Modular Quantitative Monitoring

RAJEEV ALUR, University of Pennsylvania, United States
KONSTANTINOS MAMOURAS, Rice University, United States
CALEB STANFORD, University of Pennsylvania, United States

In real-time decision making and runtime monitoring applications, declarative languages are commonly used as they facilitate modular high-level specifications with the compiler guaranteeing evaluation over data streams in an efficient and incremental manner. We introduce the model of *Data Transducers* to allow *modular* compilation of queries over streaming data. A data transducer maintains a finite set of data variables and processes a sequence of tagged data values by updating its variables using an allowed set of operations. The model allows unambiguous nondeterminism, exponentially succinct control, and combining values from parallel threads of computation. The semantics of the model immediately suggests an efficient streaming algorithm for evaluation. The expressiveness of data transducers coincides with *streamable regular transductions*, a robust and streamable class of functions characterized by MSO-definable string-to-DAG transformations with no backward edges. We show that the novel features of data transducers, unlike previously studied transducers, make them as succinct as traditional imperative code for processing data streams, but the structuring of the transition function permits modular compilation. In particular, we show that operations such as parallel composition, union, prefix-sum, and quantitative analogs of combinators for unambiguous parsing, can be implemented by natural and succinct constructions on data transducers. To illustrate the benefits of such modularity in compilation, we define a new language for quantitative monitoring, QRE-Past, that integrates features of past-time temporal logic and quantitative regular expressions. While this combination allows a natural specification of a cardiac arrhythmia detection algorithm in QRE-Past, compilation of QRE-Past specifications into efficient monitors comes for free thanks to succinct constructions on data transducers.

CCS Concepts: • **Information systems → Data streaming**; • **Software and its engineering → Domain specific languages**; • **Theory of computation → Quantitative automata**;

Additional Key Words and Phrases: quantitative monitoring, data stream processing, runtime verification

ACM Reference Format:
Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. 2019. Modular Quantitative Monitoring. *Proc. ACM Program. Lang.* 3, POPL, Article 50 (January 2019), 31 pages. https://doi.org/10.1145/3290363

1 INTRODUCTION

Applications ranging from network traffic engineering to runtime monitoring of autonomous control systems require computation over data streams in an efficient and incremental manner. Declarative programming is a particularly appealing approach to specify the desired logic in such applications as it can provide natural and high-level constructs for processing streaming data with guaranteed bounds on computational resources used by the compiled implementation. This has motivated the development of a number of declarative query languages. For example, in runtime verification, a monitor observes a sequence of events produced by a system, and issues an alert

Authors' addresses: Rajeev Alur, Computer and Information Science, University of Pennsylvania, United States, alur@cis.upenn.edu; Konstantinos Mamouras, Computer Science, Rice University, United States, mamouras@rice.edu; Caleb Stanford, Computer and Information Science, University of Pennsylvania, United States, castan@cis.upenn.edu.

2475-1421/2019/1-ART50
https://doi.org/10.1145/3290363

when a violation of a safety property is detected, where the safety property is described in a temporal logic with past-time operators such as *always-in-the-past* and *since* [Havelund and Roşu 2004; Manna and Pnueli 2012]. In quantitative monitoring, a monitor associates a numerical value with an input stream of data values, where the desired computation is described using *quantitative regular expressions* (QREs) that combine regular patterns with numerical aggregation operations such as min, max, sum, and average [Alur et al. 2016; Mamouras et al. 2017; Yuan et al. 2017]. In each such case, the declarative specification is automatically compiled into a monitor that adheres to the streaming model of computation [Muthukrishnan 2005]: memory and per-item processing time is polynomial in the size of the specification of the query and, roughly speaking, does not grow with the length of the input stream.

In existing query languages over streaming data, while a programmer can specify the desired computation in a modular fashion using constructs of the query language, the compiler generates monolithic code for a given query. What is lacking though is an intermediate representation for streaming computations that supports composition operations with succinct constructions so that high-level queries can be compiled modularly. The motivation for such a model is two-fold. From a practical viewpoint, it can facilitate the design of new query languages. For instance, suppose a user wants to specify a monitoring property using past-time temporal logic, where the atomic predicates involve comparing quantitative summaries defined using QREs. Such a specification contains combinators from two different languages (QREs and past-temporal logic), and we could try to design a compiler from scratch for streaming evaluation of the more expressive, integrated language. However, if we have a *modular* compilation algorithm for the combinators of the two component languages, we get a compiler for the integrated language for free. From a theoretical viewpoint, designing such a representation is a technical challenge since it needs to support both combining values from parallel threads of computation (i.e. parallel composition) and unambiguous regular parsing. In particular, although QREs can be compiled into quantitative automata known as *cost register automata* [Alur et al. 2013], since this compilation has provably exponential lower bound, it is not employed by current QRE evaluation algorithms, and in fact, no existing formalism can support modular compilation of QREs. The main contribution of this paper is the model of *Data Transducers* (DT) as this desired modular intermediate representation for streaming computations.

A data transducer processes a data stream—a sequence of tagged data values—and produces a numerical (or Boolean) value using a fixed set of data variables that are updated using a constant number of operations as it processes each tagged data value. A DT can be viewed as a quantitative generalization of (unambiguous) NFAs. Whereas an NFA configuration consists of a finite set of states, each of which is either inactive or active, a DT configuration consists of a finite set of data variables, each of which can be inactive (*undefined*), active with a value (*defined*), or in a special "conflict" mode (*conflicted*). A DT configuration thus consists of succinctly represented finite control integrated with data values. As a DT computes by consuming tagged data values, it updates its variables using a specified allowed set of operations. The values of defined variables can be combined using operations to form new values, but there is also the possibility of a "collision". This is analogous to how two tokens of active NFA states can be merged into one token during evaluation when they are placed on the same state. Since the merging of data values is not in general a meaningful operation, a collision of values results in a variable being set to conflict. Since multiple transitions can write to the same data variable while processing a single tagged data value, and the updated value of a variable can depend on the updated values of the others, the semantics is defined using fixed points. We show how this semantics can be implemented by an efficient streaming algorithm for evaluation that executes a linear (in the size of DT) number of data operations while processing each tagged data value.

The language of a DT, i.e. the set of stream histories for which its output is defined, is a regular language over the tags of the input stream. In fact, DTs capture a robust class of functions with an elegant logical characterization: MSO-definable string-to-DAG transformations with a special "no backward edges" requirement. This class, which we call *streamable regular transductions*, has been studied in [Alur et al. 2018; Courcelle 1994; Engelfriet and Maneth 1999], and the closure properties of this class, as opposed to some specific constructs supported by query languages in the existing literature, guide the choice of operations over DTs for which we seek succinct constructions.

In particular, we show that DTs are closed under quantitative concatenation, quantitative iteration, union, and parallel composition operations, and that the corresponding constructions are succinct. We also consider the *prefix-sum* operation that combines the outputs on all prefixes using a specified aggregator; this also has a simple and succinct construction on DTs. Temporal operators such as "always in the past", "sometime in the past", and "since" can be implemented using prefix-sum. The design choices in the precise formal definition of the model turn out to be critical in these constructions. A key restriction on DTs, which we call *restartability*, that is required for constructions related to unambiguous parsing is identified. This restriction says that it is possible to "restart" the automaton during a computation by placing new data values at its initial states. Then, although we only need to store a single automaton configuration in memory, the output is the same as if multiple copies of the automaton were computing independently on multiple stream suffixes as long as only one of these copies ultimately contributes to the final output. This ability is necessary for efficient unambiguous parsing: several parsing possibilities are explored simultaneously, but the required space is constant.

To illustrate the benefits of modular compilation, we define a new query language, called QRE-Past, that combines the features of past-time temporal logic and QREs. We specify a cardiac arrhythmia detection algorithm [Abbas et al. 2018; Zdarek and Israel 2016] in QRE-Past to illustrate how the combination of features leads to a natural high-level specification. The theory of DTs immediately leads to a streaming evaluation algorithm for QRE-Past, since every construct in QRE-Past maps to a corresponding construction on component DTs without causing blow-up. In fact, there is nothing sacred about QRE-Past: the designer of a high-level query language over streaming data for a specific domain can introduce new combinators, in addition to the ones in this paper, as long as there are corresponding succinct constructions on the low-level model of DTs.

Finally, while there are existing models with identical expressiveness, DTs are exponentially more succinct (for instance, compared to unambiguous cost register automata). To gain a better understanding of the expressiveness and succinctness of DTs, consider a (generic) streaming algorithm that maintains a fixed number of Boolean and data variables, and processes each tagged data value by updating these variables by executing a loop-free code. While such algorithms capture *all* streaming computations, the class of all streaming computations is not suitable for modular specifications. For instance, consider the quantitative concatenation operation: given transductions f and g, and a binary data operation op, $h = \texttt{split}(f, g, op)$ splits the inputs stream w uniquely into two parts $w = w_1 w_2$ and returns $h(w) = op(f(w_1), g(w_2))$. While DTs are closed under this operation, the class of all streaming algorithms is not. We can enforce regularity of a generic streaming algorithm by requiring, for instance, that the updates to the Boolean variables are not influenced by the values of the data variables. We show that streaming algorithms with these restrictions can be translated to DTs without any blow-up, thus establishing that DTs are the most succinct (up to a constant factor) representation of streamable regular transductions. The structure of a DT—as variables ranging over undefined/defined/conflict values and update code as a set of transitions of a particular form, as opposed to traditional loop-free update code—not only enforces regularity, but is also what allows us to define succinct constructions on the representation.

Outline of paper. §2 introduces the model of data transducers with illustrative examples. In §3 we consider a number of semantic operations with corresponding succinct constructions on DTs, and we define and study the key property of restartability necessary for some of them. In §4, we define the query language QRE-Past, and show how constructions on DTs immediately yield modular compilation into a streaming evaluation algorithm. We also show how QRE-Past is useful in specifying a cardiac arrhythmia detection algorithm. §5 discusses the expressiveness and succinctness of DTs compared to cost register automata, to finite automata, and to general streaming computations. We compare with related work in §6, and conclude in §7.

2 DATA TRANSDUCERS

2.1 Preliminaries

To model data streams we use *data words* [Bouyer et al. 2003]. 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 $\boldsymbol{w} \in (\Sigma \times \mathbb{D})^*$. We write $\boldsymbol{w} \downarrow \Sigma$ to denote the projection of $\boldsymbol{w}$ to a string in Σ^*. We use bold $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$ 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 $(\mathbb{D}, \mathsf{Op})$, 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. (For instance, we rely on this guarantee for the theorems in §2.5 and for the translation from QRE-Past in §4.2. It's not needed for most of the constructions in §3.)

