finitely many possible types, where each type is given as a relational schema or *header*.

Definition 2.4 (Headers and tuples)**.** A *header* H consists of a unique *header name* α and *fields* $\langle \alpha_i : \tau_i \rangle$, for $1 \leq i \leq n$, where each α_i is a *field name* and τ_i is a *field type*. A *tuple* x of type H, denoted $x : H$, is of the form $x = (x_1, x_2, \ldots, x_n)$, where each $x_i : \tau_i$.

When the context is clear, we identify each header H with its name α and likewise each field $\langle \alpha_i : \tau_i \rangle$ with its name α_i. We write $\alpha_i \in H$ to mean that α_i is a field of H, and $x.\alpha_i$ to denote the value of x on α_i. For a set $\mathcal{H}$ of headers, we write $x : \mathcal{H}$ if $x : H$ for some $H \in \mathcal{H}$.

Example 2.5. Consider a stream of taxi events, where each is a GPS measurement, an indication of a taxi ride begin or a ride end, or an end-of-hour synchronization marker. GPS data for each taxi is described by the header `GPS(location: pos, taxiID: int)`, where `pos` indicates the type of a GPS measurement—three dimensional coordinates, for instance. Completed ride data is described by the header `RideCompleted(rideID: int, taxiID: int, passengerID: int, cost: int)`. Finally, end-of-hour events are used to synchronize in time; these are described by the header `EndOfHour(date: date, hour: int)`.

The input stream can be viewed as a partially ordered set of events, each labeled with a tuple of data. The structure of such a stream is captured by a data type that we call *synchronization schemas*. A synchronization schema is a hierarchical forest-like structure that can be seen as an extension of database schemas.

Definition 2.6 (Synchronization schema)**.** A *synchronization schema* S is inductively defined as follows:

$$S \ ::= \ \mathrm{Bag}(\mathcal{H}) \mid \mathrm{Sync}(\mathcal{H}, S) \mid \mathrm{PartitionBy}(K, S) \mid \mathrm{Par}(S, S),$$

where K denotes a set of field names (partition keys) and $\mathcal{H}$ denotes a set of headers.

The base rule $\mathrm{Bag}(\mathcal{H})$ defines a stream corresponding to a bag of $\mathcal{H}$-events, that is, events labeled with tuples of type $\mathcal{H}$. To combine the base rule (leafs) in complex schemas we use the other three rules. $\mathrm{Sync}(\mathcal{H}, S)$ defines a parent relation between $\mathcal{H}$-events and S-events—events labeled with any of the headers appearing in the schema S, and denotes a stream consisting of a sequence of $\mathcal{H}$-events that act as synchronization markers, interspersed with sub-streams of type S. $\mathrm{PartitionBy}(K, S)$ partitions a schema S based on a set of key fields K, and denotes a set of streams of type S, indexed by the values for the keys in K. Finally, $\mathrm{Par}(S_1, S_2)$ describes a sibling relation between schemas S_1 and S_2, and corresponds to a parallel composition of streams of types S_1 and S_2.

We use the following notational abbreviations. When a set of headers consists of a single header H, we write H instead of $\{H\}$. We use $\mathrm{Seq}(\mathcal{H})$ for $\mathrm{Sync}(\mathcal{H}, \mathrm{Bag}(\varnothing))$, where $\varnothing$ is the empty set of headers, and such a schema corresponds to a totally ordered sequence of $\mathcal{H}$-events. Finally, we write $\mathrm{Par}(S_1, S_2, S_3)$ for the parallel composition of three schemas, which is short for $\mathrm{Par}(\mathrm{Par}(S_1, S_2), S_3)$ (note that parallel composition is associative).

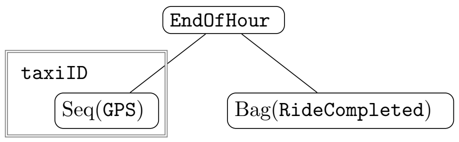


Figure 2: Example synchronization schema.

Example 2.7. We can define the schema for the input of the example in Theorem 2.5 in a bottom up fashion in the following manner. First, $S_1 = \text{PartitionBy}(\texttt{taxiID}, \text{Seq}(\texttt{GPS}))$ denotes that `GPS` events are partitioned by the key `taxiID` and are totally ordered for each taxi. Second, $S_2 = \text{Bag}(\texttt{RideCompleted})$ denotes that `RideCompleted` events are unordered, and can be considered to be a bag. Finally, $S = \text{Sync}(\texttt{EndOfHour}, \text{Par}(S_1, S_2))$ denotes that `EndOfHour` events synchronize the events in S_1 and S_2, each of which can be processed in parallel as they are independent. It is often helpful to visualize schemas as forests. Figure 2 illustrates the schema for the example. Siblings correspond to the $\text{Par}(S_1, S_2)$ constructor while the rectangular box, labeled with the key fields, corresponds to the $\text{PartitionBy}(K, S)$ constructor. A parent node with several children corresponds to the $\text{Sync}(\mathcal{H}, S)$ constructor.

We additionally require for the remainder of the document that synchronization schemas are *well-formed*, as defined below. First note that the partitioning construct $\text{PartitionBy}(K, S)$ naturally gives rise to a concept of scope for partition keys: for each partition key $k \in K$ and for each header H appearing in the schema S, we say that H is *in the scope of* k.

Definition 2.8 (Well-formed schema)**.** A *synchronization schema* S is *well-formed* if the following conditions hold: (1) no header H appears in S twice, (2) if a header H is in the scope of a partition key k, then k is a field of H, i.e. $k \in H$, and (3) if a partition schema $\text{PartitionBy}(K, S)$ is in the scope of a partition key k, then $k \notin K$.

The first condition is necessary for unambiguous parsing, while the latter two ensure that splitting on a key field is meaningful in a given context. Note that it is straightforward to check the conditions necessary for a schema to be well-formed.

2.2.1 Series-Parallel Streams

Now that we have defined synchronization schemas, which act as types, we can define a natural inductive representation of streams that are values of these types. We call these series-parallel streams, or SPS for short. We have already seen an example of such a stream in Figure 1 informally. Before we formalize the definition, consider the sequence of `GPS` events corresponding to the red taxi. The type of this sequence is $\text{Seq}(\texttt{GPS})$, but is more specialized since all these events share a common value of the field `taxiID`. We will denote such an instantiation of the schema with common key values as $\text{Seq}(\texttt{GPS})[\texttt{taxiID} = \texttt{red}]$. Such a type can be viewed as *refinement type* of the schema type $\text{Seq}(\texttt{GPS})$.

If $d : H$ and F is a subset of the fields of H, we write $d|_F$ for the restriction of d to contain only those fields in F. For a set K of partition keys, K can also be considered to be a header containing these keys as its only fields. Then, for a particular tuple v of such a header type K, for a schema S, we use $S[v]$ to denote the refinement of schema S to an instance where all the tuples are required to have key values as specified by v.

We use the following syntactic constructs to capture the structure of the desired series-parallel streams: $[x_1, x_2, \ldots, x_n]$ for a sequence (list), $\{x_1, x_2, \ldots, x_n\}$ for a bag (with standard bag equality semantics), $\langle x_1, x_2 \rangle$ for a pair of x_1 and x_2 where x_1 and x_2 are thought of as parallel instead of sequential, and $v \mapsto x$ to represent a key-indexed value.

Definition 2.9 (Series-Parallel Streams)**.** Let S be a synchronization schema. A *series-parallel stream* (SPS) $t : S[v]$ for a specific instantiation of key values $v : K$ is inductively defined as follows:

- If $d_i : \mathcal{H}$ such that $d_i|_K = v$ for $i = 1, \ldots, m$, then $t = \{d_1, \ldots d_m\}$ is an SPS of type $\mathrm{Bag}(\mathcal{H})[v]$.
- If $t_1 : S_1[v]$ and $t_2 : S_2[v]$, then $t = \langle t_1, t_2 \rangle$ is an SPS of type $\mathrm{Par}(S_1, S_2)[v]$.
- If $d_i : \mathcal{H}$ such that $d_i|_K = v$ for $i = 1, \ldots, m$, and if $t_i : S'[v]$ for $i = 0, 1, \ldots, m$, then $t = [t_0, d_1, t_1, \ldots, d_m, t_m]$ is an SPS of type $\mathrm{Sync}(\mathcal{H}, S')[v]$.
- Suppose that K' is a set of partition keys disjoint from K, and that $v'_1, v'_2, \ldots, v'_m : K'$ are *distinct* instances of key values for K'. Suppose $t_1, t_2, \ldots, t_m$ are *nonempty* streams such that $t_i : S'[v_i]$, and let $v_i : K \cup K'$ to the unique valuations such that $v_i|_K = v$ and $v_i|_{K'} = v'_i$, i.e. the extension of v'_i with the key values in v. Then $t = \{v'_1 \mapsto t_1, v'_2 \mapsto t_2, \ldots, v'_m \mapsto t_m\}$ is an SPS of type $\mathrm{PartitionBy}(K', S')[v]$.

We write $t : S$ when $K = \varnothing$ for $t : S[()]$, where $()$ is the empty tuple of type K. When $S[v]$ is clear from the context, we write $\perp : S[v]$ for the empty SPS: this abbreviates $\{\}$ for $S = \mathrm{Bag}(\mathcal{H})$, $\langle \perp, \perp \rangle$ for $S = \mathrm{Par}(S_1, S_2)$, $[\perp]$ for $S = \mathrm{Sync}(\mathcal{H}, S')$, and $\{\}$ for $S = \mathrm{PartitionBy}(K', S')$.

Example 2.10. Let us revisit the SPS shown in Figure 1 of the schema defined in Theorem 2.7. The sequence of `GPS` events of the red taxi within the first hour is the stream $t_r = [\perp, r_1, \perp, r_2, \ldots, r_n, \perp]$, where $\perp$ is the empty stream of type $\mathrm{Bag}(\varnothing)$ and $r_1, r_2, \ldots r_n$ are the corresponding events. The streams t_g and t_b of `GPS` events of the green and blue taxis, respectively, have a similar structure. The stream $t_1 = \{\texttt{red} \mapsto t_r, \texttt{green} \mapsto t_g, \texttt{blue} \mapsto t_b\}$ then captures all the `GPS` events in the first hour and is of type $S_1 = \mathrm{PartitionBy}(\texttt{taxiID}, \mathrm{Seq}(\texttt{GPS}))$. The following is the bag containing all `RideCompleted` events in the first hour, and is of type $S_2 = \mathrm{Bag}(\texttt{RideCompleted})$: $t'_1 = \{r'_1, \ldots, g'_1, \ldots, b'_1, \ldots\}$. The streams t_2 and t'_2 are analogous to the streams t_1 and t'_1, respectively, and capture the `GPS` and `RideCompleted` events in the second hour. If eoh_1, eoh_2, eoh_3 represent the first three `EndOfHour` events, then the stream $[\perp, eoh_1, \langle t_1, t'_1 \rangle, eoh_2, \langle t_2, t'_2 \rangle, eoh_3, \perp]$ represents all the events. Its type is $S = \mathrm{Sync}(\texttt{EndOfHour}, \mathrm{Par}(S_1, S_2))$.

A few remarks regarding the technical details of this definition are in order. The definition is set up so that a linear sequence of tuples over the headers appearing in a schema can be uniquely parsed (that is, interpreted) as a series-parallel stream (see Proposition 2.19). In parallel composition case, the order of the components matters, and component sub-streams may be empty. On the other hand, in key-based partitioning, the stream is defined to be a set of non-empty sub-streams, one per key value. To understand the case of nested hierarchical structure, consider the schema $\mathrm{Sync}(A, \mathrm{Sync}(B, \mathrm{Seq}(C)))$. In the corresponding stream, a C-tuple may be followed by an A-tuple without any intervening B-tuples. This requires care to make sure that a synchronizing event is able *close* all open sub-streams corresponding to its descendants.

Finally, we define *concatenation*, denoted $\circ$, of series-parallel streams. This generalizes the notion of concatenation of sequences. Intuitively, if we consider a cut through the series-parallel stream, say, shown in Figure 1, such that the left stream is closed under predecessors (and right stream is closed under successors) then concatenating the left and right substreams should give us the original stream.

Definition 2.11 (Concatenation and Prefix Ordering for SPS)**.** Let $t, u : S[v]$ be series-parallel streams over the same schema S and key valuation v. The *concatenation* $t \circ u$ is defined inductively on the structure of S:

- If $S = \mathrm{Bag}(\mathcal{H})$, $t = \{d_1, \ldots, d_m\}$, and $u = \{e_1, \ldots, e_n\}$, then
$$t \circ u = \{d_1, \ldots, d_m, e_1, \ldots, e_n\}.$$
- If we have $S = \mathrm{Sync}(\mathcal{H}, S')$, $t = [t_0, d_1, t_1, \ldots, d_m, t_m]$, and $u = [u_0, e_1, u_1, \ldots, e_n, u_n]$, then
$$t \circ u = [t_0, d_1, t_1, \cdots, d_m, (t_m \circ u_0), e_1, u_1, \cdots, e_n, u_n].$$
- If $S = \mathrm{PartitionBy}(K, S')$, then let the overlapping key values between t and u be $v_1, v_2, \ldots, v_l$, with additional keys $v'_1, v'_2, \ldots, v'_m$ in t only and $v''_1, v''_2, \ldots, v''_n$ in u only. If $t = \{v_1 \mapsto t_1, \ldots, v_l \mapsto t_l, v'_1 \mapsto t'_1, \ldots, v'_m \mapsto t'_m\}$ and $u = \{v_1 \mapsto u_1, \ldots, v_l \mapsto u_l, v''_1 \mapsto u'_1, \ldots, v''_n \mapsto u'_n\}$, then
$$\begin{aligned} t \circ u = \{\ & v_1 \mapsto t_1 \circ u_1, \ldots, v_l \mapsto t_l \circ u_l, \\ & v'_1 \mapsto t'_1, \ldots, v'_m \mapsto t'_m, \\ & v''_1 \mapsto u'_1, \ldots, v''_n \mapsto u'_n. \} \end{aligned}$$
- If $S = \mathrm{Par}(S_1, S_2)$, $t = \langle t_1, t_2 \rangle$, and $u = \langle u_1, u_2 \rangle$, then
$$t \circ u = \langle t_1 \circ u_1, t_2 \circ u_2 \rangle.$$

For $t, u : S[v]$, t is said to be a *prefix* of u, written $t \preceq u$, if there exists a series-parallel stream $t' : S[v]$ such that $t \circ t'$ equals u.

Proposition 2.12. For each type $S[v]$ and for all $t, t', t'' : S[v]$, the following hold: (1) $t \circ \bot = \bot \circ t = t$. (2) $(t \circ t') \circ t'' = t \circ (t' \circ t'')$. (3) $t \preceq t$. (4) If $t \preceq t'$ and $t' \preceq t$, then $t = t'$. (5) If $t \preceq t'$ and $t' \preceq t''$, then $t \preceq t''$.

2.2.2 Relation to Data-Trace Types

Contrasting with the series-parallel view of streams provided in the previous section, we describe here a sequential view as sequences up to equivalence. This relates the semantics of synchronization schemas to the semantics of data-trace types. To define the equivalence we first define the dependence relation on tuples derived the given synchronization schema: two tuples in the relation are said to be dependent if the order between them is important.

Definition 2.13 (Dependence Relation)**.** Let S be a synchronization schema and let $\mathbf{headers}(S)$ be all the headers appearing in S. The *dependence relation* is a binary relation on tuples of $\mathbf{headers}(S)$, written $x\, D_S\, y$ for $x, y : \mathbf{headers}(S)$, and defined inductively as follows: (i) if $S = \mathrm{Bag}(\mathcal{H})$, then D_S is the empty set; (ii) if $S = \mathrm{Sync}(\mathcal{H}, S')$, then D_S is $\{\, (x, y) \mid x : \mathcal{H} \text{ or } y : \mathcal{H} \text{ or } x\, D_{S'}\, y \,\}$; (iii) if $S = \mathrm{PartitionBy}(K, S')$, then D_S is $\{\, (x, y) \mid (x\, D_{S'}\, y) \text{ and } x|_K = y|_K \,\}$; and (iv) if $S = \mathrm{Par}(S_1, S_2)$, then D_S is $D_{S_1} \cup D_{S_2}$.

The dependence relation D_S over the set X of tuples then gives rise to the following equivalence relation on sequences s, s' over X; this equivalence gives an alternative representation of the partial order on input events.

Definition 2.14 (Equivalent sequences)**.** Let $D \subseteq X \times X$ be a symmetric relation. The equivalence relation $\equiv_D$ over sequences over X is the smallest equivalence relation (i.e. reflexive, symmetric, and transitive) such that (1) commuting independent items: for all $x, y \in X$, if *not* $x \, D \, y$, then $xy \equiv_D yx$; and (2) closure under (sequence) concatenation: for $s_1, s_1', s_2, s_2' \in X^*$, if $s_1 \equiv_D s_1'$ and $s_2 \equiv_D s_2'$ then $s_1 s_2 \equiv_D s_1' s_2'$. For a schema S, two sequences s, s' are *equivalent with respect to* S, written $s \equiv_S s'$, if $s \equiv_{D_S} s'$.

The structure of D_S reflects the hierarchical series-parallel structure of synchronization schemas. Note that not all binary relations on tuples of **headers**(S) can be represented. In particular, the (symmetric reflexive closure of) the relations $\{(A, B), (B, C), (C, D)\}$ and $\{(A, B), (B, C), (C, D), (D, A)\}$ do not have a hierarchical structure: here there is no way to choose a header out of A, B, C, D to be a root node in the synchronization schema.

The following proposition characterizes exactly the dependence relations arising as D_S for some synchronization schema, based on these two examples.

Proposition 2.15. For any schema S, the relation D_S is symmetric and reflexive. It additionally satisfies the following restriction: D_S does not contain the cycle $C = \{(A, B), (B, C), (C, D)\}$ or the path $P = \{(A, B), (B, C), (C, D), (D, A)\}$ when restricted to any set of four headers a, b, c, d.

Proof. We refer to the cycle as C_4 and the path as P_4. Symmetry and reflexivity are by construction in each case of Definition 2.13. Now suppose that we introduce a cycle C or path P. It cannot have been introduced in the base cases $\text{Bag}(\mathcal{H})$ or $\text{Seq}(\mathcal{H})$, nor in parallel composition since C and P are connected; nor in $\text{PartitionBy}(K, S)$ since there are no dependencies across keys. So it must have been introduced by the $\text{Sync}(\mathcal{H}, S)$ construct. But for either C or P, there is no way to partition the vertices into those in $\mathcal{H}$ and those in S such that every tuple in the first is dependent on every tuple in the second. □

The reflexivity of D_S is not strictly necessary (we could drop it and have the empty relation in the base case of $\text{Bag}(\mathcal{H})$), but is convenient, as it means that the dependence relation contains no extraneous information other than what it implies about event ordering. In particular, we have the following proposition: for dependence relations D_1 and D_2, $D_1 = D_2$ if and only if $\forall s, s' \in X^*, s \equiv_{D_1} s' \iff s \equiv_{D_2} s'$. This means that if S_1 and S_2 are synchronization schemas, $D_{S_1} = D_{S_2}$ iff $\equiv_{S_1}$ and $\equiv_{S_2}$ are the same.

Proposition 2.16. Let D_1 and D_2 be two dependence relations on the same set of tuples X, arising from synchronization schema S_1 and S_2. Then $D_1 = D_2$ if and only if

$$\forall s, s' \in X^*, s \equiv_{D_1} s' \iff s \equiv_{D_2} s'.$$

In particular, if S_1 and S_2 are synchronization schemas, this implies $D_{S_1} = D_{S_2}$ if and only if $\equiv_{S_1}$ and $\equiv_{S_2}$ are the same equivalence relation on sequences.

Proof. The forward direction is immediate, and the backward direction follows taking any two distinct tuples x_1, x_2 and considering the sequence $x_1 x_2$. □

We next describe a tight correspondence between sequences and series-parallel streams via dependence relations. First we have to define what it means for a sequence to be a flattening of a series-parallel stream; we then show that sequences up to equivalence exactly correspond to series-parallel streams.

Example 2.17. Let us revisit the schema the taxi example. Suppose r_1, r_2 are `GPS` events of the red taxi, b_1, b_2 are `GPS` events of the blue taxi, r'_1, r'_2 are `RideCompleted` events of the red taxi, b' is a `RideCompleted` event of the blue taxi, and *eoh* is an end-of hour event. Following equivalent sequences

$$r_1, r'_1, b', b_1, r'_2, b_2, r_2, \mathit{eoh} \quad \equiv \quad b_1, b_2, r_1, r_2, b', r'_1, r'_2, \mathit{eoh}$$

are flattening of the SPS

$$[\langle\{\texttt{red} \mapsto [\bot, r_1, \bot, r_2, \bot], \\ \texttt{blue} \mapsto [\bot, b_1, \bot, b_2, \bot]\}, \{r'_1, r'_2, b'\}\rangle, \mathit{eoh}, \bot].$$

Definition 2.18 (Flattening)**.** Let S be a synchronization schema, and let t be a series-parallel stream over S. A *flattening* s of t is a sequence of tuples of type **headers**(S) given inductively as follows:

- If $S = \text{Bag}(\mathcal{H})$, then s is a flattening of t if and only if the multiset of events in s equals t.
- If $S = \text{Sync}(\mathcal{H}, S_1)$, and suppose $t = [t_0, d_1, t_1, \cdots, d_m, t_m]$ for some sub-streams t_i. Then s is a flattening of t if and only if $s = s_0 d_1 s_1 \ldots d_m s_m$ for some sequences $s_0, s_1, \ldots, s_m$ where s_i is a flattening of t_i for each i.
- If $S = \text{PartitionBy}(K, S')$, and suppose t is a set with finitely many entries $v_i \mapsto t_i$ for $i = 1, \ldots, m$. Then s is a flattening of t if and only if s is an interleaving of the sequences $s_1, s_2, \ldots, s_m$ where s_i is a flattening of t_i for each i.
- If $S = \text{Par}(S_1, S_2)$, and suppose t is a parallel composition of t_1 and t_2. Then s is a flattening of t if and only if s is an interleaving of the sequences s_1 and s_2 for some s_1, s_2 where s_1 is a flattening of t_1 and s_2 is a flattening of t_2.

The following proposition formalizes the connection between a series-parallel stream and its flattenings. In particular, given a sequence of tuples, once we fix a schema, there is a unique way to interpret it as a series-parallel stream.

Proposition 2.19. Let S be a synchronization schema. (1) For every sequence s of tuples of type **headers**(S), there exists a unique (up to equality) $t : S$ such that s is a flattening of t. (2) For all sequences s_1, s_2 of tuples of type **headers**(S) and $t : S$, (a) if $s_1 \equiv_S s_2$ and s_1 is a flattening of t then s_2 is a flattening of t also, and (b) if s_1 and s_2 are both flattenings of t then $s_1 \equiv_S s_2$.

2.2.3 Schema Subtyping

We end the section by looking at how dependence relations defined by synchronization schemas relate to each other: in particular, this allows defining what it means for one synchronization schema to be weaker or stronger than another in terms of the ordering requirements it imposes on sequences. Relaxation can be viewed as a sub-typing relation.

Definition 2.20 (Schema relaxation)**.** For synchronization schemas S_1 and S_2, S_1 is a *relaxation* of S_2, written $S_1 \lesssim S_2$, if $\textbf{headers}(S_1) \supseteq \textbf{headers}(S_2)$ and for all tuples $x, y : \textbf{headers}(S_2)$, if $x D_{S_1} y$ then $x D_{S_2} y$. Two synchronization schemas S_1 and S_2 are *order-equivalent*, denoted $S_1 \sim S_2$, if $D_{S_1} = D_{S_2}$. (Equivalently, if both $S_1 \lesssim S_2$ and $S_2 \lesssim S_1$.)

Example 2.21. Revisiting the schema of Figure 2, suppose we want to say that the ordering of `GPS` events of the same taxi is also not important. This can be captured by the schema

$$\text{Sync}(\texttt{EndOfHour}, \text{Par}(\text{PartitionBy}(\texttt{taxiID}, \text{Bag}(\texttt{GPS})), S_2))$$

which is a relaxation of the original schema. Such a schema will restrict the allowed computations, but increase the parallelization opportunities. This revised schema is equivalent to the schema $\text{Sync}(\texttt{EndOfHour}, \text{Bag}(\{\texttt{GPS}, \texttt{RideCompleted}\}))$

Assuming that we only have the GPS events of a single taxi together with the `EndOfHour` tuples, the following two schemas are order-equivalent.

$$\text{Sync}(\texttt{EndOfHour}, \text{Seq}(\texttt{GPS}))$$
$$\text{Seq}(\{\texttt{EndOfHour}, \texttt{GPS}\})$$

This means that they describe the same stream partial orders. Their difference is only relevant for defining hierarchical queries.

Proposition 2.22. If $t : S$ and $S' \lesssim S$, then there exists a unique $t' : S'$ such that every flattening of t is a flattening of t'.

Proposition 2.23. Schema relaxation, that is on two input schemas S_1 and S_2, checking if $S_1 \lesssim S_2$, and consequently checking schema equivalence, is decidable in quadratic time.

2.3 Programming with Data-Trace Types

In this section we give an illustration of type-consistent programming using data-trace types. We will also describe how type-consistent programming can be done for synchronization schemas in the next section. The semantics for a stream processing program consists of:

1. the type X of input data traces,
2. the type Y of output data traces, and
3. a monotone mapping $\beta : X \to Y$ that specifies the cumulative output after having consumed a prefix of the input stream.

The monotonicity requirement captures the idea that output items cannot be retracted after they have been omitted. Since β takes an entire input history (data trace) as input, it can model stateful systems, where the output that is emitted at every step depends potentially on the entire input history.

We have already discussed how (monotone) functions from A^* to B^* model sequential stream processors. We will now introduce the formal notion of *consistency*, which captures the intuition that a sequential implementation does not depend on the relative order of any two elements unless the stream type considers them to be relatively ordered.

Definition 2.24 (Consistency)**.** Let $X = (A, D)$ and $Y = (B, E)$ be data-trace types. We say that a data-string transduction $f : A^* \to B^*$ is (X, Y)-*consistent* if $u \equiv_D v$ implies that $\bar{f}(u) \equiv_E \bar{f}(v)$ for all $u, v \in A^*$.

Let $f \in A^* \to B^*$ be a (X, Y)-consistent data-string transduction. The function $\beta : X \to Y$, defined by $\beta([u]) = [\bar{f}(u)]$ for all $u \in A^*$, is called the (X, Y)-*denotation* of f.

Definition 2.25 (Data-Trace Transductions)**.** Let $X = (A, D)$ and $Y = (B, E)$ be data-trace types. A ***data-trace transduction*** with input type X and output type Y is a function $\beta : X \to Y$ that is monotone w.r.t. the prefix order on data traces: $\mathbf{u} \leq \mathbf{v}$ implies that $\beta(\mathbf{u}) \leq \beta(\mathbf{v})$ for all traces $\mathbf{u}, \mathbf{v} \in X$.

It is shown in [19] that the set of data-trace transductions from X to Y is equal to the set of (X, Y)-denotations of all (X, Y)-consistent data-string transductions.

We define two kinds of ***data-trace types*** for streams of key-value pairs: *unordered* types of the form $\mathsf{U}(K, V)$, and *ordered* types of the form $\mathsf{O}(K, V)$. For a set of keys K and a set of values V, let $\mathsf{U}(K, V)$ denote the type with alphabet $K \cup \{\texttt{#}\}$, values V for every key, values $\mathbb{N}$ for the `#` tag (i.e., marker timestamps), and dependence relation $\{(\texttt{#}, \texttt{#})\} \cup \{(k, \texttt{#}), (\texttt{#}, k) \mid k \in K\}$. In other words, $\mathsf{U}(K, V)$ consists of data traces where the marker tags `#` are linearly ordered and the elements between two such tags are of the form (k, v), where $k \in K$ and $v \in V$, and are completely unordered. We define $\mathsf{O}(K, V)$ similarly, with the difference that the dependence relation also contains $\{(k, k) \mid k \in K\}$. That is, in a data trace of $\mathsf{O}(K, V)$, elements with the same key are linearly ordered between `#` markers, but there is no order across elements of different keys.