$\sqcup$	$\bot$	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 $\boldsymbol{x}, \boldsymbol{y}$. The partial order extends coordinate-wise to an ordering $\boldsymbol{x} \leq \boldsymbol{y}$ on data vectors $\boldsymbol{x}, \boldsymbol{y} \in \overline{\mathbb{D}}^X$. 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.

2.2 Syntax and Semantics

Let $(\mathbb{D}, \mathrm{Op})$ 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*.
  - $t \in \mathsf{Tm}[X \cup \{\mathrm{cur}\}]$ 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 $\mathrm{cur} \notin X$. 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* $\boldsymbol{x} \in \overline{\mathbb{D}}^I$, which assigns an extended data value to each initial state. Second, an *input data word* $\boldsymbol{w} \in (\Sigma \times D)^*$, which is a sequence of tagged data values to be processed by the transducer. On input $(\boldsymbol{x}, \boldsymbol{w})$, 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 $\boldsymbol{c} \in \overline{\mathbb{D}}^Q$. 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 $\Delta_\sigma(\boldsymbol{c}, d)(q) := \boldsymbol{c}'(q')$, where $\boldsymbol{c}' \in \overline{\mathbb{D}}^{Q \cup Q' \cup \{\mathrm{cur}\}}$ is the *least vector* satisfying $\boldsymbol{c}'(\mathrm{cur}) = d$; for all $q \in Q$, $\boldsymbol{c}'(q) = \boldsymbol{c}(q)$; and

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

where we define $[\![t]\!](\boldsymbol{c}'|_X)$ to be $\bot$ if there exists $x \in X$ such that $\boldsymbol{c}'(x) = \bot$; otherwise, $\top$ if there exists $x \in X$ such that $\boldsymbol{c}'(x) = \top$; otherwise, if all variables in X are defined, then $[\![t]\!](\boldsymbol{c}'|_X)$ is the value of the expression t with variables assigned the values in $\boldsymbol{c}'$. So, $[\![t]\!](\boldsymbol{c}'|_X)$ produces $\bot$ or $\top$ if some variable in X is $\bot$ 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 $\boldsymbol{c}_0 \in \overline{\mathbb{D}}^Q$, given the initial vector $\boldsymbol{x} \in \overline{\mathbb{D}}^I$. 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 $\boldsymbol{x}$. Define that $\boldsymbol{x}(q) = \bot$ if $q \notin I$. Then define $\Delta_\mathtt{i}(\boldsymbol{x}) = \boldsymbol{c}'$, where $\boldsymbol{c}'$ is the *least vector* satisfying, for all $q \in Q$, $\boldsymbol{c}'(q') = \boldsymbol{x}(q) \sqcup \bigsqcup_{(\mathtt{i}, X, q', t) \in \Delta} [\![t]\!](\boldsymbol{c}'|_X)$.

Now $\mathcal{A}$ is evaluated on input $(\boldsymbol{x}, \boldsymbol{w}) \in \overline{\mathbb{D}}^I \times (\Sigma \times \mathbb{D})^*$ by starting from the initial configuration and applying the update functions in sequence as illustrated in Figure 1. Finally, the output $\boldsymbol{y} \in \overline{\mathbb{D}}^F$ is given by $\boldsymbol{y} = \boldsymbol{c}|_F$, the projection of $\boldsymbol{c}$ to the final states.

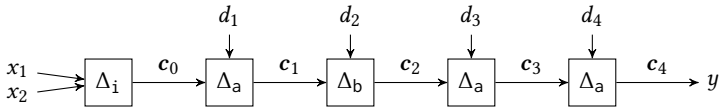

Fig. 1. Example evaluation of a data transducer $\mathcal{A}$ with two initial states and one final state on initial vector $(\boldsymbol{x}_1, \boldsymbol{x}_2)$ and an input data word $\boldsymbol{w}$ consisting of four characters (tagged data values): (a, d_1), (b, d_2), (a, d_3), (a, d_4), to produce output y. Here $\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3$, and $\boldsymbol{c}_4$ 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.

Why both initial states and initial transitions? Initial states I give the DT the ability to take an arbitrary initial vector $\boldsymbol{x}$ as input. Initial transitions Δ_i give the DT the ability to compute initial values for its *non-initial* states. Particularly, this includes the possibility of producing output by initializing the final states F. If the desired computation does not require an initial input, then it is appropriate to take $I = \varnothing$ and instead initialize only with initial transitions (see examples in §2.4).

Why do variables in X unused in t affect the semantics? All variables in X control whether a transition (σ, X, q', t) evaluates to undefined, defined, or conflict. However, this is not strictly necessary, because the same thing can be achieved by using the unused variable in a no-op operation, e.g. for an unused variable x, replacing term t with $t + 0 \cdot x$. We do not employ unused variables in the examples of §2.4, but they are convenient and do arise in the constructions of §3 and §4.

Generalizing to multiple data types. To keep the presentation simple, we have assumed that every state (if defined) takes values in the same set $\mathbb{D}$. It's possible to allow states with multiple data types; then, the signature would include operations between the various types.

2.3 Streaming Evaluation Algorithm

Complexity assumptions. We are interested in evaluation in the streaming setting, where the input data word $\boldsymbol{w}$ arrives character-by-character and must be processed in real-time without being stored. After processing each character (i.e. tagged data value), we should produce the output on the current prefix of $\boldsymbol{w}$. The complexity bounds of interest in this setting are (1) the maximum *time to process each element* of $\boldsymbol{w}$ and (2) the maximum *space usage* while reading the entire word. Both should be very small compared to the large input $\boldsymbol{w}$ (ideally constant or poly-logarithmic).

For DTs, we do not provide such unconditional bounds, as evaluation costs depend on the precise data type and operations supported. Instead, we provide a constant bound on (1) the *number of data operations* in Op to process each element and (2) the *number of data registers* of type $\mathbb{D}$ that need to be stored. In many cases, these can translate to efficient streaming complexity bounds when instantiated with a particular signature $(\mathbb{D}, \mathrm{Op})$. These bounds assume a sequential implementation.

Theorem 2.1. *Evaluation of a data transducer $\mathcal{A}$, with number of states n and size m on input $(\boldsymbol{x}, \boldsymbol{w})$, 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 $\boldsymbol{w}$.*

Proof. The evaluation algorithm is given in Figure 2. First, we know that we can process each element (σ, d) of the stream by applying Δ_σ with $\mathsf{cur} = d$ and the previous configuration as input, as in Figure 1. At each step we can produce the output $\boldsymbol{y}$ given by the extended data values of the final states. Therefore, the nontrivial remaining task is to compute $\Delta_\sigma(\boldsymbol{c}, d)$ from $\boldsymbol{c}$ and d, for a given $\sigma \in \Sigma$ and configuration $\boldsymbol{c}$, which in particular means computing the least vector $\boldsymbol{c}' \in \overline{\mathbb{D}}^{Q \cup Q' \cup \{\mathsf{cur}\}}$ defined in Equation (1). Computing the initial configuration $\Delta_i(\boldsymbol{x})$ can be done similarly.

$\boldsymbol{c} \leftarrow \Delta_{\mathtt{i}}(\boldsymbol{x})$; produce output $\boldsymbol{y} = \boldsymbol{c}|_F$
for each character (σ, d) in $\boldsymbol{w}$ **do**
  for each state $q \in Q$ **do** $val(q) \leftarrow \boldsymbol{c}(q)$; $val(q') \leftarrow \bot$
  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 τ to *worklist*
  for each $q \in Q$ **do** $\boldsymbol{c}(q) \leftarrow val(q')$
  produce output $\boldsymbol{y} = \boldsymbol{c}|_F$

Fig. 2. Data transducer evaluation algorithm (Theorem 2.1). On input $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ over $(\mathbb{D}, \mathrm{Op})$, an initial vector $\boldsymbol{x} \in \overline{\mathbb{D}}^I$, and a data stream $\boldsymbol{w} \in (\Sigma \times \mathbb{D})^*$, produces the output vector $\boldsymbol{y} \in \overline{\mathbb{D}}^F$ on each prefix of $\boldsymbol{w}$.

To compute the least fixed point, we maintain current values $val(q')$ for each variable q' in Q' and $val(\tau)$ for each transition τ in Δ_σ, as well as a worklist of values (in Q' or Δ_σ) that need to be updated. While the worklist is nonempty, we pick a value to visit. If it is a transition (respectively, state), we update its value and add the state (respectively, *each* transition) which depends on it to the worklist *only if its value will increase in the lattice*. This guarantees that each state or transition can only be added to the worklist at most 3 times (once initially, and at most twice when its value increases). Moreover, we can determine whether the value of a state or transition will increase in constant time. For a state, this is because it will always increase unless its current value is $\top$. For a transition, we have to additionally maintain a count of how many of its source states are $\bot$, which we do in the map num_undef. The transition's value will increase if it is currently defined, *or* if it is currently undefined and the number of undefined source states has just dropped to 0.

This whole process to compute $\Delta_\sigma(\boldsymbol{c}, d)$ requires $O(m)$ operations: we visit each transition at most 3 times; and we visit each (state, transition) pair, where the state is one of the transition's source states, at most 3 times (once for each time that state is added to the worklist). □

An instructive special case of Theorem 2.1 is when the transitions Δ_σ, for all $\sigma \in \Sigma \cup \{\mathtt{i}\}$, are acylic. By this we mean that the following directed graph is acyclic: take vertices $Q \cup Q'$, with an edge from x to q' if there is a transition (σ, X, q', t) with $x \in X$. If the transitions Δ_σ are acyclic for all σ then we say this is an *acyclic DT*. Then the least fixed point is also the *only* fixed point and can be obtained by iterating over the states $q' \in Q'$ in a single pass, in a topologically sorted order, and assigning the value of $\boldsymbol{c}'(q')$. This is an $O(m)$ algorithm, so in this easier case we can match the result of Theorem 2.1 without using a worklist.

2.4 Examples

We do not envision that DTs would be directly programmed by users, due to the conceptual difficulty of tracking undefined, defined, and conflicted values. Rather, DTs would be a low-level, back-end model for streaming and monitoring. The purpose of this section is mainly to illustrate, informally and through examples, the basic features and execution semantics of the model.