A ***transduction DAG*** is a tuple $(S, N, T, E, \to, \lambda)$ which represents a labelled directed acyclic graph, where: S is the set of *source vertices*, T is the set of *sink vertices*, N is the set of *processing vertices*, E is the set of *edges* (i.e., connections/channels), $\to$ is the *edge relation*, and λ is a *labelling function.* The function λ assigns: (1) a data-trace type to each edge, (2) a data-trace transduction to each processing vertex that respects the input/output types, and (3) names to the source/sink vertices. We require additionally that each source vertex has exactly one outgoing edge, and each sink vertex has exactly one incoming edge.

Example 2.26 (Time-Series Interpolation)**.** Consider a home IoT system where temperature sensors are installed at a residence. We wish to analyze the sensor time series to create real-time notifications for excessive energy loss through the windows. The sensor time series sometimes have missing data points, and therefore the application requires a pre-processing step to fill in any missing measurements using linear interpolation. We assume that the sensors first send their measurements to a hub, and then the hub propagates them to the stream processing system. The stream that arrives from the hub does not guarantee that the measurements are sent in linear order (e.g., with respect to a timestamp field). Instead, it produces synchronization markers every 10 seconds with the guarantee that all elements with timestamps $< 10 \cdot i$ have been emitted by the time the i-th marker is emitted. That is, the i-th marker can be thought of as a watermark with timestamp $10 \cdot i$. The input stream is a data trace of $\mathsf{U}(\texttt{Ut}, \texttt{M})$, where `M` is the type of measurements $(\mathit{id}, \mathit{value}, \mathit{ts})$ consisting of a sensor identifier id, a scalar value value, and a timestamp ts. This is a transduction DAG that describes the pre-processing computation:

$$\begin{tikzcd}
\boxed{\texttt{HUB}} \arrow[r, "\mathsf{U}(\texttt{Ut},\texttt{M})"] & \boxed{\texttt{JFM}} \arrow[r, "\mathsf{U}(\texttt{ID},\texttt{V})"] & \boxed{\texttt{SORT}} \arrow[r, "\mathsf{O}(\texttt{ID},\texttt{V})"] & \boxed{\texttt{LI}} \arrow[r, "\mathsf{O}(\texttt{ID},\texttt{V})"] & \boxed{\texttt{SINK}}
\end{tikzcd}$$

The vertex `HUB` represents the source of sensor measurements, and the vertex `SINK` represents the destination of the output stream. `ID` is the type of sensor identifiers, and `V` is the type of timestamped values $(\mathit{value}, \mathit{ts})$. The processing vertices are described below:

- The stage Join-Filter-Map (`JFM`) joins the input stream with a table that indicates the location of each sensor, filters out all sensors except for those that are close to windows, and reorganizes the fields of the input tuple.
- Recall the guarantee for the synchronization markers, and notice that it implies the following property for the input traces: for any two input measurements that are separated by at least one marker, the one on the left has a strictly smaller timestamp than the one on the right. The sorting stage `SORT` sorts for each sensor the measurements that are contained between markers.
- The linear interpolation stage `LI` considers each sensor independently and fills in any missing data points.

We have described informally the data-trace transductions `JFM`, `SORT` and `LI`. The transduction DAG shown earlier denotes a data-trace transduction $\mathtt{U}(\mathtt{Ut}, \mathtt{M}) \to \mathtt{O}(\mathtt{ID}, \mathtt{V})$.

The computation performed by a processing node is given in a structured fashion, by completing function definitions of a specified ***operator template***. We describe the three templates that are supported below, which encompass both ordered and unordered input streams. Each operator is defined by a sequential implementation. This means that each operator can be modeled as a data-string transduction. It can then be proved formally that these data-string transductions are consistent w.r.t. their input/output data-trace types. It follows that each operator that is programmed according to the template conventions has a denotation (semantics) as a data-trace transduction of the appropriate type.

- `OpStateless`: The simplest template concerns *stateless* computations, where only the current input event—not the input history—determines the output. The programmer fills in two function definitions: (1) `onItem` for processing key-value pairs, and (2) onMarker for processing synchronization markers. The functions have no output (the output type is `Ut`, i.e. the unit type) and their only side-effect is emitting output key-value pairs to the output channel by invoking `emit(outputKey, outputValue)`.
- `OpKeyedOrdered`: Assuming that the input is ordered per key, this template describes a stateful computation for each key independently that is order-dependent. The programmer fills in three function definitions: (1) `initialState` for obtaining the initial state, (2) `onItem` for processing a key-value pair and updating the state, and (3) `onMarker` for processing a synchronization marker and updating the state. The functions have output S, which is the type of the data structure for representing the state. As for stateless computations, the functions allow the side-effect of emitting output key-value pairs to the output channel. This template requires a crucial *restriction* for maintaining the order for the output: every occurrence of `emit` must preserve the input key. If this restriction is violated, e.g. by projecting out the key, then the output cannot be viewed as being ordered.
- `OpKeyedUnordered`: Assuming that the input is unordered, this template describes a stateful computation for each key independently. Recall that the synchronization markers are ordered, but the key-value pairs between markers are *unordered*. To guarantee that the computation does not depend on some arbitrary linear ordering of the key-value pairs, their processing does not update the state. Instead, the key-value pairs between two consecutive markers are aggregated using the operation of

a *commutative monoid* A: the programmer specifies an identity element `id()`, and a binary operation `combine()` that must be *associative* and *commutative*. Whenever the next synchronization marker is seen, `updateState` is used to incorporate the aggregate (of type A) into the state (of type S) and then `onMarker` is invoked to (potentially) emit output. The behavior `onItem` may depend on the last snapshot of the state, i.e. the one that was formed at the last marker. The functions `onItem` and `onMarker` are allowed to emit output data items (but not markers), but the rest of the functions must be pure (i.e., no side-effects).

Theorem 2.27. Every streaming computation defined using the operator templates above is consistent w.r.t. its input/output type.

2.4 Programming with Synchronization Schemas

In this section, we define *SPS-transformers* (SPSTs), a programming language for computations over series-parallel streams. If synchronization schemas are provided as types for the input and output streams of a computation, then an SPST is a (deterministic) function from the input to the output, which respects the structure given by the types. Before defining the language constructs, we begin with a discussion of our *design goals*: what properties should computations written in this language satisfy? We identify the following design goals. First, transformations over series-parallel streams should *respect* the **parallelism** in the input and output: parallel input events should be processed in parallel, and parallel input threads should produce parallel output events. For example, given an input which is two streams in parallel, the computation should be written in such a way that the two streams are processed separately, and outputs corresponding to them should be unordered. Second, to allow for the specification of potentially complex computations, we additionally want our language to be **modular**: it should be natural to construct a computation by combining sub-computations. For example, processing a stream of the hierarchical type $\text{Sync}(\mathcal{H}, S)$ should be definable, both syntactically and semantically, in terms of existing computations defined over sub-streams of type S. Finally, any computation in our language should be **streamable**: it should process the input in one pass, producing output incrementally.

To satisfy these design goals, we make the following technical choices. First, to satisfy the parallelism goal, we define SPSTs to have SPS as input and output rather than sequential objects. The input being an SPS allows us to specify the computation to exploit parallelism, and the output being an SPS requires that we respect parallelism when producing output events.

To understand the challenges in defining the semantics due to the interplay between streamability and modularity, consider a transformer P processing hierarchical streams of type $S = \text{Sync}(\mathcal{H}, S')$ that we would like define in terms of a transformer P' processing streams of type S'. Consider an input stream $t = [\bot, d, t']$ of type S for an $\mathcal{H}$-tuple d and stream t' of type S'. Suppose we want to extend the input stream t with a tuple d'. If the type of d' is one of the headers appearing in S', then it really extends the sub-stream t', and should be processed by the transformer P'. For streamability we want to make sure that, while processing d', P' extends the output stream only by adding new items. Formally, this means that the output of P' on the input stream t' should be a prefix of its output on the stream $t' \circ d$. With this motivation, we define such a semantics, which we call *open* semantics, for transformers as functions from input to output streams, and ensure that it is *monotonic* with respect to prefix ordering (see Theorem 2.34). But now suppose that the item d' is an

$\mathcal{H}$-tuple that acts as a synchronization marker for the events in the sub-stream t'. Then to process it, the transformer P' should return, and let the top-level transformer P process the item d'. During this return, the transformer P' can do additional computation and produce additional output items even though the stream it has processed is still t'. This is a typical case when S' corresponds to key-based partitioning, and the arrival of the synchronization marker d' triggers the reduce operation that aggregates the results of the computations of the key-indexed sub-streams of t'. This though requires us to define another semantics of the transformer P' on the input stream t' that extends the open semantics and includes the results of the computation upon return. We call it *closed* semantics to indicate that it is applicable when the current stream is being closed. Note that the result of computation of P on the stream $[\bot, d, t', d', \bot]$ can be described by relying on the closed semantics of P' on the stream t'. In terms of existing work on punctuation, the closed semantics can be thought of as the stream output on a stream terminated by an *end-of-stream* marker.

Finally, an SPST is a function on pairs: it takes an initial value and an input SPS to a final value and an output SPS. We need this for modularity: without the initial value as input, an SPS-transforer on a sub-stream of the input could not be initialized based on the surrounding context. Similarly, the final value (separate from the series of output items produced) can be used to describe a summary of the input stream to be used in the surrounding context when the computation finishes.

We summarize all of these choices in the following definition of the *interface* for an SPST. We also define subtyping for the interface, where the output is relaxed. Each of the language constructs will then implement this interface. In the Appendix, we give an extended example to illustrate the formal definitions in this section.

Definition 2.28 (SPS-transformer interface)**.** An SPS-transformer (SPST) P has:

- A *type* denoted (X, S, X', S'), where X is the type for the initialization value, S is an input synchronization schema, X' is a type for the final return value, and S' is an output synchronization schema. We write $P : (X, S, X', S')$.
- An *open semantics* denoted $[\![P]\!]_O(x, t) = t'$, where $x : X$ is the initial value, $t : S$ is the input SPS, and $t' : S'$ is the incrementally produced output SPS.
- A *closed semantics* denoted $[\![P]\!]_C(x, t) = (x', t')$, where $x' : X'$ is the initial value, $t : S$ is the input SPS, $x' : X'$ is the final value, and $t' : S'$ is the output SPS. We additionally enforce that the open semantics is a prefix of the closed semantics: $[\![P]\!]_O(x, t) \preceq t'$. □

Definition 2.29. If $S'' \lesssim S'$ (Definition 2.20), then (X, S, X', S'') is a *subtype* of (X, S, X', S'). If $P : (X, S, X', S')$ then we also write $P : (X, S, X', S'')$. The open and closed semantics are derived as the unique output stream given by Proposition 2.22. □

In the remainder of the section, we give one language construct corresponding to each constructor of the input series parallel stream. Some additional notation: for a set of headers $\mathcal{H}$, we write $\mathbf{tup}(\mathcal{H})$ for the set of tuples $x : \mathcal{H}$. For a synchronization schema S, we write $\mathbf{sps}(S)$ for the set of series-parallel streams $t : S$. We write $\mathbf{bag}(X)$ for the set of bags (multisets) of items of type X.

2.4.1 Relational SPST

We start with the relational SPST, which represents a standard relational operator that can be used to process a bag of items, producing another bag of items. Relational operators are well studied and are commonly defined using SQL and its extensions. Our design choice here is to not impose a particular relational base language or SQL variant; instead, the relational operator is given as two *black-box* functions, which define the open and closed semantics, respectively. We only require that these are functions on bags (i.e. independent of the input order), and that the open semantics is monotone and a prefix of the closed semantics.

Definition 2.30 (Relational SPST)**.** A relational SPST

$$P : (X, \mathrm{Bag}(\mathcal{H}), X', \mathrm{Bag}(\mathcal{H}'))$$

consists of two fields:

$$\begin{aligned} P.\mathsf{open} &: X \times \mathbf{sps}(\mathrm{Bag}(\mathcal{H})) \to \mathbf{sps}(\mathrm{Bag}(\mathcal{H}')) \\ \text{and} \quad P.\mathsf{closed} &: X \times \mathbf{sps}(\mathrm{Bag}(\mathcal{H})) \to X' \times \mathbf{sps}(\mathrm{Bag}(\mathcal{H}')). \end{aligned}$$

such that (1) $P.\mathsf{open}$ is *monotone*: if $r_1 \preceq r_2$, then $P.\mathsf{open}(x, r_1) \preceq P.\mathsf{open}(x, r_2)$; and (2) $P.\mathsf{open}$ is a prefix of $P.\mathsf{closed}$: if $P.\mathsf{closed}(x, r) = (x', r')$ then $P.\mathsf{open}(x, r) \preceq r'$. The semantics of P is defined as $[\![P]\!]_O(x, r) = P.\mathsf{open}(x, r)$ and $[\![P]\!]_C(x, r) = P.\mathsf{closed}(x, r)$. □

2.4.2 Parallel SPST

We now define the inductive SPSTs. An SPST processing inputs of type $\mathrm{Par}(S_1, S_2)$ is composed two SPSTs running in parallel independently. The question here is, can the components SPSTs produce tuples of the same type? The answer is yes, provided such tuples, since they get produced independently, are summarized using a schema $\mathrm{Bag}(\mathcal{O})$, where $\mathcal{O}$ is a set of output headers. So the output schema for the parallel SPST will be $\mathrm{Par}(S_1', S_2', \mathrm{Bag}(\mathcal{O}))$.

Definition 2.31 (Parallel SPST)**.** Let S_1, S_2, S_1', S_2' be schemas. A parallel SPST

$$P : (X, \mathrm{Par}(S_1, S_2), X', \mathrm{Par}(S_1', S_2', \mathrm{Bag}(\mathcal{O}')))$$

consists of internal types X_1, X_2, X_1', X_2' and four fields:

$$\begin{aligned} P.\mathsf{left} &: (X_1, S_1, X_1', \mathrm{Par}(S_1', \mathrm{Bag}(\mathcal{O}'))), \\ P.\mathsf{right} &: (X_2, S_2, X_2', \mathrm{Par}(S_2', \mathrm{Bag}(\mathcal{O}'))), \\ P.\mathsf{init} &: X \to X_1 \times X_2, \quad \text{and} \quad P.\mathsf{fin} : X_1' \times X_2' \to X'. \end{aligned}$$

The semantics of P is as follows: if we have that $P.\mathsf{init}(x) = (x_1, x_2)$, $[\![P.\mathsf{left}]\!]_O(x_1, t_1) = \langle t_1', r_1' \rangle$, and $[\![P.\mathsf{right}]\!]_O(x_2, t_2) = \langle t_2', r_2' \rangle$, where $r_1', r_2' : \mathrm{Bag}(\mathcal{O}')$, and additionally $[\![P.\mathsf{left}]\!]_C(x_1, t_1) = (x_1', \langle t_1'', r_1'' \rangle)$ and $[\![P.\mathsf{right}]\!]_C(x_2, t_2) = (x_2', \langle t_2'', r_2'' \rangle)$, then

$$\begin{aligned} [\![P]\!]_O(x_1, t_1) &= \langle \langle t_1', t_2' \rangle, r_1' \cup r_2' \rangle \\ [\![P]\!]_C(x_1, t_1) &= (P.\mathsf{fin}(x_1', x_2'), \langle \langle t_1'', t_2'' \rangle, r_1'' \cup r_2'' \rangle). \quad \square \end{aligned}$$

2.4.3 Hierarchical SPST

When the input schema is $S = \mathrm{Sync}(\mathcal{H}, S_1)$, we want to define the corresponding SPST P parameterized by a sub-SPST from S_1 to S_1'. The SPST P maintains its own state that gets updated sequentially whenever any of the $\mathcal{H}$-tuple is processed, is passed to the sub-SPST when called, and updated when the sub-SPST returns. The output schema of P has the same structure as the input: it is divided into synchronizing events and non-synchronizing events. On input synchronization events, any output tuple may be produced, including a synchronization event; but on input sub-stream events, it would be incorrect to produce an output synchronizing event, as this would not be produced in a consistent order. The distinction between closed and open semantics plays a key role here: synchronizing events, when processed by P, "close" the computation of the sub-SPST. To formalize this inductively, we introduce an auxiliary semantics $[\![P]\!]_{Aux}(y, t)$ where the output is an internal states (rather than a final values), and in which the input stream ends with a d_i event, i.e. the final t_i is $\perp$.

Definition 2.32 (Hierarchical SPST)**.** Let S_1 and S_1' be schemas, and $\mathcal{H}$ and $\mathcal{H}'$ be a set of input and output headers, respectively. Let $S' = \mathrm{Sync}(\mathcal{H}', S_1')$. A hierarchical SPST

$$P : (X, \mathrm{Sync}(\mathcal{H}, S_1), X', \mathrm{Sync}(\mathcal{H}', S_1'))$$

consists of internal types X_1, X_1', Y and six fields:

$$\begin{aligned} P.\mathsf{sub} &: (X_1, S_1, X_1', S_1'), \\ P.\mathsf{update} &: Y \times \mathbf{tup}(\mathcal{H}) \to Y \times \mathbf{sps}(S'), \\ P.\mathsf{call} &: Y \to X_1, \qquad P.\mathsf{return} : Y \times X_1' \to Y, \\ P.\mathsf{init} &: X \to Y, \qquad \text{and} \quad P.\mathsf{fin} : Y \to X' \times \mathbf{sps}(S'). \end{aligned}$$

The auxiliary semantics of P is denoted $[\![P]\!]_{Aux}(y, t) = (y', t')$, where $y, y' : Y$, and defined inductively *only* for t of the form $[t_0, d_1, t_1, \ldots, d_m, \perp]$. For the base case, $[\![P]\!]_{Aux}(y, \perp) = (y, \perp)$. Then inductively, if $[\![P]\!]_{Aux}(y, t) = (y', t')$, $t_1 : S_1$, and $d : \mathcal{H}$, and if we have $P.\mathsf{call}(y') = x_1$, $[\![P.\mathsf{sub}]\!]_C(x_1, t_1) = (x_1', t_1')$, $P.\mathsf{return}(y', x_1') = y''$, and $P.\mathsf{update}(y'', d) = (y''', t'')$, then $[\![P]\!]_{Aux}(y, t \circ [t_1, d, \perp]) = (y''', t' \circ t_1' \circ t'')$. Given the auxiliary semantics, we define the semantics of P on a trace decomposed as $t \circ [t_1]$, where t ends in an empty sub-trace. Let $P.\mathsf{init}(x) = y$, $[\![P]\!]_{Aux}(y, t) = (y', t')$, and $P.\mathsf{call}(y') = x_1$. Additionally, let $[\![P.\mathsf{sub}]\!]_C(x_1, t_1) = (x_1', t_1')$, $P.\mathsf{return}(y', x_1') = y''$, and $P.\mathsf{fin}(y'') = (x', t'')$. Then:

$$\begin{aligned} [\![P]\!]_O(x, t) &= t' \circ [\![P.\mathsf{sub}]\!]_O(x_1, t_1) \\ [\![P]\!]_C(x, t) &= (x', t' \circ t_1' \circ t''). \quad \square \end{aligned}$$

2.4.4 Partitioned SPST

Finally, we define SPST for the partition-by case. The idea here is analogous to the parallel composition $\mathrm{Par}(S_1, S_2)$ case: each sub-stream corresponding to a different key value may produce output corresponding to that key value, *or* produce output corresponding to a common bag of tuples $\mathcal{O}'$. The partitioned SPST initializes the state of $P.\mathsf{sub}$ for each key with a non empty series parallel stream and runs the child SPST for each (non-empty) key in parallel. We additionally need an aggregation stage (applicable to the closed semantics only), in which we combine all of the partitioned states using a black-box relational operator $P.\mathsf{agg}$, similar to what was done in the relational SPST base case.

Definition 2.33 (Partitioned SPST)**.** Let $S = \mathrm{PartitionBy}(K, S_1)$ and $S' = \mathrm{PartitionBy}(K, S_1')$ be schemas, and $\mathcal{O}'$ a set of headers. A partitioned SPST

$$P : (X, \mathrm{PartitionBy}(K, S_1), X', \mathrm{Par}(\mathrm{PartitionBy}(K, S_1'), \mathrm{Bag}(\mathcal{O}'))$$

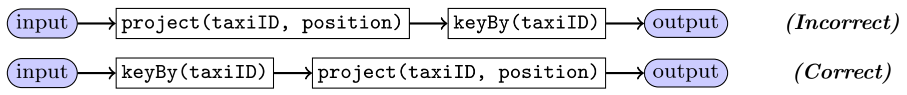


Figure 3: A subtle consequence of implicit parallelism over an input stream containing taxi location data.

consists of internal types X_1, X_1' and three fields:

$$\begin{aligned} &P.\mathsf{sub} : (X_1, S_1, X_1', \mathrm{Par}(S_1', \mathrm{Bag}(\mathcal{O}'))), \\ &P.\mathsf{init} : X \times \mathbf{tup}(K) \to X_1, \\ \text{and}\quad &P.\mathsf{agg} : X \times \mathbf{bag}((\mathbf{tup}(K) \times X_1') \to X' \times \mathbf{bag}(\mathbf{tup}(\mathcal{O}')). \end{aligned}$$

For the semantics, suppose $t = \{v_1 \mapsto t_1, \ldots, v_m \mapsto t_m\}$, and for $i = 1, \ldots, m$, $P.\mathsf{init}(x, v_i) = x_i$, $[\![P.\mathsf{sub}]\!]_C(x_i, t_i) = (x_i', \langle t_i', r_i' \rangle)$, $P.\mathsf{agg}(x, \{(v_1, x_1), \ldots, (v_m, x_m)\}) = (x', r_0')$, and $[\![P.\mathsf{sub}]\!]_O(x_i, t_i) = \langle t_i'', r_i'' \rangle$. Then

$$\begin{aligned} [\![P]\!]_C(x, t) &= (x', \langle \{v_1 \mapsto t_1', \ldots, v_m \mapsto t_m'\}, r_0' \cup r_1' \cup \cdots \cup r_m' \rangle \\ [\![P]\!]_O(x, t) &= \langle \{v_1 \mapsto t_1'', \ldots, v_m \mapsto t_m''\}, r_1'' \cup \cdots \cup r_m'' \rangle. \qquad \square \end{aligned}$$

2.4.5 Streamability

This brings us to our main theorem about SPSTs, defined using all of the above constructs, which captures the streamability (monotonicity) of the open semantics (proven in the Appendix).

Theorem 2.34. Let $P : (X, S, X', S')$ be an SPST. Then P is monotone in the following sense: for any $x : X$ and $t, u : S$, if $t \preceq u$, then $[\![P]\!]_O(x, t) \preceq [\![P]\!]_O(x, u)$. $\square$

3 Testing

3.1 Motivating Examples

Programs written in distributed stream processing frameworks exhibit implicit parallelism, which can lead to subtle bugs. Programs in such frameworks are usually written as *dataflow graphs*, where the edges are data streams and the nodes are streaming operators or transformations. Common operators include stateless transformations (*map*), and operations that group events based on the value of some field (*key-by*). For example, suppose that we have a single input stream which contains information about rides of a taxi service: each input event (`id`, `pos`, `meta`) consists of a taxi identifier, the taxi position, and some metadata. In the first stage, we want to discard the metadata component (*map*) and partition the data by taxi ID (*key-by*). In the second stage, we want to aggregate this data to report the total distance traveled by each taxi. Notice that the second stage is order-dependent (events for each taxi need to arrive in order), so it is important that the first stage does not disrupt the ordering of events for a particular taxi ID.

To make a program for the first stage of this computation in a distributed stream processing framework such as Flink or Storm, we need to build a dataflow graph representing a sequence of transformations on data streams. A first (natural) attempt to write the program

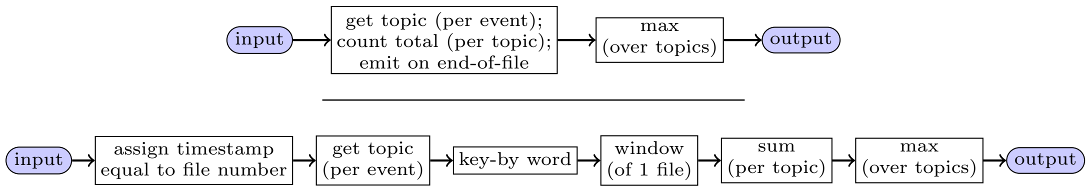


Figure 4: A difficult-to-parallelize sequential program (top), and the correct parallel version (bottom) over an input set of documents which arrive concatenated in a single stream.

is given in fig. 3 (top). Here, the `project` node projects the data to only the fields we are interested in; in this case, `taxiID` and `position`. And `keyBy` (also known as "group by" in SQL-like languages, or the concept of a "stream grouping" in Storm) partitions the data stream into substreams by `taxiID`. Although written as an operator, here `keyBy` can be thought of as modifying the stream to give it a certain property (namely, if it is parallelized, streams should be grouped by the given key).

The first attempt is incorrect, however, because it fails to preserve the order of data for a particular key (taxi ID), which is required for the second stage of the computation. The problem is that dataflow graph operators are implicitly parallelized—here, the stateless map `project` is internally replicated into several copies, and the events of the input stream are divided among the copies. Because input events of the same key may get split across substreams, when the operator `keyBy` reassigns each item to a new partition based on its key, if items of a particular key were previously split up, then they might get reassembled in the wrong order.

This issue can be addressed by ensuring that parallelization is done only on the basis of `taxiID` *from the beginning of the pipeline*. This can typically be accomplished by simply by reversing the `project` and `keyBy` transformations, as in fig. 3 (bottom). (For example, this is done explicitly in Flink, and the concept is the same in Storm, except that instead of an explicit `keyBy` operator we implicitly construct it by setting the input stream to be grouped by key.) Although the two programs are equivalent when the `project` operation is not parallelized, the second lacks the undesirable behavior in the presence of parallelism: assuming the `project` operation has the same level of parallelism as `keyBy`, most systems will continue to use the same partition of the stream to compute the projection, so data for each key will be kept in-order. In particular, this works in any framework which guarantees that the same key-based partitioning is used between stages.

We have seen that even simple programs can exhibit counterintuitive behavior. In practice, programs written to exploit parallelism are often much more complex. To illustrate this, consider a single input stream consisting of very large documents, where we want to assign a topic to each document. The documents are streamed word by word and delineated by end-of-file markers. The topic of each word is specified in a precomputed database, and the topic of a document is defined to be the most frequent topic among words in that document.

In this second example querying the database is a costly operation, so it is desirable to parallelize by partitioning the words within each document into substreams. However, the challenge is to do so in a way that allows for the end-of-file markers to act as *barriers*, so that we re-group and output the summary at the end of each document. Although a sequential solution for this problem is easy, the simplest solution we have found in Flink that exploits parallelism uses about twice as many lines of code (fig. 4). The source of the complexity is that we must first use the end-of-file events to assign a unique timestamp to each document (ignoring the usual timestamps on events used by Flink). After these timestamps are assigned, only then is it safe to parallelize, because windowing by timestamp later recovers the original file (set of events with a given timestamp). We also consulted with Flink users on the Flink mailing list, and we were not able to come up with a simpler solution. The additional complexity in developing the parallel solution, which requires changing the dataflow structure and not simply tuning some parameter, further motivates the need for differential testing.

3.2 DiffStream

These examples motivate the need for some form of testing to determine the correctness of distributed stream processing applications. We propose *differential testing* of the sequential and parallel versions. As the parallel solution might be much more involved, this helps validate that parallelization was done correctly and did not introduce bugs.

In the example of fig. 3, the programmer begins with either the correct program P_1 (bottom), or the incorrect program P_1' (top), and wishes to test it for correctness. To do so, they write a correct reference implementation P_2; this can be done by explicitly disallowing parallelism. Most frameworks allow the level of parallelism to be customized; e.g. in Flink, it can be disabled by calling `.setParallelism(1)` on the stream. The program P_1 or P_2 is then viewed as a black-box reactive system: a function from its input streams to a single *output stream* of events that are produced by the program in response to input events.

However, the specification of P_1 and P_2 alone is not enough, because we need to know whether the output data produced by either program should be considered unordered, ordered, or a mixture of both. A naive differential testing algorithm might assume that output streams are out-of-order, checking for multiset equivalence after both programs finish; but in this case, the two possible programs P_1 will both be equivalent to P_2. Alternatively, it might assume that output streams are in-order; but in this case, neither P_1 nor P_1' will be equivalent to P_2, because data for different taxi IDs will be out of order in the parallel solution. To solve this, the programmer additionally specifies a *dependence relation*: given two events of the output stream, it returns *true* if the order between them should be considered significant. For this example, output events are dependent if they have the same taxi ID. In general, the dependence relation can be used to describe a flexible combination of ordered, unordered, or partially ordered data.

The end-to-end testing architecture is shown in fig. 5. In summary, the programmer provides: (1) a program (i.e., streaming dataflow graph) P_1 which they wish to test for correctness; (2) a correct reference implementation P_2; (3) a *dependence relation* which tells the tester which events in the output stream may be out-of-order; (4) if needed, overriding the definition of equality for output stream events (for example, this can be useful if the output items may contain timestamps or metadata that is not relevant for the correctness of the computation); and (5) optionally, a custom generator of input data streams, or a custom input stream—otherwise, the default generator is used to generate random input streams. The two programs are then connected to our differential testing algorithm, which consumes the output data, monitors whether the output streams so far are equivalent, and reports a mismatch in the outputs as soon as possible.

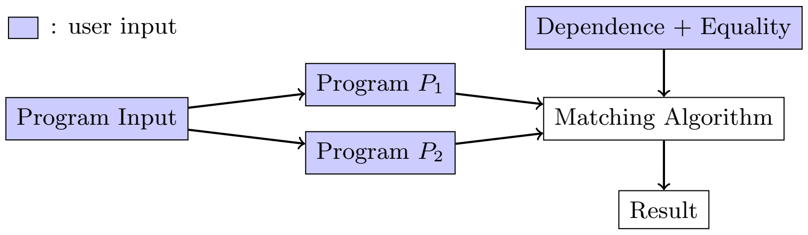
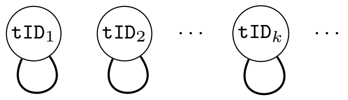


Figure 5: Architecture for our testing framework.

```
  (ev1, ev2) ->
      ev1.taxiID == ev2.taxiID
```

(a) Specification in DiffStream



(b) Dependence visualized as a graph

Figure 6: Example specification in DiffStream for the taxi example. Taxi events with the same `taxiID` are dependent.

3.3 Writing Specifications in DiffStream

In this section we describe how the programmer writes specifications in DiffStream. Let's look back at the taxi example from before. The second stage of the program computes the total distance traveled by each taxi by computing the distance between the current and the previous location, and adding that to a sum. For this computation to return correct results, location events for each taxi should arrive in order in its input—a requirement that must be checked if we want to test the first stage of the program.

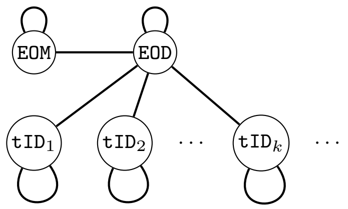
A dependence relation is a symmetric binary relation on events of a stream with the following semantics. If x D y, then the order of x and y in a stream is significant and reordering them gives us two streams that are not equivalent. This could be the case if the consumer of an output stream produces different results depending on the order of x and y. Thus, the dependence relation can be thought of as encoding the pairwise ordering requirements of the downstream consumer.

It is often helpful to visualize dependence relations as unordered graphs, where nodes are equivalence classes of the dependence relation. For the taxi example, the dependence relation is visualized in Figure 6b, and it indicates that events with the same taxi identifier are dependent. In DiffStream, dependence relations can be specified using a Boolean function on a pair of events. These functions should be pure and should only depend on the fields of the two events. The DiffStream specification of the dependence relation from Figure 6b is shown in Figure 6a.

Now let's consider an extension of the above example where the downstream consumer computes the total distance traveled by each taxi *per day*, and also computes the average daily distance by each taxi every month. To make this possible, the output of the program under test is now extended with special EOD (*end-of-day*) and EOM (*end-of-month*) events. The ordering requirements on this output, while more subtle, can still be precisely specified using a dependence relation. For example, EOD events are dependent with taxi events since all events of a specific day have to occur before the EOD event of that day for the total daily distance to be correctly computed. On the other hand, EOM events do not have to be

```
(ev1, ev2) ->
    ev1.isEOD() ||
    ev2.isEOD() ||
    (ev1.isEOM() && ev2.isEOM()) ||
    (ev1.isTaxiEv() &&
     ev2.isTaxiEv() &&
     ev1.taxiID == ev2.taxiID)
```

(a) Specification in DiffStream



(b) Dependence visualized as a graph

Figure 7: Example specification in DiffStream for the extended taxi example. Taxi events with the same `taxiID` are dependent and all events are dependent with end-of-day (EOD) events.

```
(ev1, ev2) -> distance(ev1.loc, ev2.loc) < 1
```

Figure 8: Example specification in DiffStream where events are dependent if their locations are close.

dependent with taxi events since daily distances are computed on EOD events. Therefore, an EOM event can occur anywhere between the last EOD event of the month and the first EOD event of the next month. The DiffStream specification of the dependence relation and its visualization are both shown in Figure 7.

Several frequently occurring dependence relations can be specified using a combination of the predicates seen in the above examples. This includes predicates that check if an event is of a specific type (e.g. `isEOD()`, `isTaxiEv()`), and predicates that check a field (possibly denoting a key or identifier) of the two events for equality (e.g. `ev1.taxiID == ev2.taxiID`). However, it is conceivable that the dependence of two events is determined based on a complex predicate on their fields.

Another interesting dependence relation occurs in cases where output streams contain punctuation events. Punctuations are periodic events that contain a timestamp and indicate that all events up to that timestamp, i.e. all events `ev` such that `ev.timestamp < punc.timestamp`, have *most likely* already occurred. Punctuation events allow programs to make progress, completing any computation that was waiting for events with earlier timestamps. However, since events could be arbitrarily delayed, some of them could arrive after the punctuation. Consider as an example a taxi that briefly disconnects from the network and sends the events produced while disconnected after it reconnects with the network. These events are usually processed with a custom out-of-order handler, or are completely dropped. Therefore, punctuation events are dependent with events that have an earlier timestamp, since reordering them alters the result of the computation, while they are independent of events with later timestamps. This can be specified in DiffStream as shown in Figure 9.

3.4 Differential Testing Algorithm

Algorithm DiffStream shows our algorithm for checking equivalence of two streams. As described in the overview, the algorithm has two main features: (i) it can check for equivalence

```
(ev1, ev2) -> (ev1.isPunctuation() &&
                ev2.timestamp < ev1.timestamp) ||
               (ev2.isPunctuation() &&
                ev1.timestamp < ev2.timestamp)
```

Figure 9: Example specification in DiffStream where punctuation events, used to enforce progress, depend on other events only if the punctuation timestamp is larger.

Algorithm DiffStream Checking equivalence of two streams

```
Input: Equality relation $\equiv$, dependence relation D
Input: Connected stream $s$ with $\pi_1(s) = s_1$ and $\pi_2(s) = s_2$
Require: Relations $\equiv$ and D are compatible
1: function StreamsEquivalent($s$)
2:     $u_1, u_2 \leftarrow$ empty logically ordered sets
3:     Ghost state: $p_1, p_2 \leftarrow$ empty logically ordered sets
4:     Ghost state: $f \leftarrow$ empty function $p_1 \to p_2$
5:     for $(x, i)$ in $s$ do
6:         $j \leftarrow 3 - i$
7:         if $x$ is minimal in $u_i$ and $\exists y \in \min u_j : x \equiv y$ then
8:             $u_j \leftarrow u_j \setminus \{y\}$
9:             $p_i \leftarrow p_i \cup \{x\}$; $p_j \leftarrow p_j \cup \{y\}$
10:            $f \leftarrow f[x \mapsto y]$ if $i = 1$ else $f[y \mapsto x]$
11:        else if $\exists y \in u_j : x \mathrel{D} y$ then
12:            return false
13:        else
14:            $u_i \leftarrow u_i \cup \{x\}$
15:    return $(u_1 = \emptyset$ and $u_2 = \emptyset)$
```

up to any reordering dictated by a given dependence relation, and (ii) it is online—it processes elements of the stream one at a time. In [42] we prove that our algorithm is correct, and we show that it is optimal in the amount of state it stores during execution.

3.5 Evaluation

Here we show some excerpts from the evaluation of DiffStream. Figure 10 shows an example test written using DiffStream for the taxi example. Figure 11 shows the results of a case study on MapReduce programs to find bugs due to nondeterminism. Finally, Figure 12 shows the results of the final case study on measuring the performance overhead of the algorithm.

4 Distribution

4.1 Motivating Application

Fraud detection Suppose there are two types of input events: bank transactions and fraud detection rules that are used to flag some transactions as fraudulent. Bank transactions

```
public void testKeyBy() throws Exception {
    StreamExecutionEnvironment env = ...;

    DataStream input = generateInput(env);

    StreamEquivalenceMatcher matcher =
        StreamEquivalenceMatcher.createMatcher(
            sequentialImpl(input), parallelImpl(input),
            (ev1, ev2) -> ev1.taxiID == ev2.taxiID);

    env.execute();
    matcher.assertStreamsAreEquivalent();
}
```

Figure 10: An example test in DiffStream.

		Application-Specific Requirements	
Code pattern	Determinism	Determinism under input assumptions	None (nondeterminism acceptable)
`SingleItem`	✓	✓	n/a
`IndexValuePair`	✓	✓	n/a
`MaxRow`	✓	✓	✗
`FirstN`	✓	✓	✗
`StrConcat`	✓	n/a	✓

Figure 11: Results of the MapReduce case study. A ✓ indicates successfully identifying the bug in the first column, and successfully avoiding a false positive in the second and third columns, for each of the 5 reducers implemented.

are continuously input in the system with high rate while new fraud detection rules arrive infrequently. An example of a new rule would be the addition of a specific account to an index of suspicious accounts. There is no way to partition inputs while avoiding cross-partition dependencies since all events (both transactions and rules) depend on previous rule events. Still, if the rate of bank transaction events becomes a bottleneck (and fraud detection rule events happen rarely) it would be beneficial to parallelize the processing of bank transaction events.

One of the solutions that has been used to achieve data parallelism in such computations involves extending the dataflow model with a *broadcast* pattern that allows sending some input events to *all* parallel shards (and partitioning the rest of the events as usual). An example execution of the above computation that utilized the broadcast pattern is shown in Figure 13a. By broadcasting the fraud detection rule events, a parallel implementation is achieved without requiring cross-instance communication between the parallel instances since dependencies are contained in a single shard. However, as we see below, this solution does not generalize to more complex dependencies.

Fraud detection with a machine learning (ML) extension Consider a simple extension of the above application that is inspired by today's ML workflows. In addition to user-input fraud detection rules, the extended application also trains a model in an unsupervised way using previously seen transactions. To achieve that the application keeps a sketch of previously seen bank transaction events. When a new fraud detection rule arrives,

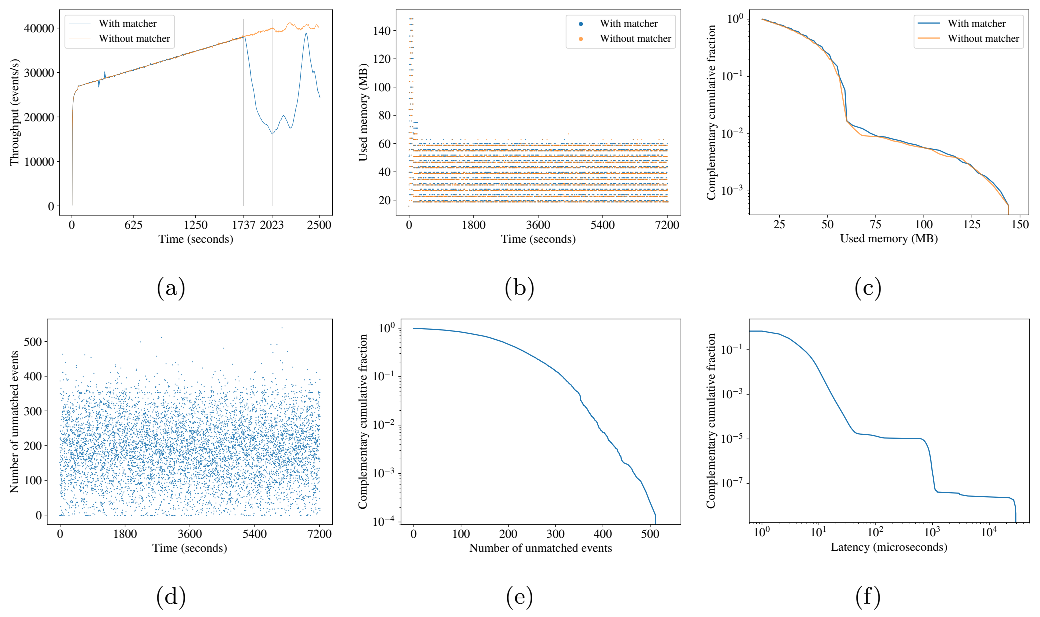


Figure 12: Results of the fourth case study: performance measurements of monitoring an application with DiffStream on the Yahoo streaming benchmark over a span of 2 hours, compared to the same application without the DiffStream matcher.

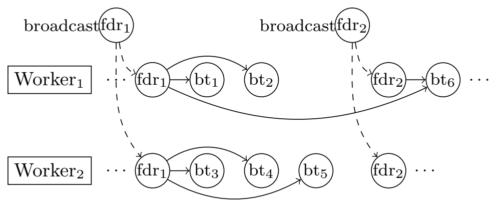
it aggregates the sketches and uses them to retrain the global fraud detection model. As shown in Figure 13b, this introduces dependencies across shards as the model retraining on rule events depends on all the previous bank transactions (even after broadcasting the rules), requiring communication between shards.

Unfortunately, standard ways of achieving data parallelism do not support computations with such cyclic dependencies, leaving the user with two unsatisfying solutions. They can either accept this shortcoming and execute their computations with restricted parallelism—introducing a throughput bottleneck in their application, or they can manually implement the required synchronization using low level external synchronization mechanisms (e.g. a key-value store with strong consistency guarantees). This is error prone and, more importantly, violates the implicit assumptions about the usage of streaming APIs possibly leading to bugs and erroneous behaviors.

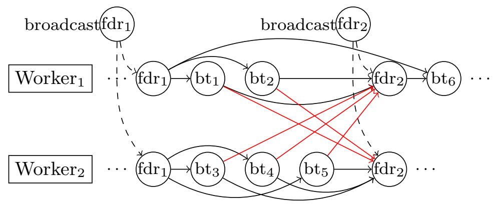
4.2 System Architecture

Our solution can achieve data parallelism for the extended fraud detection application through the architecture summarized in Figure 14. The two primary abstractions (shown in blue) encode the required complex synchronization requirements at different levels of abstractions: the DGS specification describes the computation and input dependencies in a platform-independent manner, and synchronization plans express the synchronization between processes at the implementation level, as communications between a hierarchically structured tree of processes.

The DGS specification is split in three parts. First, the user needs to provide a sequential



(a) Fraud detection.



(b) ML extension of fraud detection.

Figure 13: Illustration of an execution of two fraud detection examples and the event dependencies. Time progresses from left to right, different rows represent different parallel workers, circles represent processing of events, dashed edges represent broadcasting an event, black edges represent dependencies, and red edges represent dependencies across parallel instances.

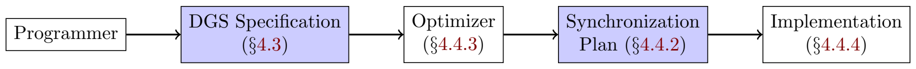


Figure 14: Architecture of the end-to-end DGS framework.

implementation of the program, where the input is assumed to arrive in order and one event at a time. The sequential implementation consists of a stateful update function that can output events and update its state every time an input event is processed. For the fraud detection example, the update function would process bank transactions by checking if they are fraudulent and by constructing a sketch of the previously seen transactions, and fraud detection rules by using the sketch of previously seen transactions and the new rule to update the statistical fraud model. Second, the user provides a dependence relation that indicates the input events for which the processing order must be preserved, inducing a partial order on input events. For the current example, the user would simply indicate that fraud detection rule events depend on all other events. The final part of a specification consists of primitives that describe how to *fork* the state into two independent copies to allow for parallel processing and how to *join* two states when synchronization is required. These primitives abstractly represent splitting the computation into independent computations and merging the results, and are not tied to a specific implementation.

Given a DGS specification, the mapping to the synchronization plan in our architecture is given by a pluggable optimization component, which picks a synchronization plan based on information about the target execution environment, e.g. the number of processing workers and the location of the input streams. All of the induced plans are shown to be correct with respect to the sequential specification, so the optimizer is free to pick any of them without endangering correctness. As a starting point, we have developed a simple optimizer that tries to minimize the number of messages exchanged between different workers using information about the execution environment and the input streams. As a final step, the synchronization plan abstraction is deployed by the runtime system, which among other implementation details enforces the ordering of input events based on input dependencies, and is implemented in our DGSStream prototype.

4.3 Dependency-Guided Synchronization (DGS)

A DGS specification consists of three components: a *sequential specification*, a *dependence relation* on input events to enforce synchronization, and *fork* and *join* parallelization primitives. At the end of the section we define specification *consistency*, which are requirements on the *fork* and *join* functions to ensure that any parallelized implementation generated from the specification is equivalent to the sequential one. As specifications are given programmatically, we also refer to DGS as a *programming model* and to specifications as *programs*.

4.3.1 Walkthrough

For a simple but illustrative example, suppose that we want to implement a stream processing application that simulates a map from keys to counters, in which there are two types of input events: *increment* events, denoted $\mathtt{i}(k)$, and *read-reset* events, denoted $\mathtt{r}(k)$, where each event has an associated key k. On each increment event, the counter associated with that key should be increased by one, and on each read-reset event, the current value of the counter should be produced as output, and then the counter should be reset to zero. The "type" of an input event is called its *tag*, and in this example there are two tags. An input event may also carry a *payload* of data that is not used for parallelization. In this example all events have an empty payload.

Sequential specification. In our programming model, the user first provides a sequential implementation of the desired computation. A pseudocode version of the sequential specification for the example above is shown in Figure 15 (left). It consists of (i) the state type `State`, i.e. the map from keys to counters, (ii) the initial value of the state `init`, i.e. 0 for all keys, and (iii) a function `update`, which contains the logic for processing input events. Conceptually, the sequential specification describes how to process the data assuming it was all combined into a single sequential stream (e.g., sorted by system timestamp). For example, if the input stream consists of the events $\mathtt{i}(1), \mathtt{i}(2), \mathtt{r}(1), \mathtt{i}(2), \mathtt{r}(1)$, then the output would be 1 followed by 0, produced by the two $\mathtt{r}(1)$ (read-reset) events.

Dependence relation. To parallelize a sequential computation, the user needs to provide a dependence relation which encodes which events are independent, and thus can be processed in parallel, and which events are dependent, and therefore require synchronization. The dependence relation abstractly captures all the dependency patterns that appear in an application, inducing a partial order on input events. In this example, there are two forms of independence we want to expose. To begin with, *parallelization by key* is possible: the counter map could be partitioned so that events corresponding to different sets of keys are processed independently. Moreover, *parallelizing increments* on the counter of the same key is also possible. In particular, different sets of increments for the same key can be processed independently; we only need to aggregate the independent counts when a read-reset operation arrives. On the other hand, read-reset events are synchronizing for a particular key; their output is affected by the processing of increments as well as other read-reset events of that key.

We capture this combination of parallelization and synchronization requirements by defining the dependence relation `dependencies` in Figure 15, which is also visualized in Figure 16. In the specification, the set of tags may be *symbolic* (infinite): here `Tag` is parameterized by an integer `Key`. To allow for this, the dependence relation is formally a *predicate*

```
// Types
Key = Integer
Tag = i(Key) | r(Key)
Payload = ()
Event = (Tag, Payload)
State = Map(Key, Integer)




// Sequential Implementation
init: () -> State
init() =
    return emptyMap()
update: (State, Event) -> State
update(s, (i(k), ())) =
    s[k] = s[k] + 1;
    return s
update(s, (r(k), ())) =
    output s[k];
    s[k] = 0;
    return s
```

```
// Dependence Relation
dependencies: (Tag, Tag) -> Bool
dependencies(r(k1), r(k2)) = k1 == k2
dependencies(r(k1), i(k2)) = k1 == k2
dependencies(i(k1), r(k2)) = k1 == k2
dependencies(i(k1), i(k2)) = false

// Fork and Join
fork: (State, Tag -> Bool, Tag -> Bool) -> (State, State)
fork(s, pred1, pred2) =
    s1 = init(); s2 = init() // two forked states
    for k in keys(s):
        if pred1(r(k)):
            s1[k] = s[k]
        else: // pred2(r(k)) OR r(k) in neither
            s2[k] = s[k]
    return (s1, s2)
join: (State, State) -> State
join(s1, s2) =
    for k in keys(s2):
        s1[k] = s1[k] + s2[k]
    return s1
```

Figure 15: Specification in DGS of a map from keys to counters, which can process increment operations `i`(k) and read-reset operations `r`(k) for each key k. Erlang syntax has been edited for readability. We use `s[k]` as shorthand for getting key k from the map *or* the default value 0 if it is not present.

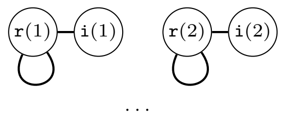


Figure 16: The `dependencies` relation from Figure 15 visualized as a graph, with 2 keys shown. Edges enforce synchronization requirements, while non-edges expose parallelism.

on pairs of tags, and is given programmatically as a function from pairs of `Tag` to `Bool`. For example, `dependencies(r(k1), r(k2))` (one of four cases) is given symbolically as equality comparison of keys, `k1 == k2`. The dependence relation should also be *symmetric*, i.e. `e1` is in `dependencies(e2)` if and only if `e2` is in `dependencies(e1)`.

Parallelization primitives: *fork* and *join*. While the dependence relation indicates the possibility of parallelization, it does not provide a mechanism for parallelizing state. The parallelization is specified using a pair of functions to `fork` one state into two, and to `join` two states into one. The fork function additionally takes as input two predicates of tags, such that the two predicates are *independent* (but not necessarily disjoint): every tag satisfying `pred1` is independent of every tag satisfying `pred2`. The contract is that after the state is forked into two independent states, each state will then *only be updated using events from the given set of tags.* A fork-join specification for our example is shown in fig. 15. The `join` function simply adds up the counts for each key to form the combined state. The `fork` function has to decide, for each key, which forked state to send the count to. Since read-reset operations `r(k)` are synchronizing and require knowing the total count, it does this by checking which forked state is responsible for processing read-reset operations, if

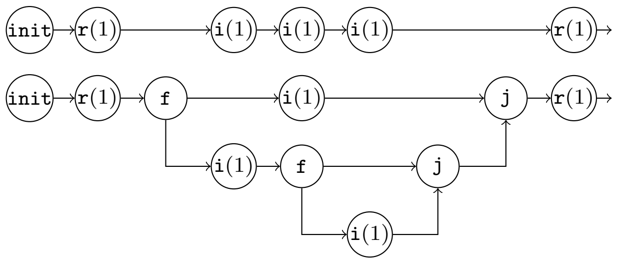


Figure 17: Example of a sequential and parallel execution of the program in fig. 15 on the input stream $\texttt{r}(1), \texttt{i}(1), \texttt{i}(1), \texttt{i}(1), \texttt{r}(1)$. Here $k = 1$ is a key, and `f` and `j` denote fork and join functions. ***Top:*** *Sequential execution: each item of the input stream is processed as an update to a global state.* ***Bottom:*** *One possible parallel execution: in between some updates, the state is forked or joined, and* `i` *events are processed as independent updates to one of the forked states.*

any.

It is necessary that the `fork` function have the predicates `pred1` and `pred2` as input, as otherwise we would not know how to partition the state `s` in case of *parallelization by key*. For example, if `pred1` contains all events of key 1 and `pred2` contains all events of key 2, then we need to propagate the count `s[1]` to the first forked state and the count `s[2]` to the second forked state. However, it is also possible that `pred1` and `pred2` are *not* disjoint, which is necessary to accommodate *parallelization by increment*. For example, `pred1` and `pred2` may both contain `i(3)` events. This is the reason that we include both `pred1` and `pred2`, even though only `pred1` is used in our example. Fortunately in this case, the independence requirement guarantees that neither forked state will be responsible for `r(3)` events (as `r(3)` is dependent on `i(3)`); in particular, neither forked state will be required to produce output for key 3. Therefore, we can choose to propagate the count `s[3]` to either state, knowing that the two forked counts will later be added back together in `join`. This illustrates that the fork function's input can indicate either a partitioning of input events, or parallelization on events on the same tag (or a combination of these across different keys).

4.3.2 Formal Definition

A DGS specification can be more general than we have discussed so far, because we allow for multiple *state types*, instead of just one. The initial state must be of a certain type, but forks and joins can convert from one state type to another: for example, forking a pair into its two components. Additionally, each state type can come with a *predicate* which restricts the allowed tags processed by a state of that type. The complete programming model is summarized in the following definition.

Definition 4.1 (DGS Specification)**.** Let `Pred(T)` be a given type of *predicates* on a type `T`, where predicates denote functions `T -> Bool`. A specification consists of the following components:

(1) A type of input events `Event = (Tag, Payload)`, where `Tag` is a type of tags.

(2) The dependence relation `dependencies: Pred(Tag, Tag)`, which is symmetric: `dependencies(tag1, tag2)` iff `dependencies(tag2, tag1)`.

(3) A type for output events `Out`.

(4) Finitely many state types `State_0`, `State_1`, etc.

(5) For each state type `State_i`, a predicate `pred_i: Pred(Tag)` — this specifies which input values this type of state can process — where `pred_0` = `true`.

(6) A sequential implementation, consisting of a single initial state `init: State_0` and for each state type `State_i`, a function `update_i: (State_i, Event) -> State_i`. The update also produces zero or more outputs, given formally by a function `out_i: (State_i, Event) -> List(Out)`.

(7) Any number of parallelization primitives, where each is either a *fork* or a *join*. A fork function has type `(State_i, Pred(Tag), Pred(Tag)) -> (State_j, State_k)`, and a join function has type `(State_j, State_k) -> State_i`, for some i, j, k.

The semantics of a specification can be visualized using wire diagrams, as in Figure 17. Computation proceeds from left to right. Each wire is associated with (i) a state (of type `State_i` for some i) and (ii) a predicate on tags (of type `Pred(Tag)`) which restricts the input events that this wire can process. Input events are processed as updates to the state, which means they take one input wire and produce one output wire, while forks take one input wire and produce two, and joins take two input wires and produce one. Notice that the same updates are present in both sequential and parallel executions. It is guaranteed in the parallel execution that *fork and join come in pairs*, like matched parentheses. Each predicate on tags that is given as input to the `fork` function indicates the set of input events that can be processed along one of the outgoing wires. Additionally, we require that updates on parallel wires must be on independent tags. In the example, the wire is forked into two parts and then forked again, and all three resulting wires process `i`(1) events. Note that `r`(1) events cannot be processed at that time because they are dependent on `i`(1) events. More specifically, we require that the predicate at each wire of type `State_i` implies `pred_i`, and that after each `fork` call, the predicates at each resulting wire denote independent sets of tags. This semantics is formalized in the following definition.

Definition 4.2 (DGS Semantics)**.** A *wire* is a triple $\langle$`State_i`, `pred`, `s`$\rangle$, where `State_i` is a state type, `s: State_i`, and `pred: Pred(Tag)` is a predicate of allowed tags, such that `pred` implies `pred_i`. We give the semantics of a specification through an inductively defined relation, which we denote $\langle \texttt{State}, \texttt{pred}, \texttt{s}\rangle \xrightarrow[\texttt{v}]{\texttt{u}} \langle \texttt{State}, \texttt{pred}, \texttt{s'}\rangle$, where $\langle$`State`, `pred`, `s`$\rangle$ and $\langle$`State`, `pred`, `s'`$\rangle$ are the starting and ending wires (with the same state type and predicate), `u: List(Event)` is an input stream, and `v: List(Out)` is an output stream. Let `l1 + l2` be list concatenation, and define `inter(l, l1, l2)` if `l` is some interleaving of `l1` and `l2`. For `e1`, `e2: Event`, let `indep(e1, e2)` denote that `e1` and `e2` are not dependent, i.e. `not(dependencies(fst(e1),fst(e2)))`. There are two base cases and two inductive cases.