Q = {sum1, sum2, sum3, avg}, $I = \emptyset$, F = {avg}

transitions(i) = $\emptyset$

transitions(a) = || sum1′ := cur
|| sum2′ := sum1 + cur
|| sum3′ := sum2 + cur
|| avg′ := sum3′ / 3

transitions(b) = || sum1′ := sum1
|| sum2′ := sum2
|| sum3′ := sum3

transitions(#) = $\emptyset$

Example evaluation on input

$\boldsymbol{w}$ = (a, 6)(a, 5)(a, 7)(b, 2)(a, 8)(#, 0)(b, 2)(a, 7).

$\boldsymbol{w}$ (input)	sum1	sum2	sum3	avg (output)
	⊥	⊥	⊥	⊥
(a, 6)	6	⊥	⊥	⊥
(a, 5)	5	11	⊥	⊥
(a, 7)	7	12	18	6.000
(b, 2)	7	12	18	⊥
(a, 8)	8	15	20	6.667
(#, 0)	⊥	⊥	⊥	⊥
(b, 2)	⊥	⊥	⊥	⊥
(a, 7)	7	⊥	⊥	⊥

Fig. 3. Data transducer $\mathcal{A}_1$ monitoring a stream of purchase events for two types of items, tagged a and b, and # to represent the end of each day. Throughout the day we output the average price in a sliding window of the last three a-items. The language of strings on which $\mathcal{A}_1$ produces output is $(a \cup b \cup \#)^* a b^* a b^* a$.

We present only *acyclic* DTs in this section, and we take $I = \emptyset$: all initialization is done with initial transitions Δ_i. Additionally, we use the abbreviation $q' := t$ to denote a transition (σ, X, q', t), where X is exactly the set of variables present in the term t (in contexts where σ is clear). In general, X may include other variables unused in t, and the semantics of the transition does depend on the unused variables as well (see §2.2, "Why do variables in X unused in t affect the semantics?").

Pattern matching. DTs are based on the idea of merging *data registers* and *finite control* into the single set of "state variables" Q. Suppose we wish to monitor a stream of a-events, b-events, and #-events, where each a- or b-event is the price at which an item was bought, and # indicates the end of a day. We thus have $\mathbb{D} = \mathbb{Q}$ and $\Sigma = \{a, b, \#\}$. For the operations Op, we allow $+, -, \cdot$, max, min, division / (this must return a default value on division by 0), and integer constants. Suppose we want to output the average price of a *sliding window* containing the last three a prices, which resets at the end of the day. This is essentially a *pattern match* over the input tags to locate the last three, which are then averaged. $\mathcal{A}_1$ in Figure 3 is based on this idea. The transitions listed under transitions(σ) are those labeled with σ; we use || to emphasize that the transitions are not ordered.

The machine $\mathcal{A}_1$ uses state variables sum1, sum2, and sum3 to keep track of the sum of the last 1, 2, and 3 a prices (in the current day). Each variable matches a certain pattern of tags in the input stream, namely, strings with at least 1, 2, and 3 a's so far. In addition to pattern-matching, the variables are updated to keep track of the sum. For example, the transition sum2′ := sum1 + cur indicates that *if* sum1 was defined before then sum2 should now be defined and equal to the sum plus the current data value. The transition avg′ := sum3′ / 3 indicates that *if* sum3 is *now* defined (note the sum3′), then avg should be set to the average of the last three prices.

Multiple transitions with a single target. The machine $\mathcal{A}_1$ has a simplifying syntactic property that for every $\sigma \in \Sigma$ and for every state q', there is only one transition $q' := t$. In other words, there is only one *rule* stating how to assign q' a value. In general, there may be multiple rules, and the resulting value of q' will be the union ($\sqcup$) over all transitions. For instance, suppose we have the same input stream over $\Sigma = \{a, b, \#\}$, and we want to output the average price of an a-item at the end of each day. However, if there are no a-items on a given day, we instead want to output the average from the previous day. A machine implementation of this is provided by $\mathcal{A}_2$ in Figure 4.

In $\mathcal{A}_2$, sum and count store the sum of a-items and number of a-items on each day, respectively, and are defined only if there has been at least one a. On the other hand, prev_avg stores the

$Q = \{\mathsf{sum}, \mathsf{count}, \mathsf{avg}, \mathsf{prev\_avg}\},\ I = \emptyset,\ F = \{\mathsf{avg}\}$

$\mathsf{transitions(i)} = \parallel \mathsf{prev\_avg}' := 0$

$\mathsf{transitions(a)} = \parallel \mathsf{sum}' := \mathsf{prev\_avg} \cdot 0 + \mathsf{cur}$

$\parallel \mathsf{sum}' := \mathsf{sum} + \mathsf{cur}$

$\parallel \mathsf{count}' := \mathsf{prev\_avg} \cdot 0 + 1$

$\parallel \mathsf{count}' := \mathsf{count} + 1$

$\mathsf{transitions(b)} = \parallel \mathsf{sum}' := \mathsf{sum}$

$\parallel \mathsf{count}' := \mathsf{count}$

$\parallel \mathsf{prev\_avg}' := \mathsf{prev\_avg}$

$\mathsf{transitions(\#)} = \parallel \mathsf{avg}' := \mathsf{sum} \,/\, \mathsf{count}$

$\parallel \mathsf{avg}' := \mathsf{prev\_avg}$

$\parallel \mathsf{prev\_avg}' := \mathsf{avg}'$

Example evaluation on input

(b, 2)(a, 6)(b, 2)(a, 8)(a, 7)(#, 0)(b, 2)(#, 0)(a, 7)(a, 6).

$\boldsymbol{w}$ (input)	sum	count	avg (output)	prev_avg
	$\bot$	$\bot$	$\bot$	0
(b, 2)	$\bot$	$\bot$	$\bot$	0
(a, 6)	6	1	$\bot$	$\bot$
(b, 2)	6	1	$\bot$	$\bot$
(a, 8)	14	2	$\bot$	$\bot$
(a, 7)	21	3	$\bot$	$\bot$
(#, 0)	$\bot$	$\bot$	7.0	7.0
(b, 2)	$\bot$	$\bot$	$\bot$	7.0
(#, 0)	$\bot$	$\bot$	7.0	7.0
(a, 7)	7	1	$\bot$	$\bot$
(a, 6)	13	2	$\bot$	$\bot$

Fig. 4. Data transducer $\mathcal{A}_2$ monitoring the stream to produce, at the end of each day, either the average price of an a-item (if there was at least one a) or the previous average (if there was no a). When there are multiple transitions $q' := t_1$ and $q' := t_2$, the semantics is such that we assign $q' := t_1 \sqcup t_2$.

previous average, but it is defined only if there has *not* been any a yet. (We also initialize this to 0 arbitrarily on the very first day.) The state avg stores the output, and is only defined after a # event. The logic of this computation involves two places where we need to have multiple transitions targeting a state. First, on receiving an a, we set sum to be equal to the previous sum plus the current value, but we also set it to be equal to $0 \cdot \mathsf{prev\_avg} + \mathsf{cur}$. This works because exactly one of these two values will be defined, and the other will be $\bot$: either we have seen an a already, in which case we can update the sum, or we haven't seen one yet, in which case prev_avg is still defined. Second, the overall output avg has two possible values, either sum/count or prev_avg, and again, exactly one of these two values will be defined, and the other will be $\bot$. Thus, we have designed $\mathcal{A}_2$ so that each union operation ($\sqcup$) never produces a conflict ($\top$).

Combining output from parallel threads of computation. Our final example attempts to illustrate the feature which gives DTs their succinctness (see §5): the ability to update multiple computations independently and then combine their results. Suppose we want to compute, at the end of each day, the difference between the maximum price of a and the maximum price of b, if there was at least one a and at least one b. The DT $\mathcal{A}_3$ in Figure 5 implements this computation. The state a_init of $\mathcal{A}_3$ stores 0 and is only defined if we haven't seen an a yet; similarly for b_init.

2.5 Regularity

Data transducers define *regular transductions* on data words (see §5.1). 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, together with some regular property of the string of tags. For data vectors $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \overline{\mathbb{D}}^X$, we say that $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ are *equivalent*, and write $\boldsymbol{x}_1 \equiv \boldsymbol{x}_2$, if for all $x \in X$, $\boldsymbol{x}_1(x)$ and $\boldsymbol{x}_2(x)$ are both undefined, both defined, or both conflict.

Theorem 2.2. *Let $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ be a DT over $(\mathbb{D}, Op)$. Then: (i) For all initial vectors $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathbb{D}^I$, and for all input words $\boldsymbol{w}_1, \boldsymbol{w}_2$, if $\boldsymbol{x}_1 \equiv \boldsymbol{x}_2$ and $\boldsymbol{w}_1 \downarrow \Sigma = \boldsymbol{w}_2 \downarrow \Sigma$, then $[\![\mathcal{A}]\!](\boldsymbol{x}_1, \boldsymbol{w}_1) \equiv [\![\mathcal{A}]\!](\boldsymbol{x}_2, \boldsymbol{w}_2)$. (ii) For every equivalence class of initial vectors $\boldsymbol{x}$ and equivalence class of output vectors $\boldsymbol{y}$, the set of strings $\boldsymbol{w} \downarrow \Sigma$ such that $[\![\mathcal{A}]\!](\boldsymbol{x}, \boldsymbol{w}) \equiv \boldsymbol{y}$ is regular.*

Q = {a_init, a_max, b_init, b_max, ab_diff}
$I = \emptyset$, F = {ab_diff}

transitions(i) = || a_init′ := 0
|| b_init′ := 0

transitions(a) = || a_max′ := a_init + cur
|| a_max′ := max(a_max, cur)
|| b_max′ := b_max
|| b_init′ := b_init

transitions(b) = || b_max′ := b_init + cur
|| b_max′ := max(b_max, cur)
|| a_max′ := a_max
|| a_init′ := a_init

transitions(#) = || ab_diff′ := a_max − b_max
|| a_init′ := 0
|| b_init′ := 0

Example evaluation on input
(b, 2)(a, 6)(b, 3)(b, 1)(a, 8)(#, 0)(b, 2)(#, 0)(a, 7)(b, 1).

w (input)	a_init	a_max	b_init	b_max	ab_diff (output)
	0	⊥	0	⊥	⊥
(b, 2)	0	⊥	⊥	2	⊥
(a, 6)	⊥	6	⊥	2	⊥
(b, 3)	⊥	6	⊥	3	⊥
(b, 1)	⊥	6	⊥	3	⊥
(a, 8)	⊥	8	⊥	3	⊥
(#, 0)	0	⊥	0	⊥	5
(b, 2)	0	⊥	⊥	2	⊥
(#, 0)	0	⊥	0	⊥	⊥
(a, 7)	⊥	7	0	⊥	⊥
(b, 1)	⊥	7	⊥	1	⊥

Fig. 5. Data transducer $\mathcal{A}_3$ monitoring the stream to produce, at the end of each day, the difference between the maximum price of an a-item and the maximum price of a b-item.