- (Emp) For any `State`, `pred`, and `s`, $\langle \texttt{State}, \texttt{pred}, \texttt{s}\rangle \xrightarrow[\texttt{[]}]{\texttt{[]}} \langle \texttt{State}, \texttt{pred}, \texttt{s}\rangle$.
- (Upd) For any `State`, `pred`, `s`, and any `e: Event`, if the tag of `e` satisfies `pred`, i.e. `pred(fst(e))`, then $\langle \texttt{State}, \texttt{pred}, \texttt{s}\rangle \xrightarrow[\texttt{out(s, e)}]{\texttt{[e]}} \langle \texttt{State}, \texttt{pred}, \texttt{update(s, e)}\rangle$.
- (Seq) For any `State`, `pred`, `s`, `s'`, `s''`, `u`, `v`, `u'`, and `v'`, if $\langle \texttt{State}, \texttt{pred}, \texttt{s}\rangle \xrightarrow[\texttt{v}]{\texttt{u}} \langle \texttt{State}, \texttt{pred}, \texttt{s'}\rangle$ and $\langle \texttt{State}, \texttt{pred}, \texttt{s'}\rangle \xrightarrow[\texttt{v'}]{\texttt{u'}} \langle \texttt{State}, \texttt{pred}, \texttt{s''}\rangle$, then $\langle \texttt{State}, \texttt{pred}, \texttt{s}\rangle \xrightarrow[\texttt{v + v'}]{\texttt{u + u'}} \langle \texttt{State}, \texttt{pred}, \texttt{s''}\rangle$.

- (Par) Lastly, for any `State`, `State1`, `State2`, `pred`, `pred1`, `pred2`, `s`, `s1'`, `s2'`, `u`, `u1`, `u2`, `v`, `v1`, `v2`, `fork`, and `join`, suppose that (the conjunction) `pred1(e1)` and `pred2(e2)` implies `indep(e1, e2)`, `pred1` implies `pred`, and `pred2` implies `pred`. Let `fork(s, pred1, pred2) = (s1, s2)` and `join(s1', s2') = s'`, and suppose that `inter(u, u1, u2)` and `inter(v, v1, v2)`. If $\langle \texttt{State1}, \texttt{pred1}, \texttt{s1} \rangle \xrightarrow[\texttt{v1}]{\texttt{u1}} \langle \texttt{State1}, \texttt{pred1}, \texttt{s1'} \rangle$ and $\langle \texttt{State2}, \texttt{pred2}, \texttt{s2} \rangle \xrightarrow[\texttt{v2}]{\texttt{u2}} \langle \texttt{State2}, \texttt{pred2}, \texttt{s2'} \rangle$ then $\langle \texttt{State}, \texttt{pred}, \texttt{s} \rangle \xrightarrow[\texttt{v}]{\texttt{u}} \langle \texttt{State}, \texttt{pred}, \texttt{s'} \rangle$.

The *semantics* $[\![S]\!]$ of the specification S is the set of pairs $(\texttt{u}, \texttt{v})$ of an input stream `u` and an output stream `v` such that $\langle \texttt{State\_0}, \texttt{true}, \texttt{init} \rangle \xrightarrow[\texttt{v}]{\texttt{u}} \langle \texttt{State\_0}, \texttt{true}, \texttt{s'} \rangle$ for some final state `s'`.

4.3.3 Consistency of Specifications

Any parallel execution is guaranteed to preserve the sequential semantics, i.e. processing all input events *in order* using the `update` function, as long as the following *consistency conditions* are satisfied. The sufficiency of these conditions is shown in Theorem 4.4, which states that *consistency implies determinism up to output reordering*. This is a key step in the end-to-end proof of correctness in Section 4.4.5. Consistency can be thought of as analogous to the commutativity and associativity requirements for a MapReduce program to have deterministic output [25]: just as with MapReduce programs, the implementation does *not* assume the conditions are satisfied, but if not the semantics will be dependent on how the computation is parallelized.

Let us briefly illustrate the consistency conditions for our running example (Figure 15). If `e` is an increment event, then condition (C1) captures the fact that counting can be done in parallel: it reduces to `(s1[k] + s2[k]) + 1` $=$ `(s1[k] + 1) + s2[k]`. Condition (C2) captures the fact that we preserve total count across states when forking: it reduces to `s[k] + 0` $=$ `s[k]`. Condition (C3) would not be valid for *general* events `e1`, `e2`, because a read-reset event does not commute with an increment of the same key (`s[k] + 1` $\neq$ `s[k]`), hence the restriction that `indep(e1, e2)`. Finally, one might think that a variant of (C1) should hold for `fork` in addition to `join`, but this turns out not to be the case: for example, starting from `s[k] = 100`, an increment followed by a fork might yield the pair of counts $(101, 0)$, while a fork followed by an increment might yield $(100, 1)$. It turns out however that commutativity only with joins, and not with forks, is enough to imply Theorem 4.4.

Definition 4.3 (Consistency)**.** A specification is *consistent* if the following equations always hold:

$$\begin{aligned}
\texttt{join(update(s1, e), s2)} &= \texttt{update(join(s1, s2), e)} && \text{(C1)}\\
\texttt{join(fork(s, pred1, pred2))} &= \texttt{s} && \text{(C2)}\\
\texttt{update(update(s, e1), e2))} &= \texttt{update(update(s, e2), e1))}. && \text{(C3)}
\end{aligned}$$

Equation (C1) is over all join functions `join: (State_j, State_k) -> State_i`, events `e: Event` such that `pred_i(e)` and `pred_j(e)`, and states `s1: State_j`, `s2: State_k`, where `update` denotes the update function on the appropriate type. Additionally the corresponding output on both sides must be the same: `out(s1, e)` $=$ `out(join(s1, s2))`. Equation (C2) is over all fork functions `fork: (State_i, Pred(Tag), Pred(Tag)) -> (State_j, State_k)` join functions `join: (State_j, State_k) -> State_i`, states `s: State_i`, and predicates `pred1` and `pred2`. Equation (C3) is over all state types `State_i`, states `s: State_i`, and *independent* events `indep(e1, e2)` such that `pred_i(e1)` and `pred_i(e2)`. As with (C1), we also require that the outputs agree: `out(s, e1) + out(update(s, e1), e2) = out(update(s, e2), e1) + out(s, e2)`.

Theorem 4.4. *If S is consistent, then S is deterministic up to output reordering: that is, for all $(\mathtt{u}, \mathtt{v}) \in [\![S]\!]$,* `set(v) = set(spec(u))`.

Proof. We show by induction on the semantics in Theorem 4.2 that every wire diagram is equivalent (up to output reordering) to the sequential sequence of updates. The (Seq) inductive step is direct by associativity of function composition on the left and right sequence of updates (no commutativity of updates is required). For the (Par) inductive step, we replace the two parallel wires with sequential wires, then apply (C1) repeatedly on the last output to move it outside of the parallel wires, then finally apply (C2) to reduce the now trivial parallel wires to a single wire. □

4.4 Distributed Implementation

This section describes the distributed runtime for DGS. In particular, we describe *synchronization plans*, which represent streaming program implementations, and our framework for generating them from the given DGS specification in section 4.3. Generation of an implementation can be conceptually split in two parts, the first ensuring correctness and the second affecting performance. First a specification S induces a set of S-*valid*, i.e. correct with respect to it, synchronization plans. Choosing one of those plans is then an independent optimization problem that does not affect correctness and can be delegated to a separate optimization component (section 4.4.3). Finally, the workers in synchronization plans need to process some incoming events in order while some can be processed out of order (depending on the dependence relation). We propose a selective reordering technique (section 4.4.4) that can be used in tandem with heartbeats to address this ordering issue. We tie everything together by providing an end-to-end proof that the implementation is correct with respect to a consistent specification S (and importantly, independent of the synchronization plan chosen as long as it is S-valid) in section 4.4.5. Before describing the separate framework components, we first articulate the necessary underlying assumptions about input streams in section 4.4.1.

4.4.1 Preliminaries

In our model the input is partitioned in some number of input streams that could be distributed, i.e. produced at different locations. We assume that the implementation has access to *some* ordering relation $\mathcal{O}$ on pairs of input events (also denoted $<_{\mathcal{O}}$), and the order of events is increasing along each input stream. This is necessary for cases where the *user-written* specification requires that events arriving in different streams are dependent, since it allows the implementation to have progress and process these dependent events in order. Concretely, in our setting $\mathcal{O}$ is implemented using *event timestamps*. Note that these timestamps do not need to correspond to real time, if this is not required by the application. In cases where real-time timestamps are required, this can be achieved with well-synchronized clocks, as has been done in other systems, e.g. Google Spanner [24].

Each event in each input stream is given by a triple $\langle \sigma, ts, v \rangle$, containing an *implementation tag* $\sigma \in$ `ITag`, a timestamp ts, and its payload v. Each implementation tag corresponds to one or more tags in the specification in section 4.3, but is also augmented with an identifier of the input stream. This ensures that each implementation tag is unique to a particular

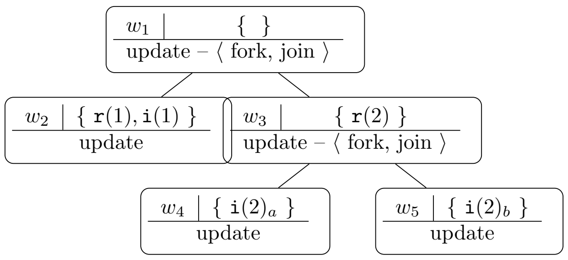

w_1 | { }
update – ⟨ fork, join ⟩
w_2 | { $\mathtt{r}(1), \mathtt{i}(1)$ }
update
w_3 | { $\mathtt{r}(2)$ }
update – ⟨ fork, join ⟩
w_4 | { $\mathtt{i}(2)_a$ }
update
w_5 | { $\mathtt{i}(2)_b$ }
update

Figure 18: Example synchronization plan derived from the specification in Figure 15 for two keys $k = 2$ and five input streams $\mathtt{r}(1), \mathtt{i}(1), \mathtt{r}(2), \mathtt{i}(2)_a, \mathtt{i}(2)_b$. Implementation tags $\mathtt{i}(2)_a, \mathtt{i}(2)_b$ both correspond to the tag $\mathtt{i}(2)$ but are separate because they arrive in different input streams. Each node contains the set of implementation tags that it is responsible for, an update function, and a pair of fork-join functions if it has children.

input stream, while a tag can appear in multiple streams (in all of the streams with a corresponding implementation tag). To simplify the presentation in this section, we assume that the set of implementation tags is finite, and this is also currently required in our prototype implementation. A dependence relation `dependencies` straightforwardly lifts from tags in the specification to predicates over tags and to implementation tags $D_I \subseteq \mathtt{ITag} \times \mathtt{ITag}$.

4.4.2 Synchronization Plans

Synchronization plans are binary tree structures that encode (i) parallelism: each node of the tree represents a sequential thread of computation that processes input events; and (ii) synchronization: parents have to synchronize with their children to process an event. Synchronization plans are inspired by prior work on concurrency models including fork-join concurrency [34, 45] and CSP [39]. An example synchronization plan for the specification in Figure 15 is shown in Figure 18. Each node has an id w_i, contains the set of implementation tags that it is responsible for, a state type (which is omitted here since there is only one state type `State`), and a triple of update, fork, join functions. Note that a node is responsible to process events from its set of implementation tags, but can potentially handle all the implementation tags of its children. The leaves of the tree can process events independently without blocking, while parent nodes can only process an input event if their children states are joined. Nodes without a ancestor-descendant relationship do not directly communicate, but instead learn about each other when their common ancestor joins and forks back the state.

Definition 4.5 (Synchronization Plans)**.** Given a specification S, a synchronization plan is a pair $(\overline{w}, \text{par})$, which includes a set of workers $\overline{w} = \{w_1, \ldots, w_N\}$, where $w_i \in W$, together with a parent relation $\text{par} \subseteq W \times W$, the transitive closure of which is an ancestor relation denoted as $\text{anc} \subseteq W \times W$. Workers have three components: (i) a state type $w.\mathsf{state}$ which references one of the state types of S, (ii) a set of implementation tags $w.\mathsf{itags}$ that the worker is responsible for, and (iii) an update $w.\mathsf{update}$ and possibly a fork-join pair $w.\mathsf{fork}$ and $w.\mathsf{join}$ if it has children.

We now define what it means for a synchronization plan to be *S-valid*. Intuitively, an S-valid plan is well-typed with respect to specification S, and the workers that do not have an ancestor-descendant relationship should handle independent and disjoint implementation tags. S-validity is checked syntactically by our framework and is a necessary requirement to ensure that the generated implementation is correct (see Section 4.4.5).

Definition 4.6 (S-valid)**.** Formally, an S-valid plan has to satisfy the following syntactic properties: (V1) The state `State_i` $= w.\mathsf{state}$ of each worker w should be consistent with its update-fork-join triple and its implementation tags. The update must be defined on the node state type, i.e., $w.\mathsf{update}$ `: (State_i, Event) -> State_i`, `State_i` should be able to handle the specification tags corresponding to $w.\mathsf{itags}$, and the fork-join pair should be defined for the state types of the node and its children. (V2) Each pair of nodes that do not have an ancestor-descendant relation, should handle pairwise independent and disjoint implementation tag sets, i.e., $\forall w, w' \in \mathsf{anc}(w, w'), w.\mathsf{itags} \cap w'.\mathsf{itags} = \emptyset$.

As an example, the synchronization plan shown in Figure 18 satisfies both properties; there is only one state type that handles all tags and implementation tag sets are disjoint for ancestor-descendants. The second property (V2) represents the main idea behind our execution model; independent events can be processed by different workers without communication. Intuitively, in the example in Figure 18, by assigning the responsibility for handling tag $\mathtt{r}(2)$ to node w_3, its children can independently process tags $\mathtt{i}(2)_a, \mathtt{i}(2)_b$ that are dependent on $\mathtt{r}(2)$.

4.4.3 Optimization Problem

As described in the previous section, a set of S-valid synchronization plans can be derived from a DGS specification S. This decouples the optimization problem of finding a well-performing implementation, allowing it to be addressed by an independent optimizer, which takes as input a description of the available computer nodes and the input streams. This design means that different optimizers could be implemented for different performance metrics (e.g. throughput, latency, network load, energy consumption).

The design space for optimizers is vast and thoroughly exploring it is outside of the scope of this work. For evaluation purpose, we have implemented an optimizer based on a simple heuristic: it tries to generate a synchronization plan with a separate worker for each input stream, and then tries to place these workers in a way that minimizes communication between them. This optimizer assumes a network of computer nodes and takes as input estimates of the input rates at each computer node. It searches for an S-valid synchronization plan that maximizes the number of events that are processed by leaves; since leaves can process events independently without blocking. The optimizer first uses a greedy algorithm that generates a graph of implementation tags (where the edges are dependencies) and iteratively removes the implementation tags with the lowest rate until it ends up with at least two disconnected components.

Example 4.7. For an example optimization run, consider Figure 18, and suppose that $\mathtt{r}(2)$ has a rate of 10 and arrives at node E_0, $\mathtt{r}(1)$, $\mathtt{i}(1)$ have rates of 15 and 100 respectively and arrive at node E_1, $\mathtt{i}(2)_a$ has rate 200 and arrives at node E_2, and $\mathtt{i}(2)_b$ has rate 300 and arrives at node E_3. Since events of different keys are independent, there are two connected components in the initial graph—one for each key. The optimizer starts by separating them into two subtrees. It then recurses on each disconnected component, until there is no implementation tag left, ending up with the tree structure shown in Figure 18. Finally, the optimizer exhaustively tries to match this implementation tag tree, with a sequence of forks, in order to produce a valid synchronization plan annotated with state types, updates, forks, and joins. This generates the implementation in Figure 19.

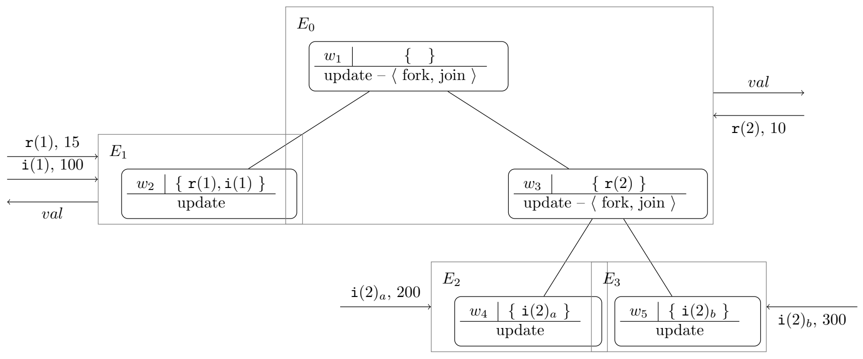


Figure 19: Example distributed implementation generated in Theorem 4.7. The large gray rectangles E_0, E_1, E_2, E_3 represent physical nodes and the incoming arrows represent input streams and their relative rates.

4.4.4 Implementation

Each node of the synchronization plan can be separated into two components: an event processing component that is responsible for computation via executing `update`, `fork`, and `join` calls; and a mailbox component that is responsible for enforcing ordering requirements and synchronization.

Event Processing The worker processes execute the `update`, `fork`, and `join` functions associated with the tree node. Whenever a worker is handed a message by its mailbox, it first checks if it has any active children, and if so, it sends them a join request and waits until it receives their responses. After receiving these responses, it executes the join function to combine their states, executes the update function on the received event, and then executes the fork function on the new state, and sends the resulting states to its children. In contrast, a leaf worker just executes the update function on the received event.

Event Reordering The mailbox of each worker ensures that it processes incoming dependent events in the correct order by implementing the following selective reordering procedure. Each mailbox contains an event buffer and a timer for each implementation tag. The buffer holds the events of a specific tag in increasing order of timestamps and the timer indicates the latest timestamp that has been encountered for each tag. When a mailbox receives an event $\langle \sigma, ts, v \rangle$ (or a join request), it follows the procedure described below. It first inserts it in the corresponding buffer and updates the timer for σ to the new timestamp ts. It then initiates a cascading process of releasing events with tags σ' that depend on σ. During that process all dependent tags σ' are added to a dependent tag workset, and the buffer of each tag in the workset is checked for potential events to release. An event $e = \langle \sigma, ts, v \rangle$ can be released to the worker process if two conditions hold. The timers of its dependent tags are higher than the timestamp ts of the event (which means that the mailbox has already seen all dependent events up to ts, making it safe to release e), and the earliest event in each buffer that σ depends on should have a timestamp $ts' > ts$ (so that events are processed in order). Whenever an event with tag σ is released, all its dependent tags are added to the workset and this process recurses until the tag workset is empty.

Heartbeats As discussed in section 4.4.1, a dependence between two implementation tags σ_1 and σ_2 requires the implementation to process any event $\langle \sigma_1, t_i, v_i \rangle$ after processing all events $\langle \sigma_2, t_j, v_j \rangle$ with $t_j \leq t_i$. However, with the current assumptions on the input streams, a mailbox has to wait until it receives the earliest event $\langle \sigma_2, t_j, v_j \rangle$ with $t_j > t_i$, which could arbitrarily delay event processing. We address this issue with *heartbeat* events, which are system events that represent the absence of events on a stream. These are commonly used in other systems under different names, e.g. heartbeats [41], punctuation [62], watermarks [22], or pulses [60]. Heartbeats are generated periodically by the stream producers, and are interleaved together with standard events of input streams. When a heartbeat event $\langle \sigma, t \rangle$ first enters the system, it is broadcast to all the worker processes that are descendants of the worker that is responsible for tag σ. Each mailbox that receives the heartbeat updates its timers and clears its buffers as if it has received an event of $\langle \sigma, t, v \rangle$ without adding the heartbeat to the buffer to be released to the worker process.

The latency (and overall performance) of the system directly depends on the rate of heartbeats for each tag. More precisely, the latency associated with an event cannot be lower than the period (1/rate) of all the streams that it depends on. However, higher hearbeat rates also negatively affect throughput and network load.

4.4.5 Proof of Correctness

We show that *any* implementation produced by the end-to-end framework is correct according to the semantics of the programming model. First, Theorem 4.8 formalizes the assumptions about the input streams outlined in Section 4.4.1, and Theorem 4.9 defines what it means for an implementation to be correct with respect to a sequential specification. Our definition is inspired by the classical definitions of distributed correctness based on observational trace semantics (e.g., [47]). However, we focus on how to interpret the independent input streams as a sequential input, in order to model possibly synchronizing and order-dependent stream processing computations.

Definition 4.8. A *valid input instance* consists of k input streams (finite sequences) `u_1`, `u_2`, ..., `u_k` of type `List(Event | Heartbeat)`, and an order relation $\mathcal{O}$ on input events and heartbeats, with the following properties. (1) *Monotonicity:* for all i, `u_i` is in strictly increasing order according to $<_{\mathcal{O}}$. (2) *Progress:* for all i, for each input event (non-heartbeat) `x` in `u_i`, for every other stream j there exists an event or heartbeat `y` in `u_j` such that `x` $<_{\mathcal{O}}$ `y`.

Given a DGS specification S, the *sequential specification* is a function `spec: List(Event) -> List(Out)`, obtained by applying only `update` functions and no `fork` and `join` calls. The output specified by `spec` is produced incrementally (or *monotonically*): if `u` is a prefix of `u'`, then `spec(u)` is a subset of `spec(u')`. Define the *sort* function `sort`$_{\mathcal{O}}$: `List(List(Event | Heartbeat)) -> List(Event)` which takes k sorted input event streams and sorts them into one sequential stream, according to the total order relation $\mathcal{O}$, and drops heartbeat events.

Definition 4.9. Given a sequential specification `spec: List(Event) -> List(Out)`, a distributed implementation is *correct* with respect to `spec` if for every valid input instance $\mathcal{O}$, `u_1`, ..., `u_k`, the set of outputs produced by the implementation is equal to `set(spec(sort`$_{\mathcal{O}}$`(u_1, ..., u_k)))`.

We show that our framework is correct according to Theorem 4.9. The key assumptions used are:

(1) The specification given by the programmer is consistent, as defined in Section 4.3.3.

(2) The input streams constitute a valid input instance, as defined in Theorem 4.8.

(3) The synchronization plan that is chosen by the optimizer is valid, as defined in Section 4.4.2.

(4) Messages that are sent between two processes in the system arrive in order and are always received. This last assumption is ensured by the Erlang runtime.

The proof decomposes the implementation into the mailbox implementations and the worker implementations, which are both sets of processes that given a set of input streams produce a set of output streams. We first show that the mailboxes transform a valid input instance to a worker input that is correct up to reordering of independent events. Then we show that given a valid worker input, the workers processes' collective output is correct. The second step relies on Theorem 4.4, from the previous section, as well as Theorem 4.12 which ties this to the mailbox implementations, showing that the implementation produces a valid wire diagram according to the formal semantics of the specification S. Given a valid input instance u and a computation specification S that contains in particular a sequential specification `spec: List(Event) -> List(Out)`, we want to show that any implementation f that is produced by our framework is correct according to Theorem 4.9.

Definition 4.10. For a valid input instance u, a worker input $m_u = (m_1, ..., m_N)$ is *valid* with respect to u if $m_i \in \text{reorder}_{D_I}(\text{filter}(\text{rec}_{w_i}, \text{sort}_O(u)))$, where reorder_{D_I} is the set of reorderings that preserve the order of dependent events, $\text{rec}_w(\langle \sigma, t, p\rangle) = \sigma \in \text{atags}(w) \cup w.\mathsf{itags}$ is a predicate of the messages that each worker w receives, and atags is the set of implementation tags that are handled by a workers ancestors, and is given by $\text{atags}_w = \{w'.\mathsf{itags} : \forall w' \in \text{anc}(w', w)\}$.

Lemma 4.11. Given a valid input instance u, any worker input m produced by the mailbox implementations (u, m) in f is valid with respect to u.

Proof. By induction on the input events of one mailbox and using assumptions (2)-(4). □

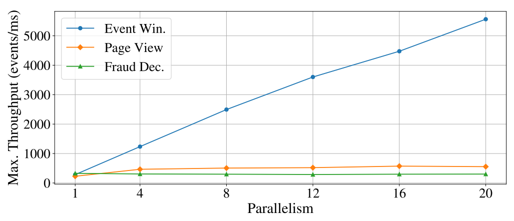
Each worker w runs the update function on each event $e = \langle \sigma, t, p\rangle$ on its stream that it is responsible for $\sigma \in w.\mathsf{itags}$ possibly producing output `o: List(Out)`. For all events in the stream that one of its ancestors is responsible for, it sends its state to its parent worker and waits until it receives an updated one. The following key lemma states that this corresponds to a particular wire diagram in the semantics of the specification.

Lemma 4.12. Let m be the worker input to f for specification S on input $\mathtt{u}$, and let $\mathtt{o}_i$: `List(Out)` be the stream of output events produced by worker i on input m_i. Then there exists `v: List(Out)` such that $\mathtt{inter}(\mathtt{v}, \mathtt{o}_1, \mathtt{o}_2, \ldots, \mathtt{o}_N)$ and $(\mathtt{u}, \mathtt{v}) \in [\![S]\!]$.

Proof. By induction on the worker input and using assumption (3), in particular validity condition (V1), we first show that the worker input corresponds to a wire diagram, in particular we show that $\langle \mathtt{State\_0}, \mathtt{true}, \mathtt{s}\rangle \xrightarrow[\mathtt{v}]{\mathtt{u}'} \langle \mathtt{State\_0}, \mathtt{true}, \mathtt{s'}\rangle$ where $\mathtt{v}$ is an interleaving of $\mathtt{o}_1, \ldots, \mathtt{o}_N$ and $\mathtt{u}'$ is *any* interleaving of the events $\mathtt{u}'_i$ processed by each mailbox, namely $\text{filter}(\text{rec}_{w_i}, w_i.\mathsf{itags})$. Applying theorem 4.11, $\mathtt{u}$ is one possible interleaving of the events $\mathtt{u}'_i$ and hence we conclude that $\langle \mathtt{State\_0}, \mathtt{true}, \mathtt{s}\rangle \xrightarrow[\mathtt{v}]{\mathtt{u}} \langle \mathtt{State\_0}, \mathtt{true}, \mathtt{s'}\rangle$, thus $(\mathtt{u}, \mathtt{v}) \in [\![S]\!]$. □

Combining Theorem 4.12 and Theorem 4.4 then yields the end-to-end correctness theorem.

Theorem 4.13 (Implementation Correctness)**.** *f is correct according to theorem 4.9.*



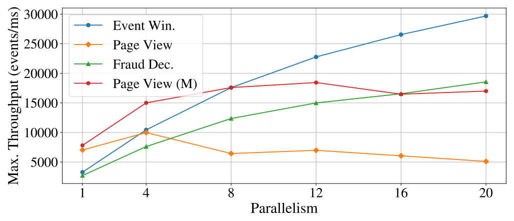


Figure 20: Flink (left) and Timely (right) implementations maximum throughput increase with increasing parallel nodes for the three applications that require synchronization.

```
updates.broadcast().filter(move |x| {
    x.name == page_partition_function(NUM_PAGES, worker_index)
});
```

Figure 21: Snippet of the manual (M) Timely implementation of the Page View example. This violates **PIP2** since it explicitly references the physical partitioning of input streams (`page_partition_function`) and the index of the worker that processes each stream (`worker_index`).

4.5 Evaluation

We show some highlights from the evaluation of DGS in Figure 20, Figure 21, Figure 22, and Figure 23.

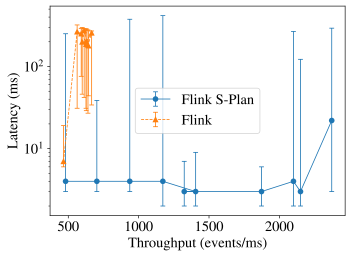
5 Monitoring

In this section we describe data transducers, an intermediate representation for modeling stream processing operators as finite state transducers over data words [3, 18, 1]. Data transducers support succinct constructions, making them compositional. We also describe the QRE-Past monitoring language, which can be used for monitoring stream processing applications.

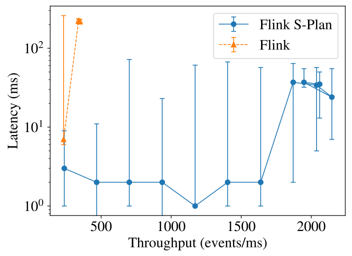
5.1 Data Transducers

To model data streams we use *data words*. Let $\mathbb{D}$ be a (possibly infinite) set of ***data values***, such as the set of integers or real numbers, and let Σ be a finite set of ***tags***. Then a ***data word*** is a sequence of tagged data values $\mathbf{w} \in (\Sigma \times \mathbb{D})^*$. We write $\mathbf{w} \downarrow \Sigma$ to denote the projection of $\mathbf{w}$ to a string in Σ^*. We use bold $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ to denote data words. We reserve non-bold u, v, w for plain strings of tags in Σ^*. We write d, d_i for elements of $\mathbb{D}$. We use σ to denote an arbitrary tag in Σ, and in the examples we write particular tags in typewriter font, e.g. `a`, `b`.