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 $\boldsymbol{x}_1, \boldsymbol{w}_1$ and $\boldsymbol{x}_2, \boldsymbol{w}_2$, and (ii) follows because $\mathcal{P}$ has finitely many possible configurations. □

We can thus define the *language* of $\mathcal{A}$ to be $L(\mathcal{A}) = \{\boldsymbol{w} \downarrow \Sigma \mid [\![\mathcal{A}]\!](\boldsymbol{x}, \boldsymbol{w}) \in \mathbb{D}^F \text{ for some } \boldsymbol{x} \in \mathbb{D}^I\}$, so $L(\mathcal{A}) \subseteq \Sigma^*$. This is the set of tag strings $\boldsymbol{w} \downarrow \Sigma$ such that, if the initial vector of values is all defined, after reading in $\boldsymbol{w}$ 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 $\{\boldsymbol{w} \downarrow \Sigma \mid [\![\mathcal{A}]\!](\boldsymbol{x}, \boldsymbol{w}) \in (\mathbb{D} \cup \{\top\})^F \text{ for some } \boldsymbol{x} \in (\mathbb{D} \cup \{\top\})^I\}$. An immediate corollary of Theorem 2.2 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 $\boldsymbol{x}_1 \equiv \boldsymbol{x}_2$ and for all $\boldsymbol{w}$, $[\![\mathcal{A}_1]\!](\boldsymbol{x}_1, \boldsymbol{w}) \equiv [\![\mathcal{A}_2]\!](\boldsymbol{x}_2, \boldsymbol{w})$.

Theorem 2.3. *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 2.1).

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 $\mathcal{A}$. □

3 CONSTRUCTIONS ON DATA TRANSDUCERS

As discussed in the introduction, 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 §4: 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 $op \in \mathrm{Op}_k$ 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).

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 $op \in \mathrm{Op}_2$, we wish to implement $\mathtt{split}(f, g, op)$: on input $\boldsymbol{w}$, split the input stream into two parts, $\boldsymbol{w} = \boldsymbol{u} \cdot \boldsymbol{v}$, such that $f(\boldsymbol{u}) \neq \bot$ and $g(\boldsymbol{v}) \neq \bot$ (respectively, f matches $\boldsymbol{u}$ and g matches $\boldsymbol{v}$), and return $op(f(\boldsymbol{u}), g(\boldsymbol{v}))$. Assume that the decomposition of $\boldsymbol{w}$ into $\boldsymbol{u}$ and $\boldsymbol{v}$ such that $f(\boldsymbol{u}) \neq \bot$ and $g(\boldsymbol{v}) \neq \bot$ is unique. In order to naively implement this operation, on an input string $\boldsymbol{w}$, we must not only keep track of the current configuration of f on $\boldsymbol{w}$, but for *every* split $\boldsymbol{w} = \boldsymbol{uv}$ where f matches $\boldsymbol{u}$, we must keep track of the current configuration of g on $\boldsymbol{v}$. If there are many possible prefixes $\boldsymbol{u}$ of $\boldsymbol{w}$ such that $f(\boldsymbol{u}) \neq \bot$, 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 §2.5. 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 §3.1 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 §3.2, 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.

3.1 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 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 $(\boldsymbol{x}, \boldsymbol{w})$ 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 i transition to preserve restartability, which will be discussed in §3.2. 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 $[\![\mathcal{A}]\!](\boldsymbol{x}, \boldsymbol{u}) = ([\![\mathcal{A}_1]\!](\boldsymbol{x}, \boldsymbol{u}), [\![\mathcal{A}_2]\!](\boldsymbol{x}, \boldsymbol{u}))$. Here, $(\boldsymbol{y}_1, \boldsymbol{y}_2)$ denotes the vector $\boldsymbol{y} \in \overline{\mathbb{D}}^F$ that is $\boldsymbol{y}_1$ on F_1 and $\boldsymbol{y}_2$ on F_2. Parallel composition is commutative and associative. The utility of parallel composition is that it allows us to combine the outputs $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ later on. This is accomplished by *concatenation* with another DT which combines the outputs (§3.2).

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

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

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 $(\boldsymbol{x}, \boldsymbol{w})$ 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

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

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 $\boldsymbol{x}$ will be copied into I_1 and I_2, and on every data word the output values $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ 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 $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ are defined, the output will be $\top$. We see therefore that the semantics is such that $[\![\mathcal{A}]\!](\boldsymbol{x},\boldsymbol{u}) = [\![\mathcal{A}_1]\!](\boldsymbol{x},\boldsymbol{u}) \sqcup [\![\mathcal{A}_2]\!](\boldsymbol{x},\boldsymbol{u})$. 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

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

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 $\boldsymbol{w}$, such that the output on the empty data word is $\boldsymbol{y}_1^{(0)}$, the output after receiving one character of the data word is $\boldsymbol{y}_1^{(1)}$, and in general the output after k characters is $\boldsymbol{y}_1^{(k)}$. The problem is to return the *sum* of these outputs: we want a DT that returns $\boldsymbol{y}^{(i)} = \boldsymbol{y}_1^{(0)} + \cdots + \boldsymbol{y}_1^{(i)}$ after receiving i characters. This is called the *prefix sum* because $\boldsymbol{y}_1^{(k)}$ 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" $\boldsymbol{y}^{(i-1)} \in \overline{\mathbb{D}}^{F}$, combines it with the new output of $\mathcal{A}_1$, $\boldsymbol{y}_1^{(i)} \in \overline{\mathbb{D}}^{F_1}$, and produces the next "sum" $\boldsymbol{y}^{(i)} \in \overline{\mathbb{D}}^{F}$. So, we'll have $G(\boldsymbol{y}^{(i-1)}, \boldsymbol{y}_1^{(i)}) = \boldsymbol{y}^{(i)}$. We want a DT that, on input initial values for I_1 and initial values $\boldsymbol{y}^{(-1)}$ for F, will return $\boldsymbol{y}^{(i)}$. 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 $\boldsymbol{y}^{(0)} = [\![G]\!](\boldsymbol{y}^{(-1)}, \boldsymbol{y}_1^{(0)})$. 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 $[\![G]\!](\boldsymbol{y}^{(0)}, \boldsymbol{y}_1^{(1)})$, and then $[\![G]\!](\boldsymbol{y}^{(1)}, \boldsymbol{y}_1^{(2)})$, 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

$$[\![\oplus_G \mathcal{A}_1]\!]((\boldsymbol{x},\boldsymbol{y}),\varepsilon) = [\![G]\!](\boldsymbol{y}, [\![\mathcal{A}_1]\!](\boldsymbol{x},\varepsilon))$$

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

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 §3.2—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 §4, 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 $\boldsymbol{x} \in \mathbb{D}^{I_1}$ (*not* $\overline{\mathbb{D}}^{I_1}$), all $\boldsymbol{u} \in (\Sigma \times \mathbb{D})^*$, and all $f_1 \in F_1$, if $[\![\mathcal{A}_1]\!](\boldsymbol{x}, \boldsymbol{u})(f_1) = \bot$ then $[\![\mathcal{A}_\bot]\!](\boldsymbol{u})(f_1) \in \mathbb{D}$, and otherwise, $[\![\mathcal{A}_\bot]\!](\boldsymbol{u})(f_1) = \bot$. 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 §2.2, 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 $f \in F$:

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

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

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

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

3.2 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 $\mathtt{split}(f, g, op)$. During the execution of f on input $\boldsymbol{w}$, whenever the current prefix $\boldsymbol{u}$ of $\boldsymbol{w}$ matches, i.e. $f(\boldsymbol{u}) \neq \bot$, we could (naively and inefficiently) implement $\mathtt{split}(f, g, op)$ by keeping a separate configuration (thread) of g from that point forward. For example, suppose that $\boldsymbol{w} = (\mathsf{a}, d_1)(\mathsf{b}, d_2)(\mathsf{a}, d_3)(\mathsf{a}, d_4)$, 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: $\boldsymbol{c}_1$

on input $(\mathsf{b}, d_2)(\mathsf{a}, d_3)(\mathsf{a}, d_4)$, $\boldsymbol{c}_2$ on input (a, d_4), and $\boldsymbol{c}_3$ on input ε. Suppose that each configuration $\boldsymbol{c}_i$ includes an final state with the value of $\boldsymbol{y}_i = op(f(\boldsymbol{u}), g(\boldsymbol{v}))$. The value of $\mathtt{split}(f, g, op)$ could then be computed as the *union* of the outputs from all these threads: $\mathtt{split}(f, g, op)(\boldsymbol{w}) = \boldsymbol{y}_1 \sqcup \boldsymbol{y}_2 \sqcup \boldsymbol{y}_3$. We apply the union here because we expect the split $\boldsymbol{w} = \boldsymbol{u} \cdot \boldsymbol{v}$, where $\boldsymbol{u} \in \overline{L}(f)$ and $\boldsymbol{v} \in \overline{L}(g)$, to be unique. Thus all but at most one of $\boldsymbol{y}_i$ will be $\bot$, and the union gives us the unique answer (if any).

A DT will be called restartable if a *single configuration* $\boldsymbol{c}$ can *simulate* the behavior of these several configurations $\boldsymbol{c}_1, \boldsymbol{c}_2$, and $\boldsymbol{c}_3$. 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 $\boldsymbol{c} \sim [\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3]$ is intended to capture that $\boldsymbol{c}$ is observationally indistinguishable from the sequence $\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3$. For starters, we require that the output is the same: if $\boldsymbol{y}$ is the output of $\boldsymbol{c}$, then $\boldsymbol{y} = \boldsymbol{y}_1 \sqcup \boldsymbol{y}_2 \sqcup \boldsymbol{y}_3$. 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 $[\boldsymbol{c}_1, \ldots, \boldsymbol{c}_k]$ is simulated by some $\boldsymbol{c}$, 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 $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$. (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 $\boldsymbol{x}_1 \sqcup \cdots \sqcup \boldsymbol{x}_k$ 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 $\boldsymbol{c} \sim [\boldsymbol{c}_1, \ldots, \boldsymbol{c}_k]$, and we now read in a character (σ, d) to g. Simultaneously, we start zero or more new threads represented by the vector $\boldsymbol{x}$ (e.g., $\boldsymbol{x}$ is the new output produced by f on input (σ, d)). Then if we update and re-initialize the initial states of $\boldsymbol{c}$ with $\boldsymbol{x}$, that configuration should simulate updating each $\boldsymbol{c}_i$ separately, *and* adding one or more new threads represented by $\boldsymbol{x}$. 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 $\boldsymbol{c}$ and simultaneously restart new threads by placing values $\boldsymbol{x}$ 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 $\boldsymbol{x} \in \overline{\mathbb{D}}^I$ we define a generalized evaluation function $\Delta_{\sigma,\boldsymbol{x}} : \overline{\mathbb{D}}^Q \times \mathbb{D} \to \overline{\mathbb{D}}^Q$. This represents executing Δ_σ and then starting *zero or more* new threads, by initializing the new initial states with $\boldsymbol{x}$. We modify the least fixed point definition of $\boldsymbol{c}'$ in Equation 1) to include the new initialization on states I': $\boldsymbol{c}'$ is the least vector satisfying

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

where $\boldsymbol{x}(q) = \bot$ if $q \notin I$. This resembles the way we already incorporated $\boldsymbol{x}$ into the definition of $\Delta_{\mathtt{i}}$. We restrict the vector $\boldsymbol{x}$ 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 3.1 (Restartability). Let $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ be a DT over signature $(\mathbb{D}, \mathrm{Op})$; 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 $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k \in \mathcal{X}$, $\Delta_{\mathrm{i}}\left(\bigsqcup_{i=1}^{k} \boldsymbol{x}_i\right) \sim [\Delta_{\mathrm{i}}(\boldsymbol{x}_1), \ldots, \Delta_{\mathrm{i}}(\boldsymbol{x}_k)]$. (If $k = 0$, we get $\Delta_{\mathrm{i}}(\vec{\bot}) \sim []$, 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 $\boldsymbol{c}, \boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_k$, $\hat{\boldsymbol{c}}_1, \hat{\boldsymbol{c}}_2, \ldots, \hat{\boldsymbol{c}}_l$, if $\boldsymbol{c} \sim [\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_k]$ and $\Delta_{\mathrm{i}}(\boldsymbol{x}) \sim [\hat{\boldsymbol{c}}_1, \hat{\boldsymbol{c}}_2, \ldots, \hat{\boldsymbol{c}}_l]$ then

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

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

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 $\boldsymbol{c} \sim [\boldsymbol{c}_1, \ldots, \boldsymbol{c}_k]$ whenever $\boldsymbol{c} = \bigsqcup_{i=1}^{k} \boldsymbol{c}_i$. Then (i), (ii), and (iii) hold. For example, the base case says that $x = \bigsqcup_{i=1}^{k} x_k$, then $(x, \bot) \sim [(x_1, \bot), \ldots, (x_k, \bot)]$, 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 $(x_1, \bot)$; 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 $(\top, \top) \sim [(x_1, x_1 + d), (x_2, \bot)]$. The problem is that this violates (iii): the output of $\mathcal{A}$ is $\top$, which is not the same as $(x_1 + d) \sqcup \bot = x_1 + d$.

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 $\{\bot, \star, \top\}$. There are only finitely many configurations up to equivalence. This is why restartability is decidable (see Theorem 3.3).

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 $(\boldsymbol{x}, \boldsymbol{w})$, consider all splits of $\boldsymbol{w}$ into two strings, $\boldsymbol{w} = \boldsymbol{w}_1\boldsymbol{w}_2$. Apply $\mathcal{A}_1$ to $(\boldsymbol{x}, \boldsymbol{w}_1)$ to get a result $\boldsymbol{y}_1$, and apply $\mathcal{A}_2$ to $(\boldsymbol{y}_1, \boldsymbol{w}_2)$ to get $\boldsymbol{y}_2$. Return the union ($\sqcup$) over all such splits of $\boldsymbol{y}_2$. In particular, assuming there is only one way to split $\boldsymbol{w} = \boldsymbol{w}_1\boldsymbol{w}_2$ such that $\boldsymbol{y}_2$ does not end up being undefined, this operation splits the input string uniquely into two parts such that $\mathcal{A}_1$ matches $\boldsymbol{w}_1$ and $\mathcal{A}_2$ matches $\boldsymbol{w}_2$, 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 $\Delta_{\sigma,\boldsymbol{x}}$ from earlier. Let us represent configurations of $\mathcal{A}$ by $(\boldsymbol{c}_1, \boldsymbol{c}_2)$, where $\boldsymbol{c}_i$ is the component restricted to Q_i, i.e. the

induced configuration of $\mathcal{A}_i$. Now consider an input $(\boldsymbol{x}, \boldsymbol{w})$ to $\mathcal{A}$. We see that for the ith configuration of $\mathcal{A}$ $(\boldsymbol{c}_1^{(i)}, \boldsymbol{c}_2^{(i)})$, $\boldsymbol{c}_1^{(i)}$ is the same as the ith configuration of $\mathcal{A}_1$ on input $(\boldsymbol{x}, \boldsymbol{w})$. Moreover, if $\boldsymbol{y}_1^{(i)}$ is the ith output of $\mathcal{A}_1$, this is used to reinitialize $\mathcal{A}_2$; so we see that $\boldsymbol{c}_2^{(i)} = \Delta_{\sigma, \boldsymbol{y}_1^{(i)}}(\boldsymbol{c}_2^{(i-1)}, d)$ (where this is the update function of $\mathcal{A}_2$). The output $\boldsymbol{y}_2^{(i)} = \boldsymbol{c}_2^{(i)}|_F$ of $\mathcal{A}_2$ is the output of $\mathcal{A}$.

Assume that $\mathcal{A}_1$ is *output-synchronized*: this means that each $\boldsymbol{y}_1^{(i)} \in \mathcal{X}$, i.e., all values are $\bot$ 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 $\boldsymbol{c}_2$ by a list of configurations where each configuration is $\mathcal{A}_2$ on a different suffix of $\boldsymbol{w}$. In particular, we recursively replace $\Delta_{\sigma, \boldsymbol{y}_1^{(i)}}(\boldsymbol{c}_2^{(i-1)}, d)$ with the list of configurations for $\Delta_\sigma(\boldsymbol{c}_2^{(i-1)}, d)$ and a single new thread $\Delta_1(\boldsymbol{y}_1^{(i)})$. Because $\boldsymbol{y}_1^{(i)} \in \mathcal{X}$, 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

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

such that $\mathrm{size}(\mathcal{A}_1 \cdot \mathcal{A}_2) = \mathrm{size}(\mathcal{A}_1) + \mathrm{size}(\mathcal{A}_2) + O(|Z|)$. 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 §4. 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 $(\boldsymbol{x}, \boldsymbol{w})$, return $[\![G_2]\!]([\![\mathcal{A}_1]\!](\boldsymbol{x}, \boldsymbol{w}))$. 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 $(\boldsymbol{x}, \boldsymbol{w})$, returns $[\![\mathcal{A}_2]\!]([\![G_1]\!](\boldsymbol{x}), \boldsymbol{w})$. 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.

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

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

such that $\mathrm{size}(\mathcal{A}_1 \cdot G_2) = \mathrm{size}(\mathcal{A}_1) + \mathrm{size}(G_2) + O(|Z|)$. 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

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

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

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 $(\boldsymbol{x}, \boldsymbol{w})$, we want to split $\boldsymbol{w}$ into $\boldsymbol{w}_1 \boldsymbol{w}_2 \boldsymbol{w}_3 \ldots$, then apply $[\![\mathcal{A}_1]\!](\boldsymbol{x}, \boldsymbol{w}_1)$ to get $\boldsymbol{y}_1$, $[\![\mathcal{A}_1]\!](\boldsymbol{y}_1, \boldsymbol{w}_2)$ to get $\boldsymbol{y}_2$, and so on. Then, the answer is the union over all possible ways to write $\boldsymbol{w} = \boldsymbol{w}_1 \boldsymbol{w}_2 \ldots \boldsymbol{w}_k$ of $\boldsymbol{y}_k$. 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

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

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 $(\boldsymbol{c}, \boldsymbol{y})$, where $\boldsymbol{c}$ is a configuration of $\mathcal{A}_1$. On an input word $\boldsymbol{w} = (\sigma_1, d_1), \ldots, (\sigma_k, d_k)$, let the sequence of configurations be $(\boldsymbol{c}_0, \boldsymbol{y}_0)$, $(\boldsymbol{c}_1, \boldsymbol{y}_1)$, …, $(\boldsymbol{c}_k, \boldsymbol{y}_k)$, so the output of $\mathcal{A}$ is $\boldsymbol{y}_k$. Then the least-fixed-point semantics of Equation (1) implies that, for $i = 1, \ldots, k$, $\boldsymbol{y}_i$ is the least vector satisfying $\boldsymbol{y}_i = \left(\Delta_{\sigma_i, \boldsymbol{y}_i}(\boldsymbol{c}_{i-1}, d_i)\right)|_{F_1}$. Similarly, for $i = 0$, $\boldsymbol{y}_0$ is the least vector satisfying $\boldsymbol{y}_0 = \left(\Delta_{\mathrm{i}}(\boldsymbol{y}_0)\right)|_{F_1}$. Now we want to show by induction that $\boldsymbol{c}_i$ simulates the list, over all possible splits of $\boldsymbol{w} = \boldsymbol{w}_1 \boldsymbol{w}_2 \cdots \boldsymbol{w}_k$, 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 $\boldsymbol{y}_i = \left(\Delta_{\sigma_i, \boldsymbol{y}_i}(\boldsymbol{c}_{i-1}, d_i)\right)|_{F_1}$ and decompose the configuration $\Delta_{\sigma_i, \boldsymbol{y}_i}(\boldsymbol{c}_{i-1}, d_i)$ using the simulation relation, and see that it simulates the list of all splits $\boldsymbol{w} = \boldsymbol{w}_1 \cdots \boldsymbol{w}_k$ where $\boldsymbol{w}_k$ has size at least 1, plus the additional initialized thread $\Delta_{\mathrm{i}}(\boldsymbol{y}_i)$.