A ***signature*** is a tuple $(\mathbb{D}, \mathsf{Op})$, where $\mathbb{D}$ is a set of data values and Op is a set of ***allowed operations***. Each operation has an ***arity*** $k \geq 0$ and is a function from $\mathbb{D}^k$ to $\mathbb{D}$. We use Op_k to denote the k-ary operations. For instance, if $\mathbb{D}$ is all 64-bit integers, we might support 64-bit arithmetic, as well as integer division and equality tests. Alternatively we might have $\mathbb{D} = \mathbb{N}$ with the operations $+$ (arity 2), min (arity 2), and 0 (arity 0). In general, we may have arbitrary user-defined operations on $\mathbb{D}$. Given a signature $(\mathbb{D}, \mathsf{Op})$,



(a) Page-view Join.



(b) Fraud Detection.

Figure 22: Throughput (x-axis) and 10th, 50th, 90th percentile latencies on the y-axis for increasing input rates (from left to right) and 12 parallel nodes. Flink represents the parallel implementation produced automatically, and Flink S-Plan represents the synchronization plan implementation.

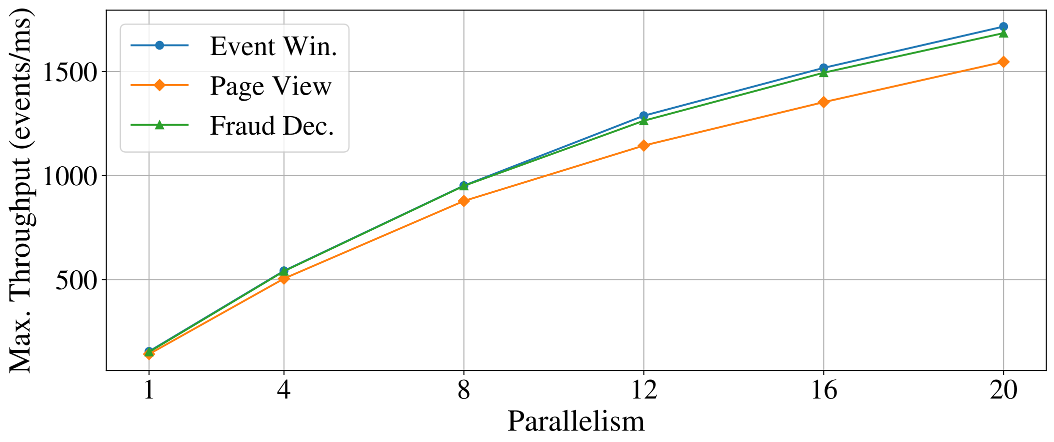


Figure 23: DGSStream maximum throughput increase with increasing parallel nodes for the three applications that require synchronization.

and a collection of variables Z, the set of ***terms*** $\mathsf{Tm}[Z]$ consists of all syntactically correct expressions with free variables in Z, using operations Op. So $\min(x,0)+\min(y,0)$ and $x+x$ are terms over the signature $(\mathbb{N},\{+,\min,0\})$ with $Z=\{x,y\}$.

We define two special values in addition to the values in $\mathbb{D}$: $\bot$ denotes ***undefined*** and $\top$ denotes ***conflict***. We let $\overline{\mathbb{D}} := \mathbb{D}\cup\{\bot,\top\}$ be the set of ***extended data values***, and refer to elements of $\mathbb{D}$ as ***defined***. We lift Op to operations on $\overline{\mathbb{D}}$ by thinking of $\bot$ as the empty multiset, elements of $\mathbb{D}$ as singleton multisets, and $\top$ as any multiset of two or more data values. The specific behavior of $op \in \mathsf{Op}$ on values in $\overline{\mathbb{D}}$ is illustrated in the table below for the case $op \in \mathsf{Op}_2$. We also define a ***union*** operation $\sqcup : \overline{\mathbb{D}}\times\overline{\mathbb{D}}\to\overline{\mathbb{D}}$: if either of its arguments is undefined it returns the other one, and in all other cases it returns conflict. This represents multiset union. Note that $d_1 \sqcup d_2 = \top$ even if $d_1 = d_2$. This is essential: it guarantees that for all operations on extended data values, whether the result is undefined, defined, or conflict can be determined from knowing only whether the inputs are undefined,

defined, or conflict.

$\sqcup$	$\bot$	d_2	$\top$
$\bot$	$\bot$	d_2	$\top$
d_1	d_1	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$

op	$\bot$	d_2	$\top$
$\bot$	$\bot$	$\bot$	$\bot$
d_1	$\bot$	$op(d_1, d_2)$	$\top$
$\top$	$\bot$	$\top$	$\top$

$\overline{\mathbb{D}}$ is a *complete lattice*, partially ordered under the relation $\leq$ which is defined by $\bot \leq d \leq \top$ for all $d \in \mathbb{D}$, and distinct elements $d, d' \in \mathbb{D}$ are incomparable. For a finite set X, we write the set of functions $X \to \overline{\mathbb{D}}$ as $\overline{\mathbb{D}}^X$; its elements are un-tagged ***data vectors***, denoted $\mathbf{x}$, $\mathbf{y}$. The partial order extends coordinate-wise to an ordering $\mathbf{x} \leq \mathbf{y}$ on data vectors $\mathbf{x}, \mathbf{y} \in \overline{\mathbb{D}}^X$. All operations in Op are *monotone increasing* w.r.t. this partial order. Union ($\sqcup$) is commutative and associative, with identity $\bot$ and absorbing element $\top$, and *all* k-ary operations distribute over it.

Syntax. Let $(\mathbb{D}, \mathsf{Op})$ be a fixed signature. A ***data transducer (DT)*** is a 5-tuple $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$, where:

- Q is a finite set of ***state variables*** (***states*** for short) and Σ is a finite set of ***tags***. We write Q' for a copy of the variables in Q: for $q \in Q$, $q' \in Q'$ denotes the copy. When the states of the DT are updated, q' will be the new, updated value of q.
- Δ is a finite set of ***transitions***, where each transition is a tuple (σ, X, q', t).
  - $\sigma \in \Sigma \cup \{\mathtt{i}\}$, where $\mathtt{i} \notin \Sigma$, and if $\sigma = \mathtt{i}$ this is a special *initial transition.*
  - $X \subseteq Q \cup Q'$ is a set of *source variables* and $q' \in Q'$ is the *target variable.*
  - $t \in \mathsf{Tm}[X \cup \{\mathsf{cur}\}]$ gives a new value of the target variable given values of the source variables and given the value of "cur", which represents the current data value in the input data word. Assume that $\mathsf{cur} \notin X$. We allow X to include some variables not used in t. For initial transitions, we additionally require that $X \subseteq Q'$ and that cur does not appear in t.
- $I \subseteq Q$ is a set of *initial states* and $F \subseteq Q$ is a set of *final states.*

The ***number of states*** of $\mathcal{A}$ is $|Q|$. The ***size*** of $\mathcal{A}$ is the the number of states plus the total length of all transitions (σ, X, q', t), which includes the length of description of all the terms t.

Semantics. The input to a DT has two components. First, an ***initial vector*** $\mathbf{x} \in \overline{\mathbb{D}}^I$, which assigns an extended data value to each initial state. Second, an ***input data word*** $\mathbf{w} \in (\Sigma \times D)^*$, which is a sequence of tagged data values to be processed by the transducer. On input $(\mathbf{x}, \mathbf{w})$, the DT's final ***output vector*** is an extended data value at each of its final states. Thus, the semantics of $\mathcal{A}$ will be

$$[\![\mathcal{A}]\!] : \overline{\mathbb{D}}^I \times (\Sigma \times \mathbb{D})^* \to \overline{\mathbb{D}}^F.$$

A ***configuration*** is a vector $\mathbf{c} \in \overline{\mathbb{D}}^Q$. For every $\sigma \in \Sigma$, the set of transitions (σ, X, q', t) collectively define a function $\Delta_\sigma : \overline{\mathbb{D}}^Q \times \mathbb{D} \to \overline{\mathbb{D}}^Q$: given the current configuration and the current data value from the input data word, Δ_σ produces the next configuration. We define

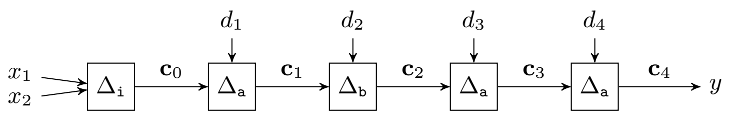


Figure 24: Example evaluation of a data transducer $\mathcal{A}$ with two initial states and one final state on initial vector $(\mathbf{x}_1, \mathbf{x}_2)$ and an input data word $\mathbf{w}$ consisting of four characters (tagged data values): $(\mathtt{a}, d_1)$, $(\mathtt{b}, d_2)$, $(\mathtt{a}, d_3)$, $(\mathtt{a}, d_4)$, to produce output y. Here $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$, and $\mathbf{c}_4$ are configurations; $d_i \in \mathbb{D}$; and $x_1, x_2, y \in \overline{\mathbb{D}}$. Each Δ_σ is a set of transitions, collectively describing the next configuration in terms of the previous one.

$\Delta_\sigma(\mathbf{c}, d)(q) := \mathbf{c}'(q')$, where $\mathbf{c}' \in \overline{\mathbb{D}}^{Q \cup Q' \cup \{\mathsf{cur}\}}$ is the *least vector* satisfying $\mathbf{c}'(\mathsf{cur}) = d$; for all $q \in Q$, $\mathbf{c}'(q) = \mathbf{c}(q)$; and

$$\text{for all } q' \in Q', \quad \mathbf{c}'(q') = \bigsqcup_{(\sigma, X, q', t) \in \Delta} [\![t]\!](\mathbf{c}'|_X), \tag{1}$$

where we define $[\![t]\!](\mathbf{c}'|_X)$ to be $\perp$ if there exists $x \in X$ such that $\mathbf{c}'(x) = \perp$; otherwise, $\top$ if there exists $x \in X$ such that $\mathbf{c}'(x) = \top$; otherwise, if all variables in X are defined, then $[\![t]\!](\mathbf{c}'|_X)$ is the value of the expression t with variables assigned the values in $\mathbf{c}'$. So, $[\![t]\!](\mathbf{c}'|_X)$ produces $\perp$ or $\top$ if some variable in X is $\perp$ or $\top$. The above union is over all transitions with label σ and target variable q'. Since $\overline{\mathbb{D}}$ is a complete lattice, this least fixed point exists by the Knaster-Tarski theorem.

The case of initial transitions ($\Delta_{\mathtt{i}}$) is slightly different. The purpose of initial transitions is to compute an initial configuration $\mathbf{c}_0 \in \overline{\mathbb{D}}^Q$, given the initial vector $\mathbf{x} \in \overline{\mathbb{D}}^I$. There is no previous configuration, and no current data value, which is why we required $X \subseteq Q'$ for initial transitions and cur was not allowed. We define the function $\Delta_{\mathtt{i}} : \overline{\mathbb{D}}^I \to \overline{\mathbb{D}}^Q$ with the same fixed point computation from Equation (1), except that the initial states are additionally assigned values given by the vector $\mathbf{x}$. Define that $\mathbf{x}(q) = \perp$ if $q \notin I$. Then define $\Delta_{\mathtt{i}}(\mathbf{x}) = \mathbf{c}'$, where $\mathbf{c}'$ is the *least vector* satisfying, for all $q \in Q$, $\mathbf{c}'(q') = \mathbf{x}(q) \sqcup \bigsqcup_{(\mathtt{i}, X, q', t) \in \Delta} [\![t]\!](\mathbf{c}'|_X)$.

Now $\mathcal{A}$ is evaluated on input $(\mathbf{x}, \mathbf{w}) \in \overline{\mathbb{D}}^I \times (\Sigma \times \mathbb{D})^*$ by starting from the initial configuration and applying the update functions in sequence as illustrated in Figure 24. Finally, the output $\mathbf{y} \in \overline{\mathbb{D}}^F$ is given by $\mathbf{y} = \mathbf{c}|_F$, the projection of $\mathbf{c}$ to the final states.

Evaluation Complexity. Evaluation complexity of a data transducer depends on the underlying operations, so we give a conditional result where the complexity is stated in terms of the number of data registers and number of operations on those data registers.

Theorem 5.1. Evaluation of a data transducer $\mathcal{A}$, with number of states n and size m on input $(\mathbf{x}, \mathbf{w})$, requires $O(n)$ data registers to store the state, and $O(m)$ operations and additional data registers to process each element in $\Sigma \times \mathbb{D}$, independent of $\mathbf{w}$. The evaluation algorithm is given in Figure 25.

Regularity Data transducers define *regular transductions* on data words (see §5.4.1). Here, we show regularity in a simpler sense: whether an output value is defined (or undefined, or conflict) depends only on whether the input values are undefined, defined, or conflict,

```
$\mathbf{c} \leftarrow \Delta_{\mathtt{i}}(\mathbf{x})$; produce output $\mathbf{y} = \mathbf{c}|_F$
for each character $(\sigma, d)$ in $\mathbf{w}$ do
    for each state $q \in Q$ do $val(q) \leftarrow \mathbf{c}(q)$; $val(q') \leftarrow \bot$
    for each transition $\tau \in \Delta_\sigma$ do $val(\tau) \leftarrow \bot$; $num\_undef(\tau) \leftarrow |X|$
    $worklist \leftarrow Q' \cup \Delta_\sigma$
    while $worklist$ is nonempty, get $item$ from $worklist$ and do
        if $item$ is a transition $\tau = (\sigma, X, q', t) \in \Delta_\sigma$: then
            $val(\tau) \leftarrow [\![t]\!](val|_X)$
            if $val(q') \neq \top$ then add $q'$ to $worklist$
        else if $item$ is a state $q' \in Q'$ then
            if $val(q') = \bot$ then
                for each $\tau \in \Delta_\sigma$ with source variable $q'$ do $num\_undef(\tau) \leftarrow num\_undef(\tau) - 1$
            $val(q') \leftarrow \bigsqcup_{\tau=(\sigma,X,q',t)} val(\tau)$
            for each $\tau \in \Delta_\sigma$ with target variable $q'$ do
                if $val(\tau) \in \mathbb{D}$ or $(val(\tau) = \bot$ and $num\_undef(\tau) = 0)$ then add $\tau$ to $worklist$
    for each $q \in Q$ do $\mathbf{c}(q) \leftarrow val(q')$
    produce output $\mathbf{y} = \mathbf{c}|_F$
```

Figure 25: Data transducer evaluation algorithm (Theorem 5.1). On input $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ over $(\mathbb{D}, \mathsf{Op})$, an initial vector $\mathbf{x} \in \overline{\mathbb{D}}^I$, and a data stream $\mathbf{w} \in (\Sigma \times \mathbb{D})^*$, produces the output vector $\mathbf{y} \in \overline{\mathbb{D}}^F$ on each prefix of $\mathbf{w}$.

together with some regular property of the string of tags. For data vectors $\mathbf{x}_1, \mathbf{x}_2 \in \overline{\mathbb{D}}^X$, we say that $\mathbf{x}_1$ and $\mathbf{x}_2$ are *equivalent*, and write $\mathbf{x}_1 \equiv \mathbf{x}_2$, if for all $x \in X$, $\mathbf{x}_1(x)$ and $\mathbf{x}_2(x)$ are both undefined, both defined, or both conflict.

Theorem 5.2. Let $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ be a DT over $(\mathbb{D}, \mathsf{Op})$. Then: (i) For all initial vectors $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{D}^I$, and for all input words $\mathbf{w}_1, \mathbf{w}_2$, if $\mathbf{x}_1 \equiv \mathbf{x}_2$ and $\mathbf{w}_1 \downarrow \Sigma = \mathbf{w}_2 \downarrow \Sigma$, then $[\![\mathcal{A}]\!](\mathbf{x}_1, \mathbf{w}_1) \equiv [\![\mathcal{A}]\!](\mathbf{x}_2, \mathbf{w}_2)$. (ii) For every equivalence class of initial vectors $\mathbf{x}$ and equivalence class of output vectors $\mathbf{y}$, the set of strings $\mathbf{w} \downarrow \Sigma$ such that $[\![\mathcal{A}]\!](\mathbf{x}, \mathbf{w}) \equiv \mathbf{y}$ is regular.

Proof. In evaluating a DT we may collapse all values in $\mathbb{D}$ to a single value $\star$, so each state takes values in $\{\bot, \star, \top\}$. This gives a projection from $\mathcal{A}$ to a DT $\mathcal{P}$ over the ***unit signature*** $(\mathbb{U}, \mathsf{UOp})$, where $\mathbb{U} = \{\star\}$ is a set with just one element, and UOp consists of, for each k, the unique map $o_k : \mathbb{U}^k \to \mathbb{U}$. The projection homomorphically preserves the semantics. Then, (i) follows because the computation of $\mathcal{P}$ is exactly the same on $\mathbf{x}_1, \mathbf{w}_1$ and $\mathbf{x}_2, \mathbf{w}_2$, and (ii) follows because $\mathcal{P}$ has finitely many possible configurations. □

We can thus define the ***language*** of $\mathcal{A}$ to be $L(\mathcal{A}) = \{\mathbf{w} \downarrow \Sigma \mid [\![\mathcal{A}]\!](\mathbf{x}, \mathbf{w}) \in \mathbb{D}^F$ for some $\mathbf{x} \in \mathbb{D}^I\}$, so $L(\mathcal{A}) \subseteq \Sigma^*$. This is the set of tag strings $\mathbf{w} \downarrow \Sigma$ such that, if the initial vector of values is all defined, after reading in $\mathbf{w}$ all final states are defined. We similarly define the set of strings on which a DT is *defined or conflict*, on input of the same form: the *extended language* $\overline{L}(\mathcal{A})$ is $\{\mathbf{w} \downarrow \Sigma \mid [\![\mathcal{A}]\!](\mathbf{x}, \mathbf{w}) \in (\mathbb{D} \cup \{\top\})^F$ for some $\mathbf{x} \in (\mathbb{D} \cup \{\top\})^I\}$. An immediate corollary of Theorem 5.2 is that (i) $L(\mathcal{A})$ is regular, (ii) $\overline{L}(\mathcal{A})$ is regular, and (iii) $L(\mathcal{A}) \subseteq \overline{L}(\mathcal{A})$. Finally, say that DTs $\mathcal{A}_1$ and $\mathcal{A}_2$ are *equivalent* if for all $\mathbf{x}_1 \equiv \mathbf{x}_2$ and for all $\mathbf{w}$, $[\![\mathcal{A}_1]\!](\mathbf{x}_1, \mathbf{w}) \equiv [\![\mathcal{A}_2]\!](\mathbf{x}_2, \mathbf{w})$.

Theorem 5.3. On input DTs $\mathcal{A}_1$, $\mathcal{A}_2$, deciding if $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent is PSPACE-complete.

Proof. We first decide if the two are *not* equivalent in NPSPACE. It suffices to project $\mathcal{A}_1$ and $\mathcal{A}_2$ to DTs over the unit signature, $\mathcal{P}_1$ and $\mathcal{P}_2$, as in the previous proof, and decide if

$\mathcal{P}_1 \not\equiv \mathcal{P}_2$. Let n be the number of states between $\mathcal{P}_1$ and $\mathcal{P}_2$, and let m be their combined size. The number of configurations for $\mathcal{P}_1$ and $\mathcal{P}_2$ together is 3^n. Therefore, if there is a counterexample, it is some string over Σ of length at most 3^n. Guessing the counterexample one character at a time requires linear in n space to record the count and $O(m)$ space to update $\mathcal{P}_1$ and $\mathcal{P}_2$ (by Theorem 5.1).

To show it is PSPACE-hard, it suffices to exhibit a translation from NFAs to DTs which reduces language equality of NFAs to equivalence of DTs. Specifically, we create $\mathcal{A}$ with one final state which is undefined on strings for which the NFA is undefined, and $\top$ on strings for which the NFA is defined. The translation works by directly copying the states and transitions of the NFA, except we add *two* additional transitions from accepting states of the NFA to the new final state of $\mathcal{A}$. □

5.2 Constructions

Our primary interest in the DT model is to support a variety of succinct *composition operations* which are not simultaneously supported by any existing model. In particular, such composition operations can enable a quantitative monitoring language like QRE-Past in §5.3: language constructs can be implemented by the compiler as constructions on DTs, rather like how (traditional) regular expressions are compiled to nondeterministic finite automata.

For example, suppose we have DTs implementing two functions $f, g : (\Sigma \times \mathbb{D})^* \to \overline{\mathbb{D}}$, and we would like to implement the function $f + g$, which applies f and g to the input stream and adds the results. To do so, we copy the states of the transducers for f and g, and we initialize and update the states in parallel (they do not interfere). Then, we provide a new final state, and a single new transition which says that the new final state should be assigned the value of the final state of f plus the value of the final state of g. This works for every operation, and not just $+$: the combination of k computations by applying a k-ary operation $op \in \mathsf{Op}_k$ can be implemented by a corresponding k-ary construct on the k underlying DTs. Moreover, the size of the DT will only be the sum of the sizes of the k DTs, plus a constant. In contrast, even this simple operation $f + g$ is not succinctly implementable using the most natural existing alternative to DTs, Cost Register Automata (see §5.4).

This construction for $f + g$ requires no assumptions about the DTs implementing f and g. However, not all operations are this straightforward. Consider the following quantitative generalization of concatenation. Given $f : (\Sigma \times \mathbb{D})^* \to \overline{\mathbb{D}}$, $g : (\Sigma \times \mathbb{D})^* \to \overline{\mathbb{D}}$, and $op \in \mathsf{Op}_2$, we wish to implement $\texttt{split}(f, g, op)$: on input $\mathbf{w}$, split the input stream into two parts, $\mathbf{w} = \mathbf{u} \cdot \mathbf{v}$, such that $f(\mathbf{u}) \neq \perp$ and $g(\mathbf{v}) \neq \perp$ (respectively, f matches $\mathbf{u}$ and g matches $\mathbf{v}$), and return $op(f(\mathbf{u}), g(\mathbf{v}))$. Assume that the decomposition of $\mathbf{w}$ into $\mathbf{u}$ and $\mathbf{v}$ such that $f(\mathbf{u}) \neq \perp$ and $g(\mathbf{v}) \neq \perp$ is unique. In order to naively implement this operation, on an input string $\mathbf{w}$, we must not only keep track of the current configuration of f on $\mathbf{w}$, but for *every* split $\mathbf{w} = \mathbf{uv}$ where f matches $\mathbf{u}$, we must keep track of the current configuration of g on $\mathbf{v}$. If there are many possible prefixes $\mathbf{u}$ of $\mathbf{w}$ such that $f(\mathbf{u}) \neq \perp$, we may have to keep arbitrarily many configurations of g. This naive approach is therefore impossible using only the finite space that a DT allows, if we treat f and g only as black boxes.

What we need to avoid this is an additional structural condition on g. Rather than keeping multiple copies of g, we would like to keep only a single configuration in memory: whenever the current prefix matches f, *restart* g with new data values on its initial states (keeping any current data values as well). To motivate this idea, consider the analogous concatenation construction for two NFAs: every time the first NFA accepts, we are able to "restart" the second NFA by adding a token to its start state (we don't need an entirely new NFA every time). This property for DTs is called *restartability.* Restartable DTs are an equally expressive subclass consisting of those DTs for which restarting computation on the same transducer does not cause interference in the output.

The set of strings that a DT "matches" is captured by its *extended language*, defined in §5.1. Correspondingly, we assume that whenever a DT is restarted, the new initial vector is either all $\bot$, or all *not* $\bot$ (in $\mathbb{D} \cup \{\top\}$). If the *output* of a DT also satisfies this property (on every input it is either all $\bot$, or all *not* $\bot$), then we say that the DT is *output-synchronized.* This property is required in the concatenation and iteration constructions, but it is not as crucial to the discussion as restartability.

We begin in §5.2.1 by giving general constructions that do not rely on restartability. We highlight the implemented semantics, the extended language, and the size of the constructed DT in terms of its constituent DTs. Then in §5.2.2, we define restartability and use it to give succinct constructions for unambiguous parsing operations, namely *concatenation* and *iteration.* Moreover, we show that (under certain conditions) our operations *preserve* restartability, thus enabling modular composition using the restartable DTs. We also show that checking restartability is hard (PSPACE-complete), and we mention converting a non-restartable DT to a restartable one, but with exponential blowup.

5.2.1 General Constructions

Notation It is convenient to introduce shorthand (ε, X, q', t) for the union of $|\Sigma| + 1$ transitions: (σ, X, q', t) for every $\sigma \in \Sigma \cup \{\mathtt{i}\}$. Because this includes an initial transition, this requires that $X \subseteq Q'$ and that $\mathtt{cur}$ does not appear in t. We call such a collection of transitions an *epsilon transition* because, like epsilon transitions from classical automata, the transition may produce a value at its target state on the empty data word and on every input character.

For readability, we abbreviate the type of a DT $\mathcal{A} : \overline{\mathbb{D}}^I \times (\Sigma \times D)^* \to \overline{\mathbb{D}}^F$ as $\mathcal{A} : I \twoheadrightarrow F$. This can be thought of as a function from input variables I of type $\overline{\mathbb{D}}$ to output variables F of type $\overline{\mathbb{D}}$, which also consumes some data word in $(\Sigma \times D)^*$ as a side effect. For sets of variables (or states) X_1, X_2, when we write $X_1 \cup X_2$ we assume that the union is disjoint, unless otherwise stated.

We also define a *data function* to be a plain function $\overline{\mathbb{D}}^I \to \overline{\mathbb{D}}^F$ which is given by a collection of one or more terms $t : \mathsf{Tm}[I]$ for each $f \in F$ (the output value of f is then the union of the values of all terms). If $G \subseteq F \times \mathsf{Tm}[I]$, then we write $G : I \Rightarrow F$ to abbreviate the semantics $[\![G]\!] : \overline{\mathbb{D}}^I \to \overline{\mathbb{D}}^F$. The ***size*** of G is the total length of description of all of the terms t it contains.

Parallel composition. Suppose we are given DTs $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \Delta_2, I_2, F_2)$, and assume that the sets of initial states are the same up to some implicit bijections $\pi_1 : I \to I_1$, $\pi_2 : I \to I_2$, for a set I with $|I| = |I_1| = |I_2|$. (It is always possible to benignly extend both DTs with extra initial states so that they match, so this assumption is not restrictive.) We wish to define a DT which feeds the input $(\mathbf{x}, \mathbf{w})$ into both DTs in parallel. To do so, we define $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$ to be the tuple $(Q, \Sigma, \Delta, I, F)$, where $Q = Q_1 \cup Q_2 \cup I$, $F = F_1 \cup F_2$, and

$$\Delta = \Delta_1 \cup \Delta_2 \cup \big\{(\varepsilon, i', \pi_1(i)', i') : i \in I\big\} \cup \big\{(\varepsilon, i', \pi_2(i)', i') : i \in I\big\}.$$

Here, the transitions we added (those in Δ but not in Δ_1 or Δ_2) *copy* values from I into both I_1 and I_2. This is only relevant on initialization $\Delta_{\mathtt{i}}$, since after that states I will not be defined, but we used an epsilon transition instead of just an $\mathtt{i}$ transition to preserve restartability, which will be discussed in §5.2.2. Since we added no other transitions, the least fixed point Equation (1) defining the next (or initial) configuration decomposes into the least fixed point on states Q_1, and on states Q_2. It follows that the semantics satisfies $[\![\mathcal{A}]\!](\mathbf{x}, \mathbf{u}) = ([\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{u}), [\![\mathcal{A}_2]\!](\mathbf{x}, \mathbf{u}))$. Here, $(\mathbf{y}_1, \mathbf{y}_2)$ denotes the vector $\mathbf{y} \in \overline{\mathbb{D}}^F$ that is $\mathbf{y}_1$ on F_1 and $\mathbf{y}_2$ on F_2. Parallel composition is commutative and associative. The utility of parallel composition is that it allows us to combine the outputs $\mathbf{y}_1$ and $\mathbf{y}_2$ later on. This is accomplished by *concatenation* with another DT which combines the outputs (§5.2.2).