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

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

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 3.2. *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 $(\boldsymbol{c}_1, \boldsymbol{c}_2)$, and we define $(\boldsymbol{c}_1, \boldsymbol{c}_2) \sim [(\boldsymbol{c}_{1,1}, \boldsymbol{c}_{2,1}), \ldots, (\boldsymbol{c}_{1,k}, \boldsymbol{c}_{2,k})]$ if *both* $\boldsymbol{c}_1 \sim [\boldsymbol{c}_{1,1}, \ldots, \boldsymbol{c}_{1,k}]$ and $\boldsymbol{c}_2 \sim [\boldsymbol{c}_{2,1}, \ldots, \boldsymbol{c}_{2,k}]$. For prefix sum, the restartability holds for somewhat trivial reasons: if we restart with only $\vec{\perp}$, the output is $\perp$: if we restart with only one non-$\vec{\perp}$ thread, the output is the prefix-sum, and if we restart with two or more non-$\vec{\perp}$ threads, the output is $\top$ everywhere. For concatenation, we have configurations that are pairs $(\boldsymbol{c}_1, \boldsymbol{c}_2)$ of a configuration in $\mathcal{A}_1$ and one in $\mathcal{A}_2$. We define $(\boldsymbol{c}_1, \boldsymbol{c}_2) \sim [(\boldsymbol{c}_{1,1}, \boldsymbol{c}_{2,1}), \ldots, (\boldsymbol{c}_{1,k}, \boldsymbol{c}_{2,k})]$ if $\boldsymbol{c}_1 \sim [\boldsymbol{c}_{1,1}, \ldots, \boldsymbol{c}_{1,k}]$ and *there exists* sequences $l_{2,1}$, $l_{2,2}$, …, $l_{2,k}$, such that $\boldsymbol{c}_{2,i}$ simulates $l_{2,i}$ and $\boldsymbol{c}_2$ 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 $\boldsymbol{c}_i \sim [\boldsymbol{c}_{i,1}, \ldots, \boldsymbol{c}_{i,k}]$ for each i, if $\boldsymbol{c} \sim [\boldsymbol{c}_1, \ldots, \boldsymbol{c}_k]$ then $\boldsymbol{c} \sim [\boldsymbol{c}_{i,j}]_{i,j}$. □

Theorem 3.3. *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 2.2, a DT over $(\mathbb{U}, \mathrm{UOp})$ where $\mathbb{U} = \{\star\}$. 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 $\mathbb{U}$; 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 $\Delta_{\sigma,x}$. 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 §2.1). 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 $(p', [p'_1, p'_2, \ldots, p'_{k-1}])$; 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 $(p, [p_1, \ldots, p_k])$ is constant: three configurations of $\mathcal{P}$. Overall, we guess a sequence of moves to get to $(p', [p'_1, \ldots, p'_{k-1}]$, 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 $\bot$ 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 5.1 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 3.2 guarantees that such exponential blowup does not arise in the compilation of the QRE-Past language of §4.

4 PROPOSED MONITORING LANGUAGE: QRE-PAST

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 [Mamouras et al. 2017]. 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 §3.

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 [Mamouras et al. 2017], 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.

4.1 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 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 [Alur et al. 2016]. 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 [Manna and Pnueli 2012] and informed by successful existing work on monitoring of safety properties [Havelund and Roşu 2004], 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 $\leq, \geq, = \in \mathrm{Op}_2$: these are comparison operations on data values returning 0 or 1. We also assume that we have Boolean operators $\neg \in \mathrm{Op}_1$ and $\wedge, \vee, \rightarrow, \leftrightarrow \in \mathrm{Op}_2$, 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 §3 apply.

4.2 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 $[\![\mathcal{A}_\alpha]\!](x, \boldsymbol{w})$ is the value of query α on input $\boldsymbol{w}$, 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: $[\![\mathcal{I}_\alpha]\!](x, \boldsymbol{w}) = x$ if $[\![\mathcal{A}_\alpha]\!](x, \boldsymbol{w}) \in \mathbb{D}$, $\top$ if $[\![\mathcal{A}_\alpha]\!](x, \boldsymbol{w}) = \top$, and $\bot$ if $[\![\mathcal{A}_\alpha]\!](x, \boldsymbol{w}) = \bot$. In particular, $\mathcal{I}_\alpha$ is *equivalent* to $\mathcal{A}_\alpha$ (definition in §2.5). 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

$\alpha :=$	$\mathtt{atom}(\sigma, t)$	$\{\sigma\}$	$\sigma \in \Sigma, t \in \mathsf{Tm}[\mathsf{cur}]$
	$\mathtt{eps}(t)$	$\{\varepsilon\}$	$t \in \mathsf{Tm}[\varnothing]$
	$\mathtt{or}(\alpha_1, \alpha_2)$	$\overline{L}(\alpha_1) \cup \overline{L}(\alpha_2)$	
	$\mathtt{split}(\alpha_1, \alpha_2, op)$	$\overline{L}(\alpha_1) \cdot \overline{L}(\alpha_2)$	$op \in \mathrm{Op}_2$
	$\mathtt{iter}(\alpha_1, init, op)$	$(\overline{L}(\alpha_1))^*$	$init \in \mathbb{D}, op \in \mathrm{Op}_2$
	$\mathtt{combine}(\alpha_1, \ldots, \alpha_k, op)$	$\overline{L}(\alpha_1) \cap \cdots \cap \overline{L}(\alpha_k)$	$op \in \mathrm{Op}_k$; well-typed if $\overline{L}(\alpha_1) = \cdots = \overline{L}(\alpha_k)$
	$\mathtt{prefix\text{-}sum}(\alpha_1, init, op)$	Σ^*	$init \in \mathbb{D}, op \in \mathrm{Op}_2$; well-typed if $L(\alpha_1) = \Sigma^*$
$\beta :=$	α_1	$\overline{L}(\alpha_1)$	
	$\mathtt{fill}(\alpha_1)$	$L(\alpha_1) \cdot \Sigma^*$	
	$\mathtt{fill\text{-}with}(\alpha_1, \alpha_2)$	$L(\alpha_1) \cup \overline{L}(\alpha_2)$	
$\varphi :=$	$\beta_1\ comp\ \beta_2$	Σ^*	$comp \in \{\leq, \geq, =\}$; well-typed if $L(\beta_1) = L(\beta_2) = \Sigma^*$
	$\varphi_1\ bop\ \varphi_2 \quad \mid \neg\varphi_1$	Σ^*	$bop \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$
	$\mid \odot\varphi_1 \quad \mid \boxdot\varphi_1 \quad \mid \diamond\varphi_1$	Σ^*	
	$\mid \varphi_1\ \mathcal{S}_w\ \varphi_2 \quad \mid \varphi_1\ \mathcal{S}_s\ \varphi_2$	Σ^*	

Fig. 6. 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.

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: atom, eps. The atomic expressions are the building blocks of all queries. For $t \in \mathsf{Tm}[\mathsf{cur}]$, the query $\mathtt{atom}(\sigma, t)$ matches a data word containing a single character (σ, d), and returns t evaluated with $\mathsf{cur} = d$. Similarly, the query $\mathtt{eps}(t)$ 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 $F = \{q_f\}$. $\mathcal{A}_{\mathtt{atom}(\sigma,t)}$ uses one transition from $\{q_i\}$ to q'_f with term t, and $\mathcal{A}_{\mathtt{eps}(t)}$ 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 3.1 (alternatively, if they aren't, just convert to an equivalent restartable DT as in §3.2, last paragraph). The definition of $\mathcal{I}_{\mathtt{atom}(\sigma,t)}$ is the same as $\mathcal{A}_{\mathtt{atom}(\sigma,t)}$ except that the term t in the transition is replaced by q_i; and likewise for $\mathcal{I}_{\mathtt{eps}(t)}$.

Regular operators: or, split, 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 $\mathtt{or}(\alpha_1, \alpha_2)$ 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 3.2. Similarly, we can take $\mathcal{I}_{\mathtt{or}(\alpha_1,\alpha_2)} = \mathcal{I}_{\alpha_1} \sqcup \mathcal{I}_{\alpha_2}$. Both of these constructions add only a constant to the size.

The operation $\mathtt{split}(\alpha_1, \alpha_2, op)$ splits a data word $\boldsymbol{w}$ into two parts, $\boldsymbol{w}_1 \cdot \boldsymbol{w}_2$, such that $\boldsymbol{w}_1$ matches α_1 and $\boldsymbol{w}_2$ matches α_2. If there are multiple splits, the result is $\top$; otherwise, the result is $op(\alpha_1(\boldsymbol{w}_1), \alpha_2(\boldsymbol{w}_2))$. Here, we have to do some work to save the value of $\alpha_1(\boldsymbol{w}_1)$ in the DT construction. We implement split as $\mathcal{A}_{\mathtt{split}(\alpha_1,\alpha_2,op)} := (\mathcal{A}_{\alpha_1} \cdot (\mathcal{I}_{\alpha_2} \parallel \mathcal{A}_{\alpha_2})) \cdot G_{op}$, 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 $\boldsymbol{w}$ into two parts $\boldsymbol{w}_1 \cdot \boldsymbol{w}_2$ such that $\boldsymbol{w}_i \in \overline{L}(\alpha_i)$, 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 $\boldsymbol{w}$ into $\boldsymbol{w}_1 \cdots \boldsymbol{w}_k$ such that $\boldsymbol{w}_i \in \overline{L}(\alpha_1)$ 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 $op(op(op(init, \alpha_1(\boldsymbol{w}_1)), \alpha_1(\boldsymbol{w}_2)), \alpha_1(\boldsymbol{w}_3))$. If the parsing is not unique, the result is $\top$. We implement this as $\mathcal{A}_{\texttt{iter}(\alpha_1, init, op)} := G_{init} \cdot \left((\mathcal{I}_{\alpha_1} \parallel \mathcal{A}_{\alpha_1}) \cdot G_{op}\right)^*$, 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 $\mathcal{A}_{\texttt{combine}(\alpha_1, \ldots, \alpha_k, op)} := \left(\mathcal{A}_{\alpha_1} \parallel \cdots \parallel \mathcal{A}_{\alpha_k}\right) \cdot G_{op}$, 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.

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

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 $\text{fst}, \text{snd} : \mathbb{D}^2 \to \mathbb{D}$ be the first and second projection operations. Then we implement the fill operations as:

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

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 $\texttt{combine}(\beta_1, \beta_2, \textit{comp})$. We do not need to worry about restartability for temporal queries, and we also don't define $\mathcal{I}$.