Parallel composition. If $\mathcal{A}_1 : I \twoheadrightarrow F_1$ and $\mathcal{A}_2 : I \twoheadrightarrow F_2$, then $\mathcal{A}_1 \parallel \mathcal{A}_2 : I \twoheadrightarrow F_1 \cup F_2$ satisfies

$$[\![\mathcal{A}_1 \parallel \mathcal{A}_2]\!](\mathbf{x}, \mathbf{w}) = ([\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w}), [\![\mathcal{A}_2]\!](\mathbf{x}, \mathbf{w})),$$

such that $\mathsf{size}(\mathcal{A}_1 \parallel \mathcal{A}_2) = \mathsf{size}(\mathcal{A}_1) + \mathsf{size}(\mathcal{A}_2) + O(|I|)$. It therefore matches the set of tag strings $\overline{L}(\mathcal{A}_1 \parallel \mathcal{A}_2) = \overline{L}(\mathcal{A}_1) \cap \overline{L}(\mathcal{A}_2)$.

Union. Suppose we are given DTs $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \Delta_2, I_2, F_2)$, and assume that the sets of initial and final states are the same up to some bijections: $\pi_1 : I \to I_1$, $\pi_2 : I \to I_2$, $\rho_1 : F \to F_1$, $\rho_2 : F \to F_2$, for sets I and F with $|I| = |I_1| = |I_2|$ and $|F| = |F_1| = |F_2|$. We wish to define a DT which feeds the input $(\mathbf{x}, \mathbf{w})$ into both DTs in parallel and returns the union ($\sqcup$) of the two results. We define $\mathcal{A} = \mathcal{A}_1 \sqcup \mathcal{A}_2 = (Q, \Sigma, \Delta, I, F)$ by $Q = Q_1 \cup Q_2 \cup I \cup F$ and

$$\begin{aligned}\Delta = \Delta_1 \cup \Delta_2 \cup \big\{(\varepsilon, i', \pi_1(i)', i') : i \in I\big\} \qquad &\cup \big\{(\varepsilon, i', \pi_2(i)', i') : i \in I\big\} \\ \cup \big\{(\varepsilon, \rho_1(f)', f', \rho_1(f)') : f \in F\big\} &\cup \big\{(\varepsilon, \rho_2(f)', f', \rho_2(f)') : f \in F\big\}.\end{aligned}$$

Similar to the parallel composition construction, the additional transitions here ensure that we copy values from I into I_1 and I_2, and copy values from F_1 and F_2 into F, whenever these values are defined. In particular, on initialization the initial vector $\mathbf{x}$ will be copied into I_1 and I_2, and on every data word the output values $\mathbf{y}_1$ and $\mathbf{y}_2$ of $\mathcal{A}_1$ and $\mathcal{A}_2$ will be copied into the *same* set of final states, so that they have to be joined by $\sqcup$. In particular, if both $\mathbf{y}_1$ and $\mathbf{y}_2$ are defined, the output will be $\top$. We see therefore that the semantics is such that $[\![\mathcal{A}]\!](\mathbf{x}, \mathbf{u}) = [\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{u}) \sqcup [\![\mathcal{A}_2]\!](\mathbf{x}, \mathbf{u})$. Like parallel composition, union is commutative and associative.

Union. If $\mathcal{A}_1 : I \twoheadrightarrow F$ and $\mathcal{A}_2 : I \twoheadrightarrow F$, then $\mathcal{A}_1 \sqcup \mathcal{A}_2 : I \twoheadrightarrow F$ implements the semantics

$$[\![\mathcal{A}_1 \sqcup \mathcal{A}_2]\!](\mathbf{x}, \mathbf{w}) = [\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w}) \sqcup [\![\mathcal{A}_2]\!](\mathbf{x}, \mathbf{w}),$$

s.t. $\mathsf{size}(\mathcal{A}_1 \sqcup \mathcal{A}_2) = \mathsf{size}(\mathcal{A}_1) + \mathsf{size}(\mathcal{A}_2) + O(|I| + |F|)$. It matches $\overline{L}(\mathcal{A}_1 \sqcup \mathcal{A}_2) = \overline{L}(\mathcal{A}_1) \cup \overline{L}(\mathcal{A}_2)$.

Prefix summation. Now we consider a more complex operation. Suppose we are given $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$, and a data word $\mathbf{w}$, such that the output on the empty data word is $\mathbf{y}_1^{(0)}$, the output after receiving one character of the data word is $\mathbf{y}_1^{(1)}$, and in general the output after k characters is $\mathbf{y}_1^{(k)}$. The problem is to return the *sum* of these outputs: we want a DT that returns $\mathbf{y}^{(i)} = \mathbf{y}_1^{(0)} + \cdots + \mathbf{y}_1^{(i)}$ after receiving i characters. This is called the *prefix sum* because $\mathbf{y}_1^{(k)}$ is the value of $\mathcal{A}$ on the kth prefix of the data word. In general, instead of $+$, we can take an arbitrary operation which folds the outputs of $\mathcal{A}_1$ on each prefix. We suppose that this operation is given by a data function G which, for some set F, is a function $\overline{\mathbb{D}}^{F \cup F_1} \to \overline{\mathbb{D}}^{F}$. It takes the previous "sum" $\mathbf{y}^{(i-1)} \in \overline{\mathbb{D}}^{F}$, combines it with the new output of $\mathcal{A}_1$, $\mathbf{y}_1^{(i)} \in \overline{\mathbb{D}}^{F_1}$, and produces the next "sum" $\mathbf{y}^{(i)} \in \overline{\mathbb{D}}^{F}$. So, we'll have $G(\mathbf{y}^{(i-1)}, \mathbf{y}_1^{(i)}) = \mathbf{y}^{(i)}$. We want a DT that, on input initial values for I_1 and initial values $\mathbf{y}^{(-1)}$ for F, will return $\mathbf{y}^{(i)}$. Formally, we convert G to a DT $\mathcal{A}_2 = (Q_2, \Sigma, \Delta_2, I_2, F_2)$, with bijections $\pi : (F \cup F_1) \to I_2$, $\rho : F \to F_2$, which only contains epsilon-transitions: for each term t in G with variables $P \subseteq (F \cup F_1)$ giving a value of $f \in F$, we create an epsilon transition $(\varepsilon, \pi(P)', \rho(f)', t)$. Then we define the prefix sum $\oplus_G \mathcal{A}_1 = (Q, \Sigma, \Delta, (I_1 \cup F), F_2)$, where $Q = Q_1 \cup Q_2 \cup F$ and

$$\begin{aligned}\Delta = \Delta_1 \cup \Delta_2 \cup \big\{(\varepsilon, f_1', \pi(f_1)', f_1') : f_1 \in F_1\big\} \\ \cup \big\{(\varepsilon, f', \pi(f)', f') : f \in F\big\} \quad \cup \big\{(\sigma, \rho(f), \pi(f)', \rho(f)) : f \in F, \sigma \in \Sigma\big\}.\end{aligned}$$

First on the empty data word, the outputs F_2' of $\mathcal{A}_1$ and the initial vector in F' are copied into I_2, and $\mathcal{A}_2$ produces the correct output $\mathbf{y}^{(0)} = [\![G]\!](\mathbf{y}^{(-1)}, \mathbf{y}_1^{(0)})$. Now, when we read in a character in $\Sigma \times \mathbb{D}$, the final states F_2' flow back into inputs to $\mathcal{A}_2$, and the new output of $\mathcal{A}_1$ also flows in. Because the machine $\mathcal{A}_2$ was constructed to be just a set of epsilon-transitions from I_2 to F_2, it does not save any internal state, but just computes the output in terms of the input again. So the next output will be $[\![G]\!](\mathbf{y}^{(0)}, \mathbf{y}_1^{(1)})$, and then $[\![G]\!](\mathbf{y}^{(1)}, \mathbf{y}_1^{(2)})$, and so forth.

Prefix sum. If $\mathcal{A}_1 : I \twoheadrightarrow Z$ and $G : F \cup Z \Rightarrow F$, then $\oplus_G \mathcal{A}_1 : I \cup F \twoheadrightarrow F$ implements the semantics

$$[\![\oplus_G \mathcal{A}_1]\!]((\mathbf{x}, \mathbf{y}), \varepsilon) = [\![G]\!](\mathbf{y}, [\![\mathcal{A}_1]\!](\mathbf{x}, \varepsilon))$$
$$[\![\oplus_G \mathcal{A}_1]\!]((\mathbf{x}, \mathbf{y}), \mathbf{w}(\sigma, d)) = [\![G]\!]([\![\oplus_G \mathcal{A}_1]\!]((\mathbf{x}, \mathbf{y}), \mathbf{w}), [\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w}(\sigma, d)))$$

such that $\mathsf{size}(\oplus_G \mathcal{A}_1) = \mathsf{size}(\mathcal{A}_1) + \mathsf{size}(G) + O(|Z| + |F|)$.

Conditioning on undefined and conflict values. A DT that is constructed using the other operations—particularly union, and concatenation and iteration from §5.2.2—may produce undefined ($\bot$) or conflict ($\top$) on certain inputs. In such a case, we may want to perform a computation which *conditions* on whether the output is undefined, defined or conflict: for instance, we may want to produce 1 if there is a conflict, or we may want to replace all $\bot$ and $\top$ outputs with concrete data values. (In particular, in §5.3, we will want to replace $\bot$ and $\top$ with Boolean values.) We give a construction for this purpose. To simplify the problem, suppose that we are given $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$, and we want to construct a DT $\mathcal{A}_\bot$ with no initial states, the same set of final states, and the following behavior: for all $\mathbf{x} \in \mathbb{D}^{I_1}$ (*not* $\overline{\mathbb{D}}^{I_1}$), all $\mathbf{u} \in (\Sigma \times \mathbb{D})^*$, and all $f_1 \in F_1$, if $[\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{u})(f_1) = \bot$ then $[\![\mathcal{A}_\bot]\!](\mathbf{u})(f_1) \in \mathbb{D}$, and otherwise, $[\![\mathcal{A}_\bot]\!](\mathbf{u})(f_1) = \bot$. Here, since $I = \varnothing$, the first argument is omitted. We similarly want to define $\mathcal{A}_{\mathbb{D}}$ which is in $\mathbb{D}$ if $\mathcal{A}_1$ is in $\mathbb{D}$, and $\bot$ otherwise, and $\mathcal{A}_\top$ which is in $\mathbb{D}$ if $\mathcal{A}_1$ is $\top$, and $\bot$ otherwise. So that $\mathbb{D}$ is not empty, we assume that there is some constant operation in Op_0, say $d_\star$ (so $d_\star \in \mathbb{D}$).

The idea of the construction is that we replace Q_1 with $Q_1 \times \{\bot, \star, \top\}$. For each state $q \in Q_1$, at all times, exactly one of $(q, \bot)$, $(q, \star)$, and $(q, \top)$ will be $d_\star$ and the other two will be $\bot$. Which state is $d_\star$ should correspond to whether q was undefined, defined, or conflict. (This is adapted from the classic trick of dealing with negation by replacing all values with pairs of either (true, false) or (false, true).) However, in order for this to work without blowup our DT needs to be *acyclic*. Therefore we begin with a preliminary stage of converting the DT to acyclic. Observe that in the semantics of §5.1, iterating the assignment (1) $2n$ times would be sufficient to reach the fixed point, where n is the number of states of the DT. So we create $2n$ copies of the states of the DT, with one set of transitions from each copy to the next. In this preliminary stage the size of the transducer may be *squared*, i.e. there is quadratic blowup. Now assuming $\mathcal{A}$ is acyclic, for each variable $q' \in Q_1'$, whether q' is undefined, defined, or conflict is a Boolean function of all the source states of transitions that target q'; this function can be built as a Boolean circuit by adding intermediate states and intermediate transitions, in number at most the total size of the transitions targeting q'. $\mathcal{A}_\bot$, $\mathcal{A}_{\mathbb{D}}$, and $\mathcal{A}_\top$ differ only in which states are final—$F_1 \times \{\bot\}$, $F_1 \times \{\star\}$, and $F_1 \times \{\top\}$, respectively.

Support. Let $d_\star \in \mathbb{D}$. If $\mathcal{A}_1 : I \twoheadrightarrow F$, then $[\mathcal{A}_1 = \bot] : \varnothing \twoheadrightarrow F$, $[\mathcal{A}_1 \in \mathbb{D}] : \varnothing \twoheadrightarrow F$, and $[\mathcal{A}_1 = \top] : \varnothing \twoheadrightarrow F$. These constructions implement the following semantics. For all $f \in F$:

$$\begin{aligned}
[\![[\mathcal{A}_1 = \bot]]\!](\mathbf{w})(f) &= d_\star \text{ if } [\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w})(f) = \bot \ \ \forall \mathbf{x} \in \mathbb{D}^I; \quad \bot \text{ otherwise}\\
[\![[\mathcal{A}_1 \in \mathbb{D}]]\!](\mathbf{w})(f) &= d_\star \text{ if } [\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w})(f) \in \mathbb{D} \ \ \forall \mathbf{x} \in \mathbb{D}^I; \quad \bot \text{ otherwise}\\
[\![[\mathcal{A}_1 = \top]]\!](\mathbf{w})(f) &= d_\star \text{ if } [\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w})(f) = \top \ \ \forall \mathbf{x} \in \mathbb{D}^I; \quad \bot \text{ otherwise}
\end{aligned}$$

such that $\mathsf{size}([\mathcal{A}_1 = \bot]) = O(\mathsf{size}(\mathcal{A}_1)^2)$ and likewise for the other two. Alternatively, if $\mathcal{A}_1$ is acyclic, the size will only be $O(\mathsf{size}(\mathcal{A}_1))$.

5.2.2 Unambiguous Parsing and Restartability

We now want to capture the idea of restartability—that multiple threads of computation may be replaced by updates to a single configuration—with a formal definition. Recall the example in the introduction of $\texttt{split}(f, g, op)$. During the execution of f on input $\mathbf{w}$, whenever the current prefix $\mathbf{u}$ of $\mathbf{w}$ matches, i.e. $f(\mathbf{u}) \neq \bot$, we could (naively and inefficiently) implement $\texttt{split}(f, g, op)$ by keeping a separate configuration (thread) of g from that point forward. For example, suppose that $\mathbf{w} = (\mathsf{a}, d_1)(\mathsf{b}, d_2)(\mathsf{a}, d_3)(\mathsf{a}, d_4)$, and that the output of f is defined after receiving each a-item, and undefined otherwise. Then f is defined on input (a, d_1), on $(\mathsf{a}, d_1)(\mathsf{b}, d_2)(\mathsf{a}, d_3)$, and on $(\mathsf{a}, d_1)(\mathsf{b}, d_2)(\mathsf{a}, d_3)(\mathsf{a}, d_4)$. Corresponding to these three inputs, we would have three threads of g: $\mathbf{c}_1$ on input $(\mathsf{b}, d_2)(\mathsf{a}, d_3)(\mathsf{a}, d_4)$, $\mathbf{c}_2$ on input (a, d_4), and $\mathbf{c}_3$ on input ε. Suppose that each configuration $\mathbf{c}_i$ includes an final state with the value of $\mathbf{y}_i = op(f(\mathbf{u}), g(\mathbf{v}))$. The value of $\texttt{split}(f, g, op)$ could then be computed as the *union* of the outputs from all these threads: $\texttt{split}(f, g, op)(\mathbf{w}) = \mathbf{y}_1 \sqcup \mathbf{y}_2 \sqcup \mathbf{y}_3$. We apply the union here because we expect the split $\mathbf{w} = \mathbf{u} \cdot \mathbf{v}$, where $\mathbf{u} \in \overline{L}(f)$ and $\mathbf{v} \in \overline{L}(g)$, to be unique. Thus all but at most one of $\mathbf{y}_i$ will be $\bot$, and the union gives us the unique answer (if any).

A DT will be called restartable if a *single configuration* $\mathbf{c}$ can *simulate* the behavior of these several configurations $\mathbf{c}_1, \mathbf{c}_2$, and $\mathbf{c}_3$. This is a relation between configurations of g and an arbitrarily long sequence of configurations of g (we could have used a multiset instead of a sequence). The relation $\mathbf{c} \sim [\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3]$ is intended to capture that $\mathbf{c}$ is observationally indistinguishable from the sequence $\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$. For starters, we require that the output is the same: if $\mathbf{y}$ is the output of $\mathbf{c}$, then $\mathbf{y} = \mathbf{y}_1 \sqcup \mathbf{y}_2 \sqcup \mathbf{y}_3$. But we also require that the simulation is preserved when we update the sequence of configurations of g, by reading in a new input character and/or starting a new thread. The definition allows the simulation to be undefined on configurations that are never reachable in an actual execution—it need not be true that *every* sequence $[\mathbf{c}_1, \ldots, \mathbf{c}_k]$ is simulated by some $\mathbf{c}$, but it should be true that every sequence that can be reached by a series of updates is simulated.

With this intuition, the simulation relation on configurations of g should satisfy the following properties (see the definition below). Property (i) addresses the base case before any input characters are received (i.e. initialization $\mathtt{i}$). Suppose that on initialization, the machine for g is started with $k \geq 0$ threads, given by initial vectors $\mathbf{x}_1, \ldots, \mathbf{x}_k$. (In our example, these threads would arise as the output of f on initialization.) Then the configuration in a single copy of g on input $\mathbf{x}_1 \sqcup \cdots \sqcup \mathbf{x}_k$ should simulate the behavior of k separate copies of g. Property (ii) requires that the simulation then be preserved as input characters are read in. Suppose that $\mathbf{c} \sim [\mathbf{c}_1, \ldots, \mathbf{c}_k]$, and we now read in a character (σ, d) to g. Simultaneously, we start zero or more new threads represented by the vector $\mathbf{x}$ (e.g., $\mathbf{x}$ is the new output produced by f on input (σ, d)). Then if we update and re-initialize the initial states of $\mathbf{c}$ with $\mathbf{x}$, that configuration should simulate updating each $\mathbf{c}_i$ separately, *and* adding one or more new threads represented by $\mathbf{x}$. Finally, property (iii) says that our simulation is sound: for every configuration which simulates a sequence of configurations, the output of the one configuration is equal to the union of the sequence of outputs.

For property (ii) in particular, we need to define what it means to update a configuration $\mathbf{c}$ and simultaneously restart new threads by placing values $\mathbf{x}$ on the initial states I'. (Such an update function is only needed for the simulating configuration, not the sequence of simulated configurations.) For each $\sigma \in \Sigma$ and for every $\mathbf{x} \in \overline{\mathbb{D}}^I$ we define a generalized evaluation function $\Delta_{\sigma,\mathbf{x}} : \overline{\mathbb{D}}^Q \times \mathbb{D} \to \overline{\mathbb{D}}^Q$. This represents executing Δ_σ and then starting *zero or more* new threads, by initializing the new initial states with $\mathbf{x}$. We modify the least fixed point definition of $\mathbf{c}'$ in Equation 1) to include the new initialization on states I': $\mathbf{c}'$ is the least vector satisfying

$$\mathbf{c}'(q') = \mathbf{x}(q) \sqcup \bigsqcup_{(\sigma, X, q', t) \in \Delta} [\![t]\!](\mathbf{c}'|_X),$$

where $\mathbf{x}(q) = \bot$ if $q \notin I$. This resembles the way we already incorporated $\mathbf{x}$ into the definition of $\Delta_{\mathtt{i}}$. We restrict the vector $\mathbf{x}$ in each restart to be in the space $\mathcal{X} = \{\bot\}^I \cup (\mathbb{D} \cup \{\top\})^I$, which is closed under $\sqcup$. Let $\vec{\bot}$ be the vector with every entry equal to $\bot$.

Definition 5.4 (Restartability)**.** Let $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ be a DT over signature $(\mathbb{D}, \mathsf{Op})$; let $C = \overline{\mathbb{D}}^Q$ be the set of configurations of $\mathcal{A}$, and $[C]$ the set of *finite lists* of configurations of $\mathcal{A}$. Let $\mathcal{X} = \{\bot\}^I \cup (\mathbb{D} \cup \{\top\})^I$ be the set of possible initializations for a restarted thread. $\mathcal{A}$ is ***restartable*** if there exists a binary relation $\sim \subseteq C \times [C]$ (called a "simulation") with the following properties:

i. **(Base case)** For all $\mathbf{x}_1, \ldots, \mathbf{x}_k \in \mathcal{X}$, $\Delta_{\mathtt{i}}\left(\bigsqcup_{i=1}^k \mathbf{x}_i\right) \sim [\Delta_{\mathtt{i}}(\mathbf{x}_1), \ldots, \Delta_{\mathtt{i}}(\mathbf{x}_k)]$. (If $k = 0$, we get $\Delta_{\mathtt{i}}(\vec{\perp}) \sim []$, where $[] \in [C]$ denotes the empty list.)

ii. **(Update with restarts)** For all $(\sigma, d) \in (\Sigma \times \mathbb{D})$, for all $x \in \mathcal{X}$, and for all $\mathbf{c}$, $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k$, $\hat{\mathbf{c}}_1, \hat{\mathbf{c}}_2, \ldots, \hat{\mathbf{c}}_l$, if $\mathbf{c} \sim [\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k]$ and $\Delta_{\mathtt{i}}(\mathbf{x}) \sim [\hat{\mathbf{c}}_1, \hat{\mathbf{c}}_2, \ldots, \hat{\mathbf{c}}_l]$ then

$$\Delta_{\sigma,\mathbf{x}}(\mathbf{c}, d) \sim [\Delta_\sigma(\mathbf{c}_1, d), \ldots, \Delta_\sigma(\mathbf{c}_k, d), \hat{\mathbf{c}}_1, \hat{\mathbf{c}}_2, \ldots, \hat{\mathbf{c}}_l].$$

iii. **(Implies same output)** If $\mathbf{c} \sim [\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k]$, and the output vectors for these configurations (extended data values at the final states) are $\mathbf{y}, \mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_k$, respectively, then we have $\mathbf{y} = \mathbf{y}_1 \sqcup \mathbf{y}_2 \sqcup \cdots \sqcup \mathbf{y}_k$.

A simple example (and counterexample) are in order. First, consider the following DT $\mathcal{A}$ with two states: $Q = \{i, f\}$, $\Sigma = \{\mathsf{a}, \mathsf{b}\}$, $I = \{i\}$, $F = \{f\}$, and one transition on input a, $f' := i + \mathsf{cur}$. The DT on input $(x, (\mathsf{a}, d))$ returns $x + d$, and on every other input is undefined. Then $\mathcal{A}$ is restartable. We can represent configurations as ordered pairs (x, y), where $x \in \overline{\mathbb{D}}$ is the value of i and $y \in \overline{\mathbb{D}}$ is the value of f. We define that $\mathbf{c} \sim [\mathbf{c}_1, \ldots, \mathbf{c}_k]$ whenever $\mathbf{c} = \bigsqcup_{i=1}^k \mathbf{c}_i$. Then (i), (ii), and (iii) hold. For example, the base case says that $x = \bigsqcup_{i=1}^k x_k$, then $(x, \perp) \sim [(x_1, \perp), \ldots, (x_k, \perp)]$, which is true by definition. The intuition is that, in this simple case, we can say that a configuration of $\mathcal{A}$ simulates a set of configurations (threads) if the configuration is the union of all those threads. The semantics just takes (x, y) to (z, x) on updating and restarting with z, so it preserves this relation.

For a counterexample, consider a DT $\mathcal{A}$ which sums the value of a single initial state and the last a: take $Q = \{i, f\}$, $I = \{i\}$, $F = \{f\}$, and the following transitions on input a: $i' := i$, $f' := i' + \mathsf{cur}$. We may represent configurations as (x, y), for the values at i, f, respectively. To see this is not restartable, consider starting $\mathcal{A}$ with a single input $x_1 \in \mathbb{D}$, then reading in (a, d) and starting a second input $x_2 \in \mathbb{D}$ (i.e. applying Δ_{a,x_2}). Starting with x_1 results in the configuration $(x_1, \perp)$; then reading in (a, d) and starting with x_2 results in $(\top, \top)$. However, if $\mathcal{A}$ were restartable, then by property (ii), we could instead read in (a, d) and add the second input x_2 separately: we thus would have $(\top, \top) \sim [(x_1, x_1 + d), (x_2, \perp)]$. The problem is that this violates (iii): the output of $\mathcal{A}$ is $\top$, which is not the same as $(x_1 + d) \sqcup \perp = x_1 + d$.

What is relevant for properties (i), (ii), and (iii) is actually only the configurations, input, and output *up to equivalence*, i.e., where we replace $\overline{\mathbb{D}}$ with $\{\perp, \star, \top\}$. There are only finitely many configurations up to equivalence. This is why restartability is decidable (see Theorem 5.6).

Concatenation Suppose we are given two DTs $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \Delta_2, I_2, F_2)$, where F_1 and I_2 are the same up to bijection (say, $\pi : F_1 \to I_2$). Now we want to compute the following parsing operation: on input $(\mathbf{x}, \mathbf{w})$, consider all splits of $\mathbf{w}$ into two strings, $\mathbf{w} = \mathbf{w}_1\mathbf{w}_2$. Apply $\mathcal{A}_1$ to $(\mathbf{x}, \mathbf{w}_1)$ to get a result $\mathbf{y}_1$, and apply $\mathcal{A}_2$ to $(\mathbf{y}_1, \mathbf{w}_2)$ to get $\mathbf{y}_2$. Return the union ($\sqcup$) over all such splits of $\mathbf{y}_2$. In particular, assuming there is only one way to split $\mathbf{w} = \mathbf{w}_1\mathbf{w}_2$ such that $\mathbf{y}_2$ does not end up being undefined, this operation splits the input string uniquely into two parts such that $\mathcal{A}_1$ matches $\mathbf{w}_1$ and $\mathcal{A}_2$ matches $\mathbf{w}_2$, and then applies $\mathcal{A}_1$ and $\mathcal{A}_2$ in sequence.

We implement this by taking $\mathcal{A} = \mathcal{A}_1 \cdot \mathcal{A}_2 = (Q, \Sigma, \Delta, I, F)$ with $Q = Q_1 \cup Q_2$, $I = I_1$, $F = F_2$, and

$$\Delta = \Delta_1 \cup \Delta_2 \cup \left\{(\varepsilon, \{f_1'\}, \pi(f_1)', f_1') : f_1 \in F_1\right\}.$$

The idea is very simple; every output of $\mathcal{A}_1$ (i.e. a value produced at a state in F_1) should be copied into the corresponding initial state of $\mathcal{A}_2$. This happens on initialization, and on every update. However, the semantics is not so simple, because every time we read in a character, $\mathcal{A}_2$'s initial states I_2 are being re-initialized with new values (the values from F_1).

This "re-initialization" is exactly captured by our generalized update function $\Delta_{\sigma,\mathbf{x}}$ from earlier. Let us represent configurations of $\mathcal{A}$ by $(\mathbf{c}_1, \mathbf{c}_2)$, where $\mathbf{c}_i$ is the component restricted to Q_i, i.e. the induced configuration of $\mathcal{A}_i$. Now consider an input $(\mathbf{x}, \mathbf{w})$ to $\mathcal{A}$. We see that for the ith configuration of $\mathcal{A}$ $(\mathbf{c}_1^{(i)}, \mathbf{c}_2^{(i)})$, $\mathbf{c}_1^{(i)}$ is the same as the ith configuration of $\mathcal{A}_1$ on input $(\mathbf{x}, \mathbf{w})$. Moreover, if $\mathbf{y}_1^{(i)}$ is the ith output of $\mathcal{A}_1$, this is used to reinitialize $\mathcal{A}_2$; so we see that $\mathbf{c}_2^{(i)} = \Delta_{\sigma,\mathbf{y}_1^{(i)}}(\mathbf{c}_2^{(i-1)}, d)$ (where this is the update function of $\mathcal{A}_2$). The output $\mathbf{y}_2^{(i)} = \mathbf{c}_2^{(i)}|_F$ of $\mathcal{A}_2$ is the output of $\mathcal{A}$.

Assume that $\mathcal{A}_1$ is *output-synchronized*: this means that each $\mathbf{y}_1^{(i)} \in \mathcal{X}$, i.e., all values are $\perp$ or all values are in $\mathbb{D} \cup \{\top\}$. And assume that $\mathcal{A}_2$ is *restartable*. Then the simulation relation allows us to, at every step, replace $\mathbf{c}_2$ by a list of configurations where each configuration is $\mathcal{A}_2$ on a different suffix of $\mathbf{w}$. In particular, we recursively replace $\Delta_{\sigma,\mathbf{y}_1^{(i)}}(\mathbf{c}_2^{(i-1)}, d)$ with the list of configurations for $\Delta_\sigma(\mathbf{c}_2^{(i-1)}, d)$ and a single new thread $\Delta_{\mathtt{i}}(\mathbf{y}_1^{(i)})$. Because $\mathbf{y}_1^{(i)} \in \mathcal{X}$, this is guaranteed by property (ii) of restartability. Property (iii) then implies the semantics given in the following summary.