Boolean operators: $\wedge, \vee, \to, \leftrightarrow, \neg$. 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: $\odot, \boxdot, \diamondsuit, \mathcal{S}_w, \mathcal{S}_s$. 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 $\diamondsuit(\varphi_1)$ 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 $\diamondsuit(\varphi_1)$; the other cases are similar. Define a data function G which computes the truth value of $\diamondsuit(\varphi_1)$ on input $\boldsymbol{w}(\sigma, d)$ given its truth value on $\boldsymbol{w}$ and given the truth value of φ_1 on input $\boldsymbol{w}(\sigma, d)$ (so, G is just disjunction). Define $\mathcal{A}_{\diamondsuit(\varphi_1)} := G_{0,0} \cdot \oplus_G \mathcal{A}_{\varphi_1}$, 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 2.1, 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 4.1. *For every well-typed base-level quantitative query α, the compilation described above via the constructions of §3 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.*

4.3 Case Study: Cardiac Arrhythmia Detection

We will present now an application of the QRE-Past monitoring language to the medical domain, in particular for the detection of potentially fatal cardiac arrhythmias. Specifically, we consider Implantable Cardioverter Defibrillators (ICDs), which are devices that continually monitor the electrical activity of the heart in order to detect potentially fatal arrhythmias. If a dangerous arrhythmia is detected, then the ICD delivers a powerful electrical shock to the patient in order to restore the normal rhythm of the heart.

The type of fatal arrhythmia that ICDs detect is called *Ventricular Tachycardia* (VT), and its identification is based on the electrical signal that the device records directly from the right ventricle (lower-right chamber of the heart). As a first step, the heart signal is analyzed by the device to detect its *peaks*, which correspond to the ventricular heart beats (contractions). We will assume here that the input signal already includes the information about peaks. More specifically, we suppose that the characters of the input stream are of the form (b, seq), where b is a Boolean value, and seq is a sequence number. The presence (resp., absence) of a heartbeat is indicated with $b = 1$ (resp., $b = 0$). The signal is uniformly sampled and the sampling period is T. So, the timestamp of (b, seq) is equal to $seq \cdot T$.

The arrhythmia detection algorithm is based on the length of the heartbeat intervals, and uses several criteria that are called *discriminators* in the medical literature:

- *Initial Rhythm Classification* (IRC): The current interval length and the average of the four most recent interval lengths are both below a threshold T_{IRC}.
- *Sudden Onset* (SO): This discriminator corresponds to the clinical observation that VT typically occurs suddenly. It quantifies the suddenness of tachycardia using the last nine interval lengths $I_1, \ldots, I_9$ (where I_9 is the last interval length). The onset of tachycardia is considered to be sudden if at least one of the differences $I_1 - I_9, \ldots, I_8 - I_9$ is greater than a threshold T_{SO}.
- *Rhythm Stability* (RS): The heartbeat interval lengths during VT usually display low variability. The rhythm stability discriminator quantifies variability as the difference between the second longest and the second shortest interval lengths among the last ten heartbeat intervals. If this difference is less than a threshold T_{RS}, then the rate is considered to be stable.

At the top level, the query for VT detection is a Boolean combination (in particular, a conjunction) of the three above discriminators. Each discriminator can be described modularly by specifying a computation that processes a single heartbeat interval, and then composing these subcomputations sequentially a fixed or an arbitrary number of times. This high-level structure of the VT detection algorithm suggests the need for a language that combines Boolean operators with sequence-based pattern matching and quantitative aggregation. The QRE-Past language provides all these features, and therefore it facilitates the modular description of the VT detection algorithm. The DT framework then provides the constructs for compiling the high-level description into an efficient streaming algorithm with precise bounds on space and per-element time usage: the complexity (for both resources) is constant in the length of the stream and quadratic in the size of the specification.

We start by describing a query that computes the length of a single heartbeat interval. The pattern is $0^+ \cdot 1$, which describes an arbitrary number of samples that correspond to the absence of a heartbeat followed by a single heartbeat. The query f_{0^+1} (shown below) matches a heartbeat interval and outputs its length.

$$
\begin{aligned}
\mathrm{f}_0 &= \texttt{atom}(0, seq) && \text{// rate } 0\\
\mathrm{f}_{0^*} &= \texttt{iter}(\mathrm{f}_0, 0, (x, y) \to 0) && \text{// rate } 0^*\\
\mathrm{f}_{0^+} &= \texttt{split}(\mathrm{f}_0, \mathrm{f}_{0^*}, (x, y) \to x) && \text{// rate } 0^+\\
\mathrm{f}_1 &= \texttt{atom}(1, seq) && \text{// rate } 1
\end{aligned}
$$

$$\mathsf{f}_{0^+1} = \texttt{split}(\mathsf{f}_{0^+}, \mathsf{f}_1, (x, y) \to |y - x| \cdot T) \quad \text{// rate } 0^+1$$

Now, the query f_{last} (shown below) outputs the length of the last interval:

$$\mathsf{f}_{\text{prev}} = \texttt{iter}(\mathsf{f}_{0^+1}, 0, (x, y) \to 0) \qquad \text{// rate } (0^+1)^*$$
$$\mathsf{f}_{\text{last}} = \texttt{split}(\mathsf{f}_{\text{prev}}, \mathsf{f}_{0^+1}, (x, y) \to y) \qquad \text{// rate } (0^+1)^+$$

W.l.o.g. we can assume that the input stream always starts with a 0-tagged value (absence of heartbeat). Now, the average of the last four intervals is given by the query

$$\mathsf{f}_{\text{last4}} = \texttt{split}(\mathsf{f}_{\text{prev}}, \mathsf{f}_{0^+1}, \mathsf{f}_{0^+1}, \mathsf{f}_{0^+1}, \mathsf{f}_{0^+1}, (x, y_1, y_2, y_3, y_4) \to (y_1 + y_2 + y_3 + y_4)/4)$$

with rate $(0^+1)^* \cdot (0^+1)^4$. The satisfaction of the IRC discriminator is described by the conjunction, denoted φ_{IRC} of the following two QRE-Past formulas:

$$\texttt{fill-with}(\mathsf{f}_{\text{last}}, T_{\text{IRC}}) < T_{\text{IRC}} \qquad\qquad \texttt{fill-with}(\mathsf{f}_{\text{last4}}, T_{\text{IRC}}) < T_{\text{IRC}}$$

Notice that this conjunction can only be satisfied at the occurrence of a heartbeat. Since the discriminators SO and RS are similar to IRC in that they are computed using a fixed number of the most recent heartbeat intervals, we leave it as an exercise to the reader to write the formulas φ_{SO} and φ_{RS}. Finally, detection of VT is given by the formula $\varphi_{\text{IRC}} \wedge \varphi_{\text{SO}} \wedge \varphi_{\text{RS}}$.

Using QRE-Past instead of the QRE (or StreamQRE) language for this application has the advantage that it directly allows the use of tests on aggregates. In [Abbas et al. 2018] this behavior was encoded by annotating the stream with additional information, streaming the intermediate output to another processing stage (using the operation of streaming composition), and then applying tests on the annotated data values.

Temporal operators for signal invariants. The arrhythmia detection algorithm described in this case study is based on standard techniques employed by major manufacturers of medical devices (see, for example, [Zdarek and Israel 2016]). The discriminators IRC, SO and RS can be expressed easily without the use of past-time temporal operators. There are, however, important invariants about the cardiac signal that can be conveniently expressed using the "always in the past" temporal operator. For example, consider the temporal formula

$$\phi = \boxdot(T_{\text{short}} < \texttt{fill-with}(\texttt{iter}(\mathsf{f}_{0^+1}, T_{\text{normal}}, (x, y) \to y), T_{\text{normal}}) < T_{\text{long}}),$$

where T_{short}, T_{normal} and T_{long} are constants that correspond to short, normal and long heart interval lengths respectively, and $c < \mathsf{f} < d$ is abbreviation for the conjunction $(c < \mathsf{f}) \wedge (\mathsf{f} < d)$. The formula ϕ asserts the invariant that every heartbeat interval is always within physically realizable bounds.

5 SUCCINCTNESS AND EXPRESSIVENESS

5.1 Comparison with Cost Register Automata

Cost register automata (CRAs) were introduced in [Alur et al. 2013] 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 [Alur et al. 2018], 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 5.1. *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 5.2. *For some* $(\mathbb{D}, Op)$*, (restartable) DTs can be exponentially more succinct than CRAs.*

Proof sketch. Let $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$, $\mathbb{D} = \mathbb{N}$, and $\mathrm{Op} = \{+\}$ (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. □

5.2 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 [Chandra et al. 1981; Salomaa et al. 2000] 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 [Chandra et al. 1981; Fellah et al. 1990] 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).

5.3 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 Stream(Op) to denote the class of such efficient streaming algorithms. The problem with the class Stream(Op) is that it is not suitable for modular specifications. As the following theorem shows, it is not closed under the `split` combinator.

Theorem 5.3. 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:

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

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

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 Stream(Op)), but $h = \texttt{split}(f, g, (x, y) \texttt{->} 1)$ 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 Stream(Op), however, employs a finite number of integer registers whose size (in bits) can grow only logarithmically. □

Theorem 5.3 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 5.4. A streaming algorithm of Stream(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 Stream(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 RELATED WORK

6.1 Quantitative Automata

The literature contains various proposals of automata-based models that are some kind of quantitative extension of classical finite-state automata.

Weighted automata, which were introduced in [Schützenberger 1961] (see also the more recent monograph [Droste et al. 2009]), extend nondeterministic finite-state automata by annotating transitions with *weights* (which are elements of a semiring) and can be used for the computation of simple quantitative properties. A weighted automaton maps an input string w to the minimum over costs of all accepting paths of the automaton over w. Extensions such as *nested weighted automata* [Chatterjee et al. 2015] enjoy increased expressiveness, but fall short of capturing an arbitrary set of data types and operations as CRAs and DTs do. We recently studied arbitrary hierarchical nesting of weighted automata in [Alur et al. 2017], which does allow arbitrary types and operations. We showed that under certain typing restrictions there is a streaming evaluation algorithm. In contrast, here we introduce a model that admits streaming evaluation *sans* typing restrictions; which is "flat", i.e. not recursively defined; such that the transition structure makes modular composition feasible; and for which we have clean succinctness results.

Another approach to augment classical automata with quantitative features has been with the addition of *registers* that can store values from a potentially infinite set. These models are typically varied in two aspects: by the choice of data types and operations that are allowed for register manipulation, and by the ability to perform tests on the registers for control flow.

The literature on data words, data/register automata and their associated logics [Björklund and Schwentick 2010; Bojańczyk et al. 2011; Demri and Lazić 2009; Kaminski and Francez 1994; Neven et al. 2004] studies models that operate on words over an infinite alphabet, which is typically of the form $\Sigma \times \mathbb{N}$, where Σ is a finite set of tags and $\mathbb{N}$ is the set of the natural numbers. They allow comparing data values for equality, and these equality tests can affect the control flow. In DTs, tests on the data are not allowed to affect the underlying control flow, that is, whether each state variable is undefined, defined, or conflicted (see Theorem 2.2).

The work on cost register automata (CRAs) [Alur et al. 2013; Alur and Raghothaman 2013] and streaming transducers [Alur and Cerný 2010; Alur and D'Antoni 2012; Alur and Černý 2011] is about models where the control and data registers are kept separate by allowing write access to the registers but no testing. As discussed in §5.1, DTs are exponentially more succinct than CRAs. The exponential gap arises for the useful construction of performing several subcomputations in parallel and combining their results. DTs recognize the class of *streamable regular transductions*, which is equivalently defined by CRAs and attribute grammars [Alur et al. 2018].

The recent work [Bojańczyk et al. 2018] gives a characterization of the first-order definable and MSO-definable string-to-string transformations using algebras of functions that operate on objects such as lists, lists of lists, pairs of lists, lists of pairs of lists, and so on. Monitors with finite-state control and unbounded integer registers are studied in [Ferrère et al. 2018] and a hierarchy of expressiveness is established on the basis of the number of available registers. These papers focus on issues related to expressiveness, whereas we focus here on modularity and succinctness.

6.2 Query Languages for Runtime Monitoring

Runtime monitoring (see the survey [Leucker and Schallhart 2009]) is a lightweight verification technique for testing whether a finite execution trace of a system satisfies a given specification. The specification is translated into a *monitor*, which executes along with the monitored system: it consumes system events in a streaming manner and outputs the satisfaction or falsification of the specification. A widely used formalism for describing specifications for monitoring is *Linear*

Temporal Logic (LTL) [Havelund and Roşu 2004]. Metric Temporal Logic (MTL) has been used for monitoring real-time temporal properties [Thati and Roşu 2005]. Signal Temporal Logic (STL), which extends MTL with value comparisons, has been used for monitoring real-valued signals [Deshmukh et al. 2017]. Computing statistical aggregates of LTL-defined properties, as in [Finkbeiner et al. 2002], is a limited form of *quantitative* monitoring. The Eagle specification language [Barringer et al. 2004] can also express some quantitative monitoring properties, since it supports data-bindings.

The line of work on synchronous languages [Benveniste et al. 2003] also deals with processing data streams. The focus in the design of these languages is the decomposition of the computation into logically concurrent tasks. Here, we focus on the control structure for parsing the input stream and applying quantitative aggregators. Examples of synchronous languages designed for runtime monitoring include LOLA [d'Angelo et al. 2005] and its extensions [Bozzelli and Sánchez 2016].

7 CONCLUSIONS

Data transducers are a succinct, implementation-level machine model for general streaming computations. They combine finite control (active vs. inactive states) and register updates (e.g. in CRAs) into one integrated model, with *state variables* that can be undefined, defined, or conflicted. All DTs admit streaming evaluation (Theorem 2.1), but in order to additionally support modular constructions, we identified an equally-expressive subclass of *restartable DTs*. We showed that DTs admit succinct *union, concatenation, iteration, parallel composition, prefix-sum, and support* constructions (§3), where restartability is required for concatenation and iteration.

Although converting a DT to a restartable DT may involve exponential blowup, we showed that because the constructions preserve restartability (Theorem 3.2), such blowup does not arise in the modular compilation of realistic queries using DTs. To illustrate this point, we proposed a query language called QRE-Past, which combines elements of StreamQRE [Mamouras et al. 2017] with past-time temporal logic operators [Havelund and Roşu 2004], such that all queries compile modularly to quadratic-size DTs via the succinct constructions.

We formally justified that DTs are exponentially more succinct over CRAs (§5.1), and that they relate to a certain kind of finite automata and are more succinct than DFAs and NFAs (§5.2). In fact, we showed that DTs are as succinct as general streaming computations, where the update function is specified using loop-free code, and a *regularity restriction* is enforced (§5.3). Without such a restriction, general streaming computations are not closed under concatenation (Theorem 5.3).

One notable operation missing from our constructions in §3 is that of *sequential composition*, in which we pass the sequence of outputs of one machine as the input stream to another. This operation is crucial to many applications, has been included in some previous presentations of QREs [Alur et al. 2016; Mamouras et al. 2017], and should be considered in future implementations of this work. We omitted it here because it introduces notational complexity: in order to define sequential composition, both the input and output streams need to be tagged (not just the input stream), and (at least) the final states of a DT need to be associated with output tags.

In future work, we hope to explore opportunities for query optimization using the DT model. The DT framework lends itself more easily to this purpose than QREs or unstructured streaming algorithms. In the case of NFAs, bisimulation relations can be used to reduce the size of the automata via a quotient construction, and it seems plausible that an analogous notion can be defined for DTs to reduce the number of variables. The Java implementation of the StreamQRE language (reported in [Mamouras et al. 2017]) does not currently use DTs, and could benefit from such optimizations.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive criticism. This research was supported in part by NSF award CCF 1763514 and by a Simons Investigator award.

REFERENCES

Houssam Abbas, Rajeev Alur, Konstantinos Mamouras, Rahul Mangharam, and Alena Rodionova. 2018. Real-time Decision Policies with Predictable Performance. *To appear in the Proceedings of the IEEE, Special Issue on Design Automation for Cyber-Physical Systems* (2018).

Rajeev Alur and Pavol Cerný. 2010. Expressiveness of Streaming String Transducers. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*, Vol. 8.

Rajeev Alur and Loris D'Antoni. 2012. Streaming Tree Transducers. In *39th International Colloquium on Automata, Languages, and Programming (ICALP '12)*.

Rajeev Alur, Loris D'Antoni, Jyotirmoy Deshmukh, Mukund Raghothaman, and Yifei Yuan. 2013. Regular Functions and Cost Register Automata. In *Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '13)*.

Rajeev Alur, Dana Fisman, Konstantinos Mamouras, Mukund Raghothaman, and Caleb Stanford. In submission, 2018. Streamable Regular Transductions. (In submission, 2018). arXiv:arXiv:1807.03865

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

Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford. 2017. Automata-Based Stream Processing. In *Proceedings of the 44th International Colloquium on Automata, Languages, and Programming (ICALP '17)*.

Rajeev Alur and Mukund Raghothaman. 2013. Decision Problems for Additive Regular Functions. In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP '13)*.

Rajeev Alur and Pavol Černý. 2011. Streaming Transducers for Algorithmic Verification of Single-pass List-processing Programs. In *Proceedings of the 38th Annual Symposium on Principles of Programming Languages (POPL '11)*.

Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. 2004. Rule-based runtime verification. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*.

Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The Synchronous Languages 12 Years Later. *Proc. IEEE* 91, 1 (2003).

Henrik Björklund and Thomas Schwentick. 2010. On Notions of Regularity for Data Languages. *Theoretical Computer Science* 411, 4 (2010).

Mikołaj Bojańczyk, Laure Daviaud, and Shankara Narayanan Krishna. 2018. Regular and First-Order List Functions. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*.

Mikołaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. 2011. Two-variable Logic on Data Words. *ACM Transactions on Computational Logic (TOCL)* 12, 4 (2011).

Patricia Bouyer, Antoine Petit, and Denis Thérien. 2003. An algebraic approach to data languages and timed languages. *Information and Computation* 182, 2 (2003).

Laura Bozzelli and César Sánchez. 2016. Foundations of Boolean stream runtime verification. *Theoretical Computer Science* 631 (2016).

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

Krishnendu Chatterjee, Thomas A. Henzinger, and Jan Otop. 2015. Nested Weighted Automata. In *Proceedings of the 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '15)*.

Bruno Courcelle. 1994. Monadic Second-Order Definable Graph Transductions: A Survey. *Theoretical Computer Science* 126, 1 (1994).

Ben d'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. LOLA: Runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME 2005)*.

Stéphane Demri and Ranko Lazić. 2009. LTL with the Freeze Quantifier and Register Automata. *ACM Transactions on Computational Logic (TOCL)* 10, 3 (2009).

Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. 2017. Robust online monitoring of signal temporal logic. *Formal Methods in System Design* 51, 1 (2017).

Manfred Droste, Werner Kuich, and Heiko Vogler (Eds.). 2009. *Handbook of Weighted Automata*.

Joost Engelfriet and Sebastian Maneth. 1999. Macro Tree Transducers, Attribute Grammars, and MSO Definable Tree Translations. *Information and Computation* 154, 1 (1999).

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

Thomas Ferrère, Thomas A. Henzinger, and N. Ege Saraç. 2018. A Theory of Register Monitors. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*.

Bernd Finkbeiner, Sriram Sankaranarayanan, and Henny Sipma. 2002. Collecting statistics over runtime executions. *Electronic Notes in Theoretical Computer Science* 70, 4 (2002).

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

Michael Kaminski and Nissim Francez. 1994. Finite-memory Automata. *Theoretical Computer Science* 134, 2 (1994).

Martin Leucker and Christian Schallhart. 2009. A Brief Account of Runtime Verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009). The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G. Ives, and Sanjeev Khanna. 2017. 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)*.

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

Shanmugavelayutham Muthukrishnan. 2005. Data Streams: Algorithms and Applications. *Foundations and Trends® in Theoretical Computer Science* 1, 2 (2005).

Frank Neven, Thomas Schwentick, and Victor Vianu. 2004. Finite State Machines for Strings over Infinite Alphabets. *ACM Transactions on Computational Logic (TOCL)* 5, 3 (2004).

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

Marcel Paul Schützenberger. 1961. On the Definition of a Family of Automata. *Information and control* 4, 2 (1961).

Prasanna Thati and Grigore Roşu. 2005. Monitoring Algorithms for Metric Temporal Logic Specifications. *Electronic Notes in Theoretical Computer Science* 113 (2005). Proceedings of the Fourth Workshop on Runtime Verification (RV 2004).

Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. 2017. Quantitative Network Monitoring with NetQRE. *ACM SIGCOMM Conference on Data Communication* (2017).

Jan Zdarek and Carsten W. Israel. 2016. Detection and Discrimination of Tachycardia in ICDs Manufactured by St. Jude Medical. *Herzschrittmachertherapie + Elektrophysiologie* 27, 3 (2016).