Concatenation. Let $\mathcal{A}_1 : I \twoheadrightarrow Z$ and $\mathcal{A}_2 : Z \twoheadrightarrow F$, such that $\mathcal{A}_1$ is output-synchronized and $\mathcal{A}_2$ is restartable. Then $\mathcal{A}_1 \cdot \mathcal{A}_2 : I \twoheadrightarrow F$ implements the semantics

$$[\![\mathcal{A}_1 \cdot \mathcal{A}_2]\!](\mathbf{x}, \mathbf{w}) = \bigsqcup_{\mathbf{w}=\mathbf{w}_1\mathbf{w}_2} [\![\mathcal{A}_2]\!]([\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w}_1), \mathbf{w}_2).$$

such that $\mathsf{size}(\mathcal{A}_1 \cdot \mathcal{A}_2) = \mathsf{size}(\mathcal{A}_1) + \mathsf{size}(\mathcal{A}_2) + O(|Z|)$. It matches $\overline{L}(\mathcal{A}_1 \cdot \mathcal{A}_2) = \overline{L}(\mathcal{A}_1) \cdot \overline{L}(\mathcal{A}_2)$.

Concatenation with data functions. A special case of concatenation can be described which does *not* require restartability, and which we use in §5.3. Suppose we are given $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$ and we want to concatenate with a data function $G_2 : F_1 \Rightarrow F_2$: on input $(\mathbf{x}, \mathbf{w})$, return $[\![G_2]\!]([\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w}))$. This can be implemented by converting G_2 into a DT $\mathcal{A}_2$ on states $F_1 \cup F_2$ (as in the prefix sum construction), and then simply constructing $\mathcal{A}_1 \cdot \mathcal{A}_2$. Even if $\mathcal{A}_2$ is not restartable, we can see directly that on every input, the final states F_2 are equal to G_2 applied to the output of $\mathcal{A}_1$. Similarly, if $G_1 : I_1 \Rightarrow I_2$ and $\mathcal{A}_2 : (Q_2, \Sigma, \Delta_2, I_2, F_2)$, then we may convert G_1 into a DT $\mathcal{A}_1$ on states $I_1 \cup I_2$. Then the construction $\mathcal{A}_1 \cdot \mathcal{A}_2$, on every input $(\mathbf{x}, \mathbf{w})$, returns $[\![\mathcal{A}_2]\!]([\![G_1]\!](\mathbf{x}), \mathbf{w})$. We overload the concatenation notation and write these constructions as $\mathcal{A}_1 \cdot G_2$ and $G_1 \cdot \mathcal{A}_2$. For these constructions, as with prefix sum, we do not write out the extended language of matched strings explicitly.

Iteration Now suppose we are given $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$, where I_1 and F_1 are the same up to some bijection. On input $(\mathbf{x}, \mathbf{w})$, we want to split $\mathbf{w}$ into $\mathbf{w}_1\mathbf{w}_2\mathbf{w}_3\ldots$, then apply $[\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w}_1)$ to get $\mathbf{y}_1$, $[\![\mathcal{A}_1]\!](\mathbf{y}_1, \mathbf{w}_2)$ to get $\mathbf{y}_2$, and so on. Then, the answer is the

Concatenation with data functions. If $\mathcal{A}_1 : I \twoheadrightarrow Z$ and $G_2 : Z \Rightarrow F$, then $\mathcal{A}_1 \cdot G_2 : I \twoheadrightarrow F$ implements the semantics

$$[\![\mathcal{A}_1 \cdot G_2]\!](\mathbf{x}, \mathbf{w}) = [\![G_2]\!]([\![\mathcal{A}_1]\!](\mathbf{x}, \mathbf{w})),$$

such that $\mathsf{size}(\mathcal{A}_1 \cdot G_2) = \mathsf{size}(\mathcal{A}_1) + \mathsf{size}(G_2) + O(|Z|)$. Likewise, if $G_1 : I \Rightarrow Z$ and $\mathcal{A}_2 : Z \twoheadrightarrow F$, then $G_1 \cdot \mathcal{A}_2 : I \twoheadrightarrow F$ implements the semantics

$$[\![G_1 \cdot \mathcal{A}_2]\!](\mathbf{x}, \mathbf{w}) = [\![\mathcal{A}_2]\!]([\![G_1]\!](\mathbf{x}), \mathbf{w}),$$

such that $\mathsf{size}(G_1 \cdot \mathcal{A}_2) = \mathsf{size}(G_1) + \mathsf{size}(\mathcal{A}_2) + O(|Z|)$.

union over all possible ways to write $\mathbf{w} = \mathbf{w}_1\mathbf{w}_2 \dots \mathbf{w}_k$ of $\mathbf{y}_k$. Let I be a set the same size as I_1, F_1 with bijections $\pi : I \to I_1$, $\rho : F \to F_1$. Then we implement this by taking $\mathcal{A} = (\mathcal{A}_1)^* = (Q, \Sigma, \Delta, I, I)$ with $Q = Q_1 \cup I$ and

$$\Delta = \Delta_1 \cup \big\{(\varepsilon, \{i'\}, \pi(i)', i') : i \in I\big\} \cup \big\{(\varepsilon, \{\rho(i)'\}, i', \rho(i)') : i \in I\big\}.$$

The idea is again very simple; we have a set of states I that is both initial and final; we always copy the values of these states into the input of $\mathcal{A}_1$ and copy the final states of $\mathcal{A}_1$ back into I. But the semantics is again more complicated. Here (unlike all other constructions), we do not necessarily preserve acyclicity. When we copy F_2 into I and back into I_2, this may then propagate back into F_2 again. Essentially, if $\mathcal{A}_1$ produces output on the empty data word, then $(\mathcal{A}_1)^*$ will always be $\top$, as this will create a cycle with least fixed point $\top$.

We assume that $\mathcal{A}_1$ is both output-synchronized and restartable. We can write configurations of $\mathcal{A}$ as $(\mathbf{c}, \mathbf{y})$, where $\mathbf{c}$ is a configuration of $\mathcal{A}_1$. On an input word $\mathbf{w} = (\sigma_1, d_1), \dots, (\sigma_k, d_k)$, let the sequence of configurations be $(\mathbf{c}_0, \mathbf{y}_0)$, $(\mathbf{c}_1, \mathbf{y}_1)$, $\dots$, $(\mathbf{c}_k, \mathbf{y}_k)$, so the output of $\mathcal{A}$ is $\mathbf{y}_k$. Then the least-fixed-point semantics of Equation (1) implies that, for $i = 1, \dots, k$, $\mathbf{y}_i$ is the least vector satisfying $\mathbf{y}_i = \big(\Delta_{\sigma_i, \mathbf{y}_i}(\mathbf{c}_{i-1}, d_i)\big)\,|_{F_1}$. Similarly, for $i = 0$, $\mathbf{y}_0$ is the least vector satisfying $\mathbf{y}_0 = (\Delta_{\mathtt{i}}(\mathbf{y}_0))\,|_{F_1}$. Now we want to show by induction that $\mathbf{c}_i$ simulates the list, over all possible splits of $\mathbf{w} = \mathbf{w}_1\mathbf{w}_2 \cdots \mathbf{w}_k$, of the configuration of $\mathcal{A}_1$ obtained by sequentially applying $\mathcal{A}_1$ k times. The proof of the inductive step is to take the property $\mathbf{y}_i = \big(\Delta_{\sigma_i, \mathbf{y}_i}(\mathbf{c}_{i-1}, d_i)\big)\,|_{F_1}$ and decompose the configuration $\Delta_{\sigma_i, \mathbf{y}_i}(\mathbf{c}_{i-1}, d_i)$ using the simulation relation, and see that it simulates the list of all splits $\mathbf{w} = \mathbf{w}_1 \cdots \mathbf{w}_k$ where $\mathbf{w}_k$ has size at least 1, plus the additional initialized thread $\Delta_{\mathtt{i}}(\mathbf{y}_i)$.

Iteration. Let $\mathcal{A} : I \twoheadrightarrow I$ be output-synchronized and restartable. Then $\mathcal{A}^* : I \twoheadrightarrow I$ satisfies

$$[\![\mathcal{A}^*]\!](\mathbf{x}, \mathbf{w}) = \bigsqcup_{\mathbf{w} = \mathbf{w}_1\mathbf{w}_2\cdots\mathbf{w}_k} [\![\mathcal{A}]\!](\dots [\![\mathcal{A}]\!]([\![\mathcal{A}]\!](\mathbf{x}, \mathbf{w}_1), \mathbf{w}_2) \dots, \mathbf{w}_k),$$

s.t. $\mathsf{size}(\mathcal{A}^*) = \mathsf{size}(\mathcal{A}) + O(|I|)$. It matches $\overline{L}(\mathcal{A}^*) = \overline{L}(\mathcal{A})^*$.

Properties of restartability. All operations except "support" preserve restartability. The "output-synchronized" property is also preserved by union, concatenation, and iteration, but is not guaranteed with parallel composition: $\mathcal{A}_1 \parallel \mathcal{A}_2$ is output-synchronized only if $\overline{L}(\mathcal{A}_1) = \overline{L}(\mathcal{A}_2)$.

Theorem 5.5. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are restartable, then so are $\mathcal{A}_1 \parallel \mathcal{A}_2$ and $\mathcal{A}_1 \sqcup \mathcal{A}_2$. If $\mathcal{A}_1$ is additionally output-synchronized, then $\mathcal{A}_1 \cdot \mathcal{A}_2$ and $\mathcal{A}_1{}^*$ are restartable. If $\mathcal{A}_1$ is restartable and output-synchronized and additionally $\overline{L}(\mathcal{A}_1) = \Sigma^*$, and if G is a data function where each output value is given by a single term over the input values, then $\oplus_G \mathcal{A}_1$ is restartable.

Proof. For $\mathcal{A}_1 \parallel \mathcal{A}_2$ and $\mathcal{A}_1 \sqcup \mathcal{A}_2$, we represent configurations of the machine has pairs $(\mathbf{c}_1, \mathbf{c}_2)$, and we define $(\mathbf{c}_1, \mathbf{c}_2) \sim [(\mathbf{c}_{1,1}, \mathbf{c}_{2,1}), \ldots, (\mathbf{c}_{1,k}, \mathbf{c}_{2,k})]$ if *both* $\mathbf{c}_1 \sim [\mathbf{c}_{1,1}, \ldots, \mathbf{c}_{1,k}]$ and $\mathbf{c}_2 \sim [\mathbf{c}_{2,1}, \ldots, \mathbf{c}_{2,k}]$. For prefix sum, the restartability holds for somewhat trivial reasons: if we restart with only $\vec{\bot}$, the output is $\bot$: if we restart with only one non-$\vec{\bot}$ thread, the output is the prefix-sum, and if we restart with two or more non-$\vec{\bot}$ threads, the output is $\top$ everywhere. For concatenation, we have configurations that are pairs $(\mathbf{c}_1, \mathbf{c}_2)$ of a configuration in $\mathcal{A}_1$ and one in $\mathcal{A}_2$. We define $(\mathbf{c}_1, \mathbf{c}_2) \sim [(\mathbf{c}_{1,1}, \mathbf{c}_{2,1}), \ldots, (\mathbf{c}_{1,k}, \mathbf{c}_{2,k})]$ if $\mathbf{c}_1 \sim [\mathbf{c}_{1,1}, \ldots, \mathbf{c}_{1,k}]$ and *there exists* sequences $l_{2,1}$, $l_{2,2}$, …, $l_{2,k}$, such that $\mathbf{c}_{2,i}$ simulates $l_{2,i}$ and $\mathbf{c}_2$ simulates the entire sequence of sequences, $l_{2,1} \circ l_{2,2} \circ \cdots \circ l_{2,k}$. The idea is that a configuration in $\mathcal{A} = \mathcal{A}_1 \cdot \mathcal{A}_2$ simulates a list of configurations where each configuration consists of only a single thread in $\mathcal{A}_1$, but may have many threads in $\mathcal{A}_2$ (since one thread in $\mathcal{A}_1$ may cause $\mathcal{A}_2$ to be restarted several times). However, we still need that *there exists* some further simulation of the configuration in $\mathcal{A}_2$ into a set of individual threads, such that the overall configuration of $\mathcal{A}_2$ in $\mathcal{A}$ simulates all of these individual threads. For iteration $\mathcal{A} = \mathcal{A}_1{}^*$, we have to do this recursively. The simulation on $\mathcal{A}$ includes $\mathcal{A}_1$ but extends it to the least relation such that whenever $\mathbf{c}_i \sim [\mathbf{c}_{i,1}, \ldots, \mathbf{c}_{i,k}]$ for each i, if $\mathbf{c} \sim [\mathbf{c}_1, \ldots, \mathbf{c}_k]$ then $\mathbf{c} \sim [\mathbf{c}_{i,j}]_{i,j}$. □

Theorem 5.6. Given a DT $\mathcal{A}$ as input, checking if $\mathcal{A}$ is restartable is PSPACE-complete.

Proof. Construct $\mathcal{P}$ as in the proof of Theorem 5.2, a DT over $(\mathbf{u}, \mathsf{UOp})$ where $\mathbf{u} = \{\star\}$. Use c_i and p_i to denote configurations of $\mathcal{A}$ and $\mathcal{P}$, respectively.

We first prove a lemma: that $\mathcal{A}$ is restartable iff $\mathcal{P}$ is restartable. The forward direction is immediate: define the relation $p \sim [p_1, p_2, \ldots, p_k]$ if there exists $c \sim [c_1, c_2, \ldots, c_k]$ such that p_i is the projection of c_i to $\mathbf{u}$; then facts (i), (ii), and (iii) are homomorphically preserved. The backward direction is nontrivial. We need to define the simulation relation between configurations and lists of configurations. We define the *reachable* relation $R \subseteq C \times [C]$ to be the minimal relation that is implied by properties (i) and (ii), i.e. the set of pairs $(c, [c_1, c_2, \ldots, c_k])$ reachable from initialization followed by some sequence of updates-with-restarts $\Delta_{\sigma,\mathbf{x}}$. We will show that R is a simulation by showing that (iii) holds of all reachable pairs. The key observation—which holds even if $\mathcal{A}$ is not restartable—is that for every reachable pair $(c, [c_1, c_2, \ldots, c_k])$, $c \geq c_i$ for all i (where $\geq$ is the coordinate-wise partial ordering on data vectors defined in §5.1). This is proven inductively. Using this we claim that R satisfies (iii). Let $(c, [c_1, c_2, \ldots, c_k])$ be reachable. Fix $f \in F$. Since $\mathcal{P}$ is restartable, we know that $c(f)$ and $c_1(f) \sqcup \cdots \sqcup c_k(f)$ are either both undefined, both defined, or both conflict. Thus the only way they can be unequal (violating (iii)) is if they are both in $\mathbb{D}$, and distinct. If they are both in $\mathbb{D}$, then $c_i(f) = \bot$ for all i except one, say $c_j(f) = d'$. But from the key observation above, $c(f) \geq c_j(f)$, and since $c(f), c_j(f) \in \mathbb{D}$, we have equality $c(f) \geq c_j(f)$.

We give a coNPSPACE algorithm to check restartability of a DT $\mathcal{A}$. By the above lemma, it is enough to work with $\mathcal{P}$ instead. So we need to check if there exists a reachable pair $(p, [p_1, \ldots, p_k])$, where p and p_i are configurations of $\mathcal{P}$, such that $F(p) = F(p_1) \sqcup F(p_2) \sqcup \cdots \sqcup F(p_k)$. But choose k to be minimal; then we do not need to keep track of $p_1, \ldots, p_{k-1}$, but can instead collapse these into a single configuration p'. Specifically, before the kth restart, suppose we are at $(p', [p'_1, p'_2, \dots, p'_{k-1}])$; then rather than keeping p'_1 through p'_{k-1}, we know the output will always be the same as taking p', so we keep track only of p'. Using this trick, the space required to store $(p, [p_1, \dots, p_k])$ is constant: three configurations of $\mathcal{P}$. Overall, we guess a sequence of moves to get to $(p', [p'_1, \dots, p'_{k-1}]$, then guess a sequence of moves to get to p from there, and guess a place to stop and try checking if $p(f) = p_1(f) \sqcup p_2(f) \sqcup \cdots \sqcup p_k(f)$ for all $f \in F$. The total space is bounded and some thread accepts if and only if there is a counterexample, meaning the machine is not restartable.

PSPACE-hardness can be shown by a reduction from the problem of universality for NFAs. We carefully exploit that if NFAs N_1 and N_2 are translated to DTs which always output $\perp$ or $\top$, and G is a single binary operation, the DT construction $(N_1 \parallel N_2) \cdot G$ is restartable iff there do not exist strings u, v such that $u \in L(N_1)$, $u \notin L(N_2)$, $uv \notin L(N_1)$, $uv \in L(N_2)$, or vice versa. □

Converting to restartable. It is shown in Theorem 5.8 that a DT of size m can be converted to a deterministic CRA of size $\exp(m)$; and that a deterministic CRA of size m can be converted into a *restartable* DT of size $O(m)$. This gives a procedure to convert DT to restartable DT, unfortunately with exponential blowup. Fortunately, Theorem 5.5 guarantees that such exponential blowup does not arise in the compilation of the QRE-Past language of §5.3.

5.3 The QRE-Past Monitoring Language

In this section we present the QRE-Past query language for quantitative runtime monitoring (Quantitative Regular Expressions with Past-time temporal operators). Each query compiles to a streaming algorithm, given as a DT, whose evaluation has precise complexity guarantees in the size of the query. Specifically, the complexity is a quadratic number of registers and quadratic number of operations to process each element, in the size of the query, independent of the input stream. Our language employs several constructs from the StreamQRE language [49]. To this core set of combinators we add the `prefix-sum` operation, `fill` and `fill-with` operations, and also past-time temporal logic operators which allow querying temporal safety properties: for example, "is the average of the last five measurements always more than two standard deviations above the average over the last two days?" We have picked constructs which we believe to be intuitive to program and useful in the application domains we have studied, but we do not intend them to be exhaustive; there are many other combinators which could be defined, added to the language, and implemented using the back-end support provided by the constructions of §5.2.

By compiling to the DT machine model, we show that the compiled code has the same precise complexity guarantee of the code produced by the StreamQRE engine of [49], including the additional temporal operators. Since compiled StreamQRE code was shown to have better throughput than popular existing streaming engines (RxJava, Esper, and Flink) when deployed on a single machine, this is good evidence that QRE-Past would see similar success with more flexible language constructs.

5.3.1 Syntax of QRE-Past

Expressions in the language are divided into three types: *quantitative queries* of two types, either base-level (α) or top-level (β), and *temporal queries* (φ). Base-level quantitative

$$
\begin{array}{lll}
\alpha := \mid \texttt{atom}(\sigma, t) & \{\sigma\} & \sigma \in \Sigma, t \in \mathsf{Tm}[\mathsf{cur}] \\
\quad\;\; \mid \texttt{eps}(t) & \{\varepsilon\} & t \in \mathsf{Tm}[\varnothing] \\
\quad\;\; \mid \texttt{or}(\alpha_1, \alpha_2) & \overline{L}(\alpha_1) \cup \overline{L}(\alpha_2) & \\
\quad\;\; \mid \texttt{split}(\alpha_1, \alpha_2, \mathit{op}) & \overline{L}(\alpha_1) \cdot \overline{L}(\alpha_2) & \mathit{op} \in \mathsf{Op}_2 \\
\quad\;\; \mid \texttt{iter}(\alpha_1, \mathit{init}, \mathit{op}) & (\overline{L}(\alpha_1))^* & \mathit{init} \in \mathbb{D}, \mathit{op} \in \mathsf{Op}_2 \\
\quad\;\; \mid \texttt{combine}(\alpha_1, \ldots, \alpha_k, \mathit{op}) & \overline{L}(\alpha_1) \cap \cdots \cap \overline{L}(\alpha_k) & \mathit{op} \in \mathsf{Op}_k;\ \text{well-typed if } \overline{L}(\alpha_1) = \cdots = \overline{L}(\alpha_k) \\
\quad\;\; \mid \texttt{prefix-sum}(\alpha_1, \mathit{init}, \mathit{op}) & \Sigma^* & \mathit{init} \in \mathbb{D}, \mathit{op} \in \mathsf{Op}_2;\ \text{well-typed if } L(\alpha_1) = \Sigma^* \\
\beta := \mid \alpha_1 & \overline{L}(\alpha_1) & \\
\quad\;\; \mid \texttt{fill}(\alpha_1) & L(\alpha_1) \cdot \Sigma^* & \\
\quad\;\; \mid \texttt{fill-with}(\alpha_1, \alpha_2) & L(\alpha_1) \cup \overline{L}(\alpha_2) &
\end{array}
$$

$$
\begin{array}{lll}
\varphi := \mid \beta_1 \mathbin{\mathit{comp}} \beta_2 & \Sigma^* & \mathit{comp} \in \{\leq, \geq, =\};\ \text{well-typed if } L(\beta_1) = L(\beta_2) = \Sigma^* \\
\quad\;\; \mid \varphi_1 \mathbin{\mathit{bop}} \varphi_2 \quad \mid \neg\varphi_1 & \Sigma^* & \mathit{bop} \in \{\wedge, \vee, \rightarrow, \leftrightarrow\} \\
\quad\;\; \mid \odot\varphi_1 \quad \mid \boxdot\varphi_1 \quad \mid \Diamond\varphi_1 & \Sigma^* & \\
\quad\;\; \mid \varphi_1 \mathcal{S}_w \varphi_2 \quad \mid \varphi_1 \mathcal{S}_s \varphi_2 & \Sigma^* &
\end{array}
$$

Figure 26: Summary of the QRE-Past language: syntax for quantitative queries α, β and temporal queries φ. The second column gives the rate of the query as a regular expression.

queries specify functions from data words to quantities (extended data values $\overline{\mathbb{D}}$), and are compiled to *restartable DTs* with a single initial state and single final state, of quadratic size. These queries are based on StreamQRE and the original Quantitative Regular Expressions of [16]. Top-level quantitative queries also specify functions from data words to quantities, but the compiled DT may not be restartable. Temporal queries specify functions from data words to Booleans, may be constructed from quantitative queries, and are compiled to DTs which output Booleans. Temporal queries are based on the operators of past-time temporal logic [51] and informed by successful existing work on monitoring of safety properties [37], which adapts to our setting via constructions on DTs.

We model Booleans as elements in $\mathbb{D}$. Thus, we assume that $0, 1 \in \mathbb{D}$, and that $\leq, \geq, = \in \mathsf{Op}_2$: these are comparison operations on data values returning 0 or 1. We also assume that we have Boolean operators $\neg \in \mathsf{Op}_1$ and $\wedge, \vee, \rightarrow, \leftrightarrow \in \mathsf{Op}_2$, which treat 0 as false and every $d \neq 0$ as true.

Each query has an associated regular *rate* $\overline{L}(\alpha)$, given by a regular expression on Σ defined recursively with the query. The rate expresses the set of strings on which the compiled DT is defined *or* conflict. For temporal queries φ, the rate is Σ^*. We also may refer to the *language* $L(\alpha) \subseteq \overline{L}(\alpha)$, which is the set of strings on which the compiled DT is defined. There are a few *typing restrictions*, mainly constraints on the rates of the queries. Because each rate is given by a regular expression, the typing restrictions are *type-checkable* in polynomial time. The typing restrictions arise in order to guarantee restartability so that the constructions of §5.2 apply.

5.3.2 Semantics and Compilation Algorithm

We describe each construction's semantics, and how it is directly implemented as a DT. For technical reasons, for each quantitative query (*not* for temporal queries) α or β we produce *two* DTs. The first is $\mathcal{A}_\alpha : X \twoheadrightarrow Y$, where $|X| = |Y| = 1$. The semantics will be such that $[\![\mathcal{A}_\alpha]\!](x, \mathbf{w})$ is the value of query α on input $\mathbf{w}$, if x is defined. So x is not really used, except to allow the machine to be restartable (at least one initial state is needed for restarts). The second is $\mathcal{I}_\alpha : X \twoheadrightarrow Y$, where $|X| = |Y| = 1$, which has the following *identity* semantics: $[\![\mathcal{I}_\alpha]\!](x, \mathbf{w}) = x$ if $[\![\mathcal{A}_\alpha]\!](x, \mathbf{w}) \in \mathbb{D}$, $\top$ if $[\![\mathcal{A}_\alpha]\!](x, \mathbf{w}) = \top$, and $\bot$ if $[\![\mathcal{A}_\alpha]\!](x, \mathbf{w}) = \bot$. In particular, $\mathcal{I}_\alpha$ is *equivalent* to $\mathcal{A}_\alpha$ (definition in §5.1). We use this second machine $\mathcal{I}_\alpha$ to *save* values for using later. For example, to implement $\texttt{split}(f, g, op)$ we concatenate the machine for f with a machine which both saves the output of f *and* starts g; then when g is finished we combine the saved output of f with the output of g via op. We will guarantee in the translation that $\mathcal{I}_\alpha$ has size only linear in the query, but $\mathcal{A}_\alpha$ has worst-case quadratic size.

Atomic expressions: $\texttt{atom}, \texttt{eps}$. The atomic expressions are the building blocks of all queries. For $t \in \mathsf{Tm}[\mathsf{cur}]$, the query $\texttt{atom}(\sigma, t)$ matches a data word containing a single character (σ, d), and returns t evaluated with $\texttt{cur} = d$. Similarly, the query $\texttt{eps}(t)$ matches the empty data word and returns the evaluation of t. Both of these are implementable using a DT with two states, $Q = \{q_i, q_f\}$, with $I = \{q_i\}$ and $F = \{q_f\}$. $\mathcal{A}_{\texttt{atom}(\sigma,t)}$ uses one transition from $\{q_i\}$ to q_f' with term t, and $\mathcal{A}_{\texttt{eps}(t)}$ uses an epsilon transition from $\{q_i'\}$ to q_f' with term t. These machines are restartable by a similar argument as the example immediately following Definition 5.4 (alternatively, if they aren't, just convert to an equivalent restartable DT as in §5.2.2, last paragraph). The definition of $\mathcal{I}_{\texttt{atom}(\sigma,t)}$ is the same as $\mathcal{A}_{\texttt{atom}(\sigma,t)}$ except that the term t in the transition is replaced by q_i; and likewise for $\mathcal{I}_{\texttt{eps}(t)}$.

Regular operators: $\texttt{or}, \texttt{split}, \texttt{iter}$. These regular operators are like traditional union, concatenation, and iteration (respectively), except that if the parsing of the string (data word) is not unique, the result will be $\top$. The union operation $\texttt{or}(\alpha_1, \alpha_2)$ should match every data word that matches either α_1 or α_2; if it matches only one, its value is that query, but if it matches both, its value is $\top$. In particular, conflict values "propagate upwards" because even if only one of α_1, α_2 matches, if the value is $\top$ then the result is $\top$. This is exactly the semantics of the DT construction $\mathcal{A}_{\alpha_1} \sqcup \mathcal{A}_{\alpha_2}$. It is restartable because $\mathcal{A}_{\alpha_1}$ and $\mathcal{A}_{\alpha_2}$ are restartable, by Theorem 5.5. Similarly, we can take $\mathcal{I}_{\texttt{or}(\alpha_1,\alpha_2)} = \mathcal{I}_{\alpha_1} \sqcup \mathcal{I}_{\alpha_2}$. Both of these constructions add only a constant to the size.

The operation $\texttt{split}(\alpha_1, \alpha_2, op)$ splits a data word $\mathbf{w}$ into two parts, $\mathbf{w}_1 \cdot \mathbf{w}_2$, such that $\mathbf{w}_1$ matches α_1 and $\mathbf{w}_2$ matches α_2. If there are multiple splits, the result is $\top$; otherwise, the result is $op(\alpha_1(\mathbf{w}_1), \alpha_2(\mathbf{w}_2))$. Here, we have to do some work to save the value of $\alpha_1(\mathbf{w}_1)$ in the DT construction. We implement $\texttt{split}$ as $\mathcal{A}_{\texttt{split}(\alpha_1,\alpha_2,op)} := (\mathcal{A}_{\alpha_1} \cdot (\mathcal{I}_{\alpha_2} \parallel \mathcal{A}_{\alpha_2})) \cdot G_{op}$, where G_{op} is a data function with two inputs y_1, y_2 which returns one output $op(y_1, y_2)$, where y_1 is the final state of $\mathcal{I}_{\alpha_2}$ and y_2 is the final state of $\mathcal{A}_{\alpha_2}$. Let's parse what this is saying. We split the string $\mathbf{w}$ into two parts $\mathbf{w}_1 \cdot \mathbf{w}_2$ such that $\mathbf{w}_i \in \overline{L}(\alpha_i)$, and apply α_1 to the first part; for the second part, we have a transducer which takes the output of α_1 as input and produces both that value as y_1, as well as the new output of α_2 as y_2. Then both of these are passed to G_{op} which returns $op(y_1, y_2)$. To define $\mathcal{I}_{\texttt{split}(\alpha_1,\alpha_2,op)}$ is easier: we take $\mathcal{I}_{\alpha_1} \cdot \mathcal{I}_{\alpha_2}$.

The operation $\texttt{iter}(\alpha_1, init, op)$ splits $\mathbf{w}$ into $\mathbf{w}_1 \cdots \mathbf{w}_k$ such that $\mathbf{w}_i \in \overline{L}(\alpha_1)$ and then *folds* *op* over the list of outputs of α_1, starting from *init*, to get a result: for instance if $k = 3$, the result is $op(op(op(init, \alpha_1(\mathbf{w}_1)), \alpha_1(\mathbf{w}_2)), \alpha_1(\mathbf{w}_3))$. If the parsing is not unique, the result is $\top$. We implement this as $\mathcal{A}_{\texttt{iter}(\alpha_1, init, op)} := G_{init} \cdot ((\mathcal{I}_{\alpha_1} \parallel \mathcal{A}_{\alpha_1}) \cdot G_{op})^*$, where G_{init} is a data function which outputs the initial value *init*. The idea here is that $(\mathcal{I}_{\alpha_1} \parallel \mathcal{A}_{\alpha_1}) \cdot G_{op}$ takes an input, both saves it and performs a new computation $\mathcal{A}_{\alpha_1}$, and then produces *op* of the old value and the new value. When this is iterated, we get the desired fold operation. For $\mathcal{I}_{\texttt{iter}(\alpha_1, init, op)}$ we can simply take $(\mathcal{I}_{\alpha_1})^*$.

We claim that these constructions preserve restartability. For concatenation, we need that the $\parallel$ is output-synchronized: we need that $\mathcal{A}_{\alpha_2}$ and $\mathcal{I}_{\alpha_2}$ have the same rate. This is true by construction: $\mathcal{I}$ is equivalent to $\mathcal{A}$ and only differs in that it is the identity function from input to output. So the three DTs concatenated are all output-synchronized. Restartability is preserved because the data function G_{op} is converted to a restartable DT in the concatenation construction. The size of the concatenation construction is bounded by a quadratic polynomial because we have added additional size equal to the size of $\mathcal{I}_{\alpha_2}$, which is bounded by a linear polynomial. For iteration, $\parallel$ is similarly only applied to equivalent DTs, and G_{op} is converted to a restartable DT in concatenation. As with `split`, the size of our construction includes the size of $\mathcal{A}_{\alpha_1}$ but adds a linear size due to inclusion of $\mathcal{I}_{\alpha_1}$, so we preserve a quadratic bound on size. The constructions $\mathcal{I}_{\alpha_1} \cdot \mathcal{I}_{\alpha_2}$ and $(\mathcal{I}_{\alpha_1})^*$ preserve a linear bound and are restartable because $\mathcal{I}_{\alpha_1}$ and $\mathcal{I}_{\alpha_2}$ are restartable.

Parallel combination: `combine`. This is the first operation in our language which requires a typing restriction. For $\texttt{combine}(\alpha_1, \ldots, \alpha_k, op)$, the computation is simple: apply every α_i to the input stream to get a result, then *combine* all these results via operation *op*. The implementation as a DT is $\mathcal{A}_{\texttt{combine}(\alpha_1, \ldots, \alpha_k, op)} := (\mathcal{A}_{\alpha_1} \parallel \cdots \parallel \mathcal{A}_{\alpha_k}) \cdot G_{op}$, where G_{op} applies *op* to the k final states of the $\parallel$. For $\mathcal{I}_{\texttt{combine}(\alpha_1, \ldots, \alpha_k, op)}$, we do the same thing but replace *op* by the term y_1 (i.e. we use $G_{y_1} : \{y_1, \ldots, y_k\} \Rightarrow \{y\}$ where y is the final output variable). The construction for `combine` is well-defined even if the typing restriction is not satisfied, but does not preserve restartability in that case. We use the non-restartable version in some other constructions. If the typing restriction *is* satisfied, then this exactly states that the left part of the concatenation is output-synchronized, and given that the right data function is converted to a restartable DT, restartability is preserved. The size of both $\mathcal{A}_{\texttt{combine}(\alpha_1, \ldots, \alpha_k, op)}$ and $\mathcal{I}_{\texttt{combine}(\alpha_1, \ldots, \alpha_k, op)}$ are linear in the sizes of the constituent DTs, so these constructions preserve the quadratic and linear bound on size, respectively.

Prefix sum: `prefix-sum`. The prefix sum $\texttt{prefix-sum}(\alpha_1, init, op)$ is defined only if α_1 is defined (not conflict) on all input. Its value should be $op(init, \alpha_1(\varepsilon))$ on the empty string, and then fold *op* over the outputs of α_1 after that. This is implemented directly using the prefix-sum constructor.

$$\mathcal{A}_{\texttt{prefix-sum}(\alpha_1, init, op)} := G_{init, init} \cdot \left(\bigoplus_{G_{op}} \mathcal{A}_{\alpha_1}\right).$$

Here, $G_{init, init}$ is a data function to return two copies of *init*. We need two copies because $\mathcal{A}_{\alpha_1}$ has one initial state, which needs an initial value (anything in $\mathbb{D}$ would work just as well).

Fill operations: `fill`, `fill-with`. These operations are ways to *fill in* the values which are $\bot$ and $\top$ with other values. This will not preserve restartability, so it is only allowed in top-level queries; however, it is useful to do this in order to get a query defined on all input data words, so that comparison β_1 *comp* β_2 can be applied. The query $\texttt{fill}(\alpha_1)$ always returns the *last* defined value returned by α_1. For instance, if the sequence of outputs of α_1 is $\bot, \top, 3, \top, 4, 5, \bot$, the outputs of $\texttt{fill}(\alpha_1)$ should be $\bot, \bot, 3, 3, 4, 5, 5$. The query $\texttt{fill-with}(\alpha_1, \alpha_2)$, instead of outputting the last defined value returned by α_2, just outputs the value returned by α_2 if α_1 is not defined. So, if α_2 is the constant always returning 0, the sequence of outputs of $\texttt{fill-with}(\alpha_1, \alpha_2)$ should be $0, 0, 3, 0, 4, 5, 0$.

To accomplish these constructions, we first obtain two DTs $\mathcal{A}_+$ and $\mathcal{A}_-$ which are defined when α_1 is defined and when α_1 is not defined, respectively: $\mathcal{A}_+ = [\mathcal{I}_{\alpha_1} \in \mathbb{D}]$ and $\mathcal{A}_- = [\mathcal{I}_{\alpha_1} = \bot] \sqcup [\mathcal{I}_{\alpha_1} = \top]$. Here, $[= \bot]$ and $[= \top]$ have quadratic blowup, but because we use $\mathcal{I}$ in the argument to those constructions instead of $\mathcal{A}$, $\mathcal{A}_+$ and $\mathcal{A}_-$ only have quadratic size. Now, let $\mathrm{fst}, \mathrm{snd} : \mathbb{D}^2 \to \mathbb{D}$ be the first and second projection operations. Then we implement the fill operations as:

$$\begin{aligned} \texttt{fill-with}(\alpha_1, \alpha_2) &:= \texttt{combine}(\mathcal{A}_{\alpha_1}, \mathcal{A}_+, \mathrm{fst}) \sqcup \texttt{combine}(\mathcal{A}_{\alpha_2}, \mathcal{A}_-, \mathrm{fst}) \\ \texttt{fill}(\alpha_1) &:= \oplus_G \left(\texttt{combine}(\mathcal{A}_{\alpha_1}, \mathcal{A}_+, \mathrm{fst}) \parallel \mathcal{A}_-\right), \end{aligned}$$

where G is a data function which expresses how to update the fill result based on the previous fill result, and whether $\mathcal{A}_{\alpha_1}$ is defined or not: if defined, we should take the new defined value, and otherwise, we should take the old fill result.

Comparison: $\leq, \geq, =$. The semantics of β_1 *comp* β_2 is just to apply *comp*: for example if *comp* is $<$, and if y_1 and y_2 are the outputs of β_1 and β_2 (which are always defined), then $\beta_1 < \beta_2$ should output $y_1 < y_2$ (which is 0 or 1). Therefore, this construction can be implemented as $\texttt{combine}(\beta_1, \beta_2, \mathit{comp})$. We do not need to worry about restartability for temporal queries, and we also don't define $\mathcal{I}$.

Boolean operators: $\wedge, \vee, \rightarrow, \leftrightarrow, \neg$. Similarly, the Boolean operators are implemented by applying the corresponding operation. For example, $\varphi_1 \vee \varphi_2$ is implemented as $\texttt{combine}(\varphi_1, \varphi_2, \vee)$.

Past-temporal operators: $\odot, \boxdot, \Diamond, \mathcal{S}_w, \mathcal{S}_s$. These have the usual semantics on finite traces: for example $\odot(\varphi_1)$ says that φ_1 was true at the previous item, and is false initially, and $\Diamond(\varphi_1)$ says that φ_1 was true at some point in the trace up to this point (including at the present time). The implementation of $\odot$ uses concatenation while the others all use prefix sum. Define $\mathcal{A}_{\odot(\varphi_1)} := \mathcal{A}_{\varphi_1} \cdot \mathcal{I}_\Sigma$, where we define $\mathcal{I}_\Sigma$ to be a DT which matches any data word of length 1, and has the identity semantics (returns the initial value as output). This concatenation is defined because $\mathcal{I}_\Sigma$ is restartable; it has the correct semantics because $\odot$ means to look at the prefix of the input except the last character. For the `prefix-sum` temporal operators, we illustrate only the example of $\Diamond(\varphi_1)$; the other cases are similar. Define a data function G which computes the truth value of $\Diamond(\varphi_1)$ on input $\mathbf{w}(\sigma, d)$ given its truth value on $\mathbf{w}$ and given the truth value of φ_1 on input $\mathbf{w}(\sigma, d)$ (so, G is just disjunction). Define $\mathcal{A}_{\Diamond(\varphi_1)} := G_{0,0} \cdot \oplus_G \mathcal{A}_{\varphi_1}$, where $G_{0,0}$ is a data function outputting two copies of 0 (false) to initialize the computation.

Complexity of QRE-Past evaluation. Our implementations give us the following theorem. In particular, combining with Theorem 5.1, the evaluation of any query on an input data stream requires quadratically many registers and quadratically many operations per element, independent of the length of the stream.

Theorem 5.7. For every well-typed base-level quantitative query α, the compilation described above via the constructions of §5.2 produces a restartable DT $\mathcal{A}_\alpha$ of quadratic size in the length of the query. For every well-typed top-level quantitative query β or temporal query φ, the compilation produces a DT of quadratic size which implements the semantics.

5.4 Succinctness

5.4.1 Comparison with Cost Register Automata

Cost register automata (CRAs) were introduced in [13] as a machine-based characterization of the class of *regular transductions*, which is a notion of regularity that relies on the theory of MSO-definable string-to-tree transductions. One advantage of CRAs over other approaches is that they suggest an obvious algorithm for computing the output in a streaming manner. A CRA has a finite-state control that is updated based only on the tag values of the input data word, and a finite set of write-only registers that are updated at each step using the given operations. The original CRA model is a deterministic machine, whose registers can hold data values as well as functions represented by terms with parameters. Each register update is required to be *copyless*, that is, a register can appear at most once in the right-hand-side expressions of the updates.

In [14], the class of *Streamable Regular* (SR) transductions is introduced, which has two equivalent characterizations: in terms of MSO-definable string-to-dag (directed acyclic graph) transductions without backward edges, and in terms of *possibly copyful* CRAs. Since the focus is on streamability, and terms can grow linearly with the size of the input stream, the registers are restricted to hold only values, not terms. This CRA model is expressively equivalent to DTs.

Theorem 5.8. The class of transductions computed by data transducers is equal to the class SR.

Proof sketch. It suffices to show semantics-preserving translations from (unambiguously nondeterministic, copyful) CRAs to DTs and vice versa. Suppose $\mathcal{A}$ is an unambiguous CRA with states Q and registers X. We construct a DT $\mathcal{B}$ with states $Q \times X$. In the other direction, suppose $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ is a DT. We construct a deterministic CRA $\mathcal{B}$ with states $\{\bot, \star, \top\}^Q$ and variables Q. A configuration of $\mathcal{B}$ consists of a state in $\{\bot, \star, \top\}^Q$ and an assignment $\mathbb{D}^Q$, and therefore uniquely specifies a configuration of $\mathcal{A}$. For each state in $\mathcal{B}$ and each σ, the transition to the next state can be determined from the set of transitions Δ_σ in $\mathcal{A}$. □

However, DTs—even restartable DTs—are exponentially more succinct than (unambiguously nondeterministic, copyful) CRAs. The succinct modular constructions on DTs are not possible on CRAs. For example, the parallel composition of CRAs requires a product construction, whereas the parallel composition of DTs employs a disjoint union construction ($\|$). This is why multiple parallel compositions of CRAs can cause an exponential blowup of the state space, but the corresponding construction on DTs causes only a linear increase in size.

Theorem 5.9. For some $(\mathbb{D}, \mathsf{Op})$, (restartable) DTs can be exponentially more succinct than CRAs.

Proof sketch. Let $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$, $\mathbb{D} = \mathbb{N}$, and $\mathsf{Op} = \{+\}$ (addition). Suppose that $\mathcal{A}_i$ for $i = 1, \ldots, k$ is a DT that outputs the sum of all values if the input contains σ_i, and

0 otherwise. Notice that $\mathcal{A}_i$ can be implemented with two state variables. Now, $\mathcal{A}$ is the restartable DT with $O(k)$ states that adds the results of $\mathcal{A}_1, \ldots, \mathcal{A}_k$. A CRA that implements the same function as $\mathcal{A}$ needs finite control that remembers which tags have appeared so far. This implies that the CRA needs exponentially many states, and this is true even if unambiguous nondeterminism is allowed. $\square$

5.4.2 Comparison with Finite-State Automata

Another perspective on succinctness is to compare DTs with finite automata for expressing regular languages. To simplify this, consider DTs over a singleton data set $\mathbb{D} = \{\star\}$, with no initial states and one final state. Each such DT $\mathcal{A}$ computes a regular language $\overline{L}(\mathcal{A})$. If we further restrict to *acyclic* DTs, they are exactly as succinct as *reversed alternating finite automata* (r-AFA). In particular, this implies that acyclic DTs (and hence DTs) are exponentially more succinct than DFAs and NFAs.

An r-AFA [23, 59] consists of $(Q, \Sigma, \delta, I, F)$ where the transition function δ assigns to each state in Q a *Boolean combination* of the previous values of Q. For example, we could assign $\delta(q_3) = q_1 \wedge (q_2 \vee \neg q_3)$. An r-AFA is equivalent to an AFA where the input string is read in the opposite order. The translation from DT to r-AFA copies the states, and on each update, sets each state to be equal to the disjunction of the transitions into it, where each transition is the conjunction of the source variables. Thus, the total size of δ is bounded by the size of the DT. For the other direction, we first remove negation in the standard way; then, conjunction becomes *op* and disjunction becomes $\sqcup$ (multiple transitions with a single target) in the DT.

It is known [23, 28] that L is recognized by a r-AFA with n states if and only if it is recognized by a DFA with 2^n states. This gives an exponential gap in state complexity between acyclic DTs and finite automata, both DFAs and NFAs. To see the gap for NFAs, consider a DFA with 2^n states which has no equivalent NFA with a fewer number of states. Acyclic DTs are a special case, so DTs are exponentially more succinct than both DFAs (uniformly) and NFAs (in the worst case).

5.4.3 Comparison with General Stream-Processing Programs

Finally, we consider a general model of computation for efficient streaming algorithms. The algorithm's maintained state consists of a fixed number of Boolean variables (in $\{0, 1\}$) and data variables (in $\mathbb{D}$), where the Boolean variables support all Boolean operations, but the data variables can only be accessed or modified using operations in Op. The behavior of the algorithm is given by an *initialization* function, an *update* function, a distinguished *output* data variable and a Boolean output flag (which is set to indicate output is present). The initialization and update functions are specified using a *loop-free* imperative language with the following constructs: assignments to Boolean or data variables, sequential composition, and conditionals. This model captures all efficient (bounded space and per-element processing time) streaming computations over a set of allowed data operations Op. We write $\textsc{Stream}(\mathsf{Op})$ to denote the class of such efficient streaming algorithms. The problem with the class $\textsc{Stream}(\mathsf{Op})$ is that it is not suitable for modular specifications. As the following theorem shows, it is not closed under the `split` combinator.

Theorem 5.10. Let $\Sigma = \{a, b\}$, $\mathbb{D} = \mathbb{N}$, and let Op be the family of operations that includes unary increment, unary decrement, the constant 0, and the binary equality predicate. Define the transductions $f, g : (\Sigma \times \mathbb{D})^* \rightharpoonup \mathbb{D}$ as follows:

$$L(f) = \{w \in \Sigma^* \;:\; |w|_a = 2 \cdot |w|_b\} \qquad\qquad L(g) = \{w \in \Sigma^* \;:\; |w|_a = |w|_b\}$$

$$f(\mathbf{w}) = \begin{cases} 1, & \text{if } \mathbf{w} \downarrow \Sigma \in L(f) \\ \bot, & \text{otherwise} \end{cases} \qquad\qquad g(\mathbf{w}) = \begin{cases} 1, & \text{if } \mathbf{w} \downarrow \Sigma \in L(g) \\ \bot, & \text{otherwise} \end{cases}$$

where $|w|_a$ is notation for the number of a's that appear in w. Both f and g are streamable functions (i.e. are computable in $\textsc{Stream}(\mathsf{Op})$), but $h = \mathtt{split}(f, g, (x, y) \mapsto 1)$ is not.

Proof. Both f and g can be implemented efficiently by maintaining two counters for the number of a's and the number of b's seen so far. On the other hand, any streaming algorithm that computes h requires a linear number of bits (in the size of the stream seen so far). Specifically, consider the behavior of such a streaming algorithm on inputs of the form $a(aab|aba)^n ab$. On these 2^n distinct inputs, each of length $3n + 3$, the streaming algorithm would have to reach 2^n different internal states, because the inputs are pairwise distinguished by reading in a further string of the form b^k. Thus on inputs of size $O(n)$ the streaming algorithm requires at least n bits to store the state. Any streaming algorithm in $\textsc{Stream}(\mathsf{Op})$, however, employs a finite number of integer registers whose size (in bits) can grow only logarithmically. □

Theorem 5.10 suggests that some restriction on the domains of transductions is necessary in order to maintain closure under modular constructions. We therefore enforce *regularity* of a generic streaming algorithm by requiring that the values of the Boolean variables depend solely on the input tags. That is, they do not depend on the input data values or the values of the data variables. Under this restriction, a streaming algorithm can be encoded as a DT of roughly the same size.

Theorem 5.11. A streaming algorithm of $\textsc{Stream}(\mathsf{Op})$ that satisfies the regularity restriction can be implemented by a DT over Op. This construction can be performed in linear time and space.

Proof sketch. Consider an arbitrary streaming algorithm of $\textsc{Stream}(\mathsf{Op})$ that satisfies the regularity restriction. Each data variable is encoded as a DT state that is always defined. Each Boolean variable b is encoded using two DT states x_b and $x_{\bar{b}}$ as follows: if $b = 0$ then $x_b = \bot$ and $x_{\bar{b}} = d_\star$, and if $b = 1$ then $x_b = d_\star$ and $x_{\bar{b}} = \bot$, where $d_\star$ is some fixed element of $\mathbb{D}$. □

6 Future Work

In the remainder of the thesis, we plan the following two research directions spanning approximately a 6-month period. Much of this research follows an implementation of the ideas laid out in sections 3 and 4 of our recent paper [17]. We plan to target machine learning and data analytics as target applications.

6.1 Implementation of synchronization schema types

We plan to implement synchronization schemas as a typing abstraction in Timely Dataflow [27] in Rust. The goal of this implementation is to offer a type-safe programming abstraction on top of the low-level dataflow model which guarantees ordering of events in each stream.

Type safety conditions are partially checked statically, and partially compiled to external *verification conditions* checked by an SMT solver.

Here is a sketch of the proposed solution: we write a library in which operators can be defined, where each operator has an input synchronization schema I and output synchronization schema O, and these can then be composed as a dataflow graph. For each operator, it is compiled to a back-end Timely Dataflow operator, which is implemented as a partitioned state update function; in the Timely representation, we also indicate partitioning of events consistent with the synchronization schema. Then, for each composition (edge in the dataflow graph), we generate a verification condition that states whether the output synchronization schema of each operator implies the input synchronization schema of the next operator.

6.2 Implementation of the execution model

Second, we plan to leverage the state-machine representation of data transducers as part of this framework. Here is how that would look: part of the library allows writing a query using a distributed variant of quantitative regular expressions. The query is then compiled to Timely with an input and output synchronization schema derived from the query as part of the dataflow representation above.

While this allows for ordering guarantees, one question is how to then derive performance guarantees for the operator. We propose a combination of empirical and analytical information: using the state machine representation we can calculate an analytical space and time bound for processing items, and then using empirical testing we can derive running time bounds for the operators in the graph. Another approach would be to define an abstract notion of latency and throughput in number of steps only (without the empirical component), and use this to derive logical verification conditions related to number of state updates required to process a single input item for each operator (latency), and number of state updates that can be processed in parallel (throughput).

6.3 Other ideas

We do not propose these for the thesis, but we are also working on an implementation of an optimization framework for database queries over smart watches for optimizing energy use. Another interesting direction is how to incorporate edge computing metrics into the stream processing framework; some ideas are laid out in [9].

Primary References

[1] Rajeev Alur, Dana Fisman, Konstantinos Mamouras, Mukund Raghothaman, and Caleb Stanford. Streamable regular transductions. *Theoretical Computer Science*, 807:15–41, 2020.

[2] Rajeev Alur, Phillip Hilliard, Zachary G Ives, Konstantinos Kallas, Konstantinos Mamouras, Filip Niksic, Caleb Stanford, Val Tannen, and Anton Xue. Synchronization schemas. 2021.

[3] Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. Automata-based stream processing. In *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[4] Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. Modular quantitative monitoring. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–31, 2019.

[5] Rajeev Alur, Konstantinos Mamouras, Caleb Stanford, and Val Tannen. Interfaces for stream processing systems. In *Principles of Modeling*, pages 38–60. Springer, 2018.

[6] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Diffstream: differential output testing for stream processing programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[7] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Stream processing with dependency-guided synchronization. *arXiv preprint arXiv:2104.04512*, 2021.

[8] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G Ives, and Val Tannen. Data-trace types for distributed stream processing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 670–685, 2019.

[9] Caleb Stanford. Geo-distributed stream processing. *University of Pennsylvania Doctoral Written Preliminary Exam (WPE-2), Technical Report*, 2020.

Other References

[10] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR '05)*, pages 277–289, 2005.

[11] Daniel J. Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[12] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, August 2013.

[13] Rajeev Alur, Loris D'Antoni, Jyotirmoy Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular functions and cost register automata. In *Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '13)*, 2013.

[14] Rajeev Alur, Dana Fisman, Konstantinos Mamouras, Mukund Raghothaman, and Caleb Stanford. Streamable regular transductions, In submission, 2018.

[15] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In *Proceedings of the 25th European Symposium on Programming (ESOP '16)*, pages 15–40, 2016.

[16] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In *Proceedings of the 25th European Symposium on Programming (ESOP '16)*, pages 15–40, 2016.

[17] Rajeev Alur, Phillip Hilliard, Zachary G Ives, Konstantinos Kallas, Konstantinos Mamouras, Filip Niksic, Caleb Stanford, Val Tannen, and Anton Xue. Synchronization schemas. 2021.

[18] Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. Modular quantitative monitoring. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–31, 2019.

[19] Rajeev Alur, Konstantinos Mamouras, Caleb Stanford, and Val Tannen. Interfaces for stream processing systems. In *Principles of Modeling*, pages 38–60. Springer, 2018.

[20] Kevin Ashton et al. That 'internet of things' thing. *RFID journal*, 22(7):97–114, 2009.

[21] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[22] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

[23] Ashok K Chandra, Dexter C Kozen, and Larry J Stockmeyer. Alternation. *Journal of the ACM (JACM)*, 28(1), 1981.

[24] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[25] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[26] Volker Diekert and Grzegorz Rozenberg. *The Book of Traces*. World Scientific, 1995.

[27] Frank McSherry et al. Timely dataflow (rust implementation). https://github.com/TimelyDataflow/timely-dataflow/, 2014. [Online; accessed January 14, 2021].

[28] Abdelaziz Fellah, Helmut Jürgensen, and Sheng Yu. Constructions for alternating finite automata. *International journal of computer mathematics*, 35(1-4), 1990.

[29] Danyel Fisher, Rob DeLine, Mary Czerwinski, and Steven Drucker. Interactions with big data analytics. *interactions*, 19(3):50–59, 2012.

[30] Apache Software Foundation. Apache flink. https://flink.apache.org/, 2019. [Online; accessed March 31, 2019].

[31] Apache Software Foundation. Apache samza. http://samza.apache.org/, 2019. [Online; accessed March 31, 2019].

[32] Apache Software Foundation. Apache spark streaming. https://spark.apache.org/streaming/, 2019. [Online; accessed March 31, 2019].

[33] Apache Software Foundation. Apache storm. http://storm.apache.org/, 2019. [Online; accessed March 31, 2019].

[34] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, 1998.

[35] KAHN Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[36] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. Bigdebug: Debugging primitives for interactive big data processing in spark. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 784–795. IEEE, 2016.

[37] Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 6(2), 2004.

[38] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):1–34, 2014.

[39] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[40] Cisco Global Cloud Index. Forecast and methodology, 2016–2021 white paper. *Updated: February*, 1, 2018.

[41] Theodore Johnson, Shanmugavelayutham Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. A heartbeat mechanism and its application in gigascope. In *Proceedings of the 31st international conference on Very large data bases*, pages 1079–1088. VLDB Endowment, 2005.

[42] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Diffstream: differential output testing for stream processing programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[43] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Stream processing with dependency-guided synchronization. *arXiv preprint arXiv:2104.04512*, 2021.

[44] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[45] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, 2000.

[46] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[47] Nancy A Lynch. *Distributed algorithms.* Elsevier, 1996.

[48] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G. Ives, and Sanjeev Khanna. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '17, pages 693–708, New York, NY, USA, 2017. ACM.

[49] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G. Ives, and Sanjeev Khanna. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In *Proceedings of the 38th Conference on Programming Language Design and Implementation*, PLDI '17, 2017.

[50] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G Ives, and Val Tannen. Data-trace types for distributed stream processing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 670–685, 2019.

[51] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer Science & Business Media, 2012.

[52] Antoni Mazurkiewicz. Trace theory. In *Advanced course on Petri nets*, pages 278–324. Springer, 1986.

[53] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[54] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, August 2017.

[55] Online. Aurora project page. http://cs.brown.edu/research/aurora/.

[56] Online. The borealis project. http://cs.brown.edu/research/borealis/public/.

[57] Online. Rust programming language. https://www.rust-lang.org/.

[58] Online. Storm github release commit 9d91adb. https://github.com/nathanmarz/storm/commit/9d91adbdbde22e91779b91eb40805f598da5b004.

[59] Kai Salomaa, Xiuming Wu, and Sheng Yu. Efficient implementation of regular languages using reversed alternating finite automata. *Theoretical Computer Science*, 231(1), 2000.

[60] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. Safe data parallelism for general streaming. *IEEE transactions on computers*, 64(2):504–517, 2013.

[61] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[62] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[63] Twitter. Heron. https://apache.github.io/incubator-heron/, 2019. [Online; accessed March 31, 2019].

[64] Alexandre Vianna, Waldemar Ferreira, and Kiev Gama. An exploratory study of how specialists deal with testing in data stream processing applications. *arXiv preprint arXiv:1909.11069*, 2019.

[65] Tian Xiao, Jiaxing Zhang, Hucheng Zhou, Zhenyu Guo, Sean McDirmid, Wei Lin, Wenguang Chen, and Lidong Zhou. Nondeterminism in mapreduce considered harmful? an empirical study on non-commutative aggregators in mapreduce programs. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 44–53. ACM, 2014.

[66] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.