INVITED PAPER

Real-Time Decision Policies With Predictable Performance

This paper introduces the usage of declarative streaming languages, in particular StreamQRE, for modeling and analyzing real-time streaming applications. The approach is based on the formalism of quantitative regular expressions. It can guarantee constant memory, runtime, and energy cost per data item, and can calculate the upper bounds on the per-item cost.

By Houssam Abbas, *Member IEEE*, Rajeev Alur, *Fellow IEEE*, Konstantinos Mamouras, *Member IEEE*, Rahul Mangharam, *Member IEEE*, and Alena Rodionova, *Member IEEE*

ABSTRACT | As methods and tools for cyber–physical systems (CPS) grow in capabilities and use, one-size-fits-all solutions start to show their limitations. In particular, tools and languages for programming an algorithm or modeling a CPS that are specific to the application domain are typically more usable, and yield better performance, than general-purpose languages and tools. In the domain of cardiac arrhythmia monitoring, a small, implantable medical device continuously monitors the patient's cardiac rhythm and delivers electrical therapy when needed. The algorithms executed by these devices are *streaming algorithms*, so they are best programmed in a streaming language that allows the programmer to reason about the incoming data stream as the basic object, rather than force her to think about lower-level details like state maintenance and minimization. Because these devices are resource-constrained, it is useful if the programming language allowed predictable performance in terms of processing runtime and energy consumption, or more general costs. StreamQRE is a declarative streaming programming language, with an efficient and portable implementation and strong theoretical guarantees. In particular, its evaluation algorithm guarantees constant cost (runtime, memory, energy) per data item and also calculates upper bounds on the per-item cost. Such an estimate of the cost allows early exploration of the algorithmic possibilities, while maintaining a handle on worst case performance, on the basis of which hardware can be designed and algorithms can be tuned.

KEYWORDS | Arrhythmia monitoring; quantitative regular expressions; real time; streaming languages; Tachycardia

I. INTRODUCTION

The last few years have witnessed an explosion of Internet of Things (IoT) systems in applications such as smart buildings, wearable devices, and healthcare. A key component of an effective IoT system is the ability to make decisions in real time in response to data it receives. For instance, a gateway router in a smart home should detect and respond in a timely manner to security threats based on monitored network traffic, and a healthcare system should issue alerts in real time based on measurements collected from all the devices for all the monitored patients. Programming the desired logic as a deployable implementation is challenging due to the volume of data and hard constraints on available memory, power usage, and response time.

In current practice, a general-purpose imperative language such as C is used to program real-time decision making policies. Due to the challenges in analyzing such code, this approach does not lead to *predictable* performance and does not facilitate *exploration of design options* at early stages. A specialized language for specifying these policies in a declarative manner, with programming abstractions suitable for processing data streams with performance guarantees, can be a potential solution to both these challenges. It can play the same role as *model-based design* does for safety-critical embedded control software [1]–[4].

To specify the decision logic based on computing quantitative summaries of data streams we advocate quantitative

Manuscript received July 4, 2017; revised January 31, 2018 and June 4, 2018; accepted June 25, 2018. Date of publication August 7, 2018; date of current version September 14, 2018. *(Corresponding author: Houssam Abbas.)*

H. Abbas, R. Mangharam, and **A. Rodionova** are with the Department of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, PA 19104 USA (e-mail: houssamyma@gmail.com).

R. Alur and **K. Mamouras** are with the Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104 USA.

Digital Object Identifier 10.1109/JPROC.2018.2853608

regular expressions (QREs) [5], [6]. The language allows the computation to be expressed as a streaming composition of stages. The core QRE combinators, which are quantitative extensions of operations in classical regular expressions, can be used to impart to the input data stream a logical hierarchical structure facilitating modular specifications (for instance, to view patient data as a sequence of episodes and to view network traffic as a sequence of voice-over-IP sessions). The QRE compiler translates a high-level query into a streaming algorithm with precise complexity bounds on per-item processing time and total memory footprint. The StreamQRE library, an implementation in Java, has been shown experimentally to have superior performance compared to other existing high-performance engines for processing streaming data [6]. This experimental evaluation involved workloads that are representative of clickstream analysis (Yahoo streaming benchmark [7]) and real-time analytics for business event streams (NEXMark benchmark [8]). A variant of StreamQRE (called NetQRE) has been shown to be useful for network monitoring [9].

Medical devices offer an ideal testbed for exploring the applications of formal methods in system design due to their safety-critical nature that demands predictable operation [10]. Recently, the implantable pacemaker has been used to illustrate the benefits of model-based design [11]–[13]. This involves specifying the algorithms for detecting slower-than-normal rhythms used by pacemakers using formal modeling languages, such as timed automata [14] and hybrid automata [15], and verifying correctness requirements using a model checker such as UPPAAL [16].

While this previous work dealt with pacemakers, implantable cardioverter defibrillators (ICDs) and insertable loop recorders (ILRs) are a more sophisticated class of implantable cardiac devices that must do multibeat rhythm classification, not only detect whether a beat was missing, like pacemakers do. The goal of such an arrhythmia monitoring algorithm (AMA) is to detect undesirable patterns in the (discretized) input signal being monitored. We argue that such a classification task is best viewed as a matching algorithm over streaming data, and the desired decision logic can be naturally expressed using QREs.

In particular, we program a representative AMA, used in an ICD by Boston Scientific [17], using the QRE language. The QRE compiler then generates the low-level implementation whose space complexity and per-item processing time complexity are constant — that is, independent of the number of samples processed so far (see [6, Sec. 4]). Furthermore, we show how the QRE compiler can *statically* compute an upper bound on the cost of processing each item, where the cost can be, for example, the energy consumption on a specific platform. This assures predictable real-time performance. Such estimates, provided early in the design cycle, allow one to compare design alternatives (that is, different variants of the monitoring algorithm) statically in terms of their achievable worst case costs. Such analysis complements average-case analysis (i.e., measured performance when running the algorithm on a typical load). We demonstrate the latter type of analysis by profiling the energy consumption of the QRE on a signals database on a given hardware platform.

The paper is organized as follows. Section II gives a background on cardiac function, necessary for understanding the complexity of arrhythmia monitoring. Section III motivates the programming of AMAs in QREs, and Section IV introduces the QRE formalism and the Java library that implements it. This library is available online at [18]. Section V describes one representative AMA and Section VI details its QRE implementation. The Java library is used in Section VII to illustrate the implemented AMA on a database of arrhythmia episodes. Section VIII describes how to compute upper bounds on QRE cost, like per-item energy consumption. Section IX summarizes related work and Section X concludes the paper.

II. BACKGROUND ON CARDIAC FUNCTION

To understand the arrhythmia monitoring algorithm presented in this paper and appreciate its complexities, it is necessary to first understand some basics of cardiac electrophysiology: how the heart beats normally, why it could go into arrhythmia, and what measurements are available to an implantable device to detect this.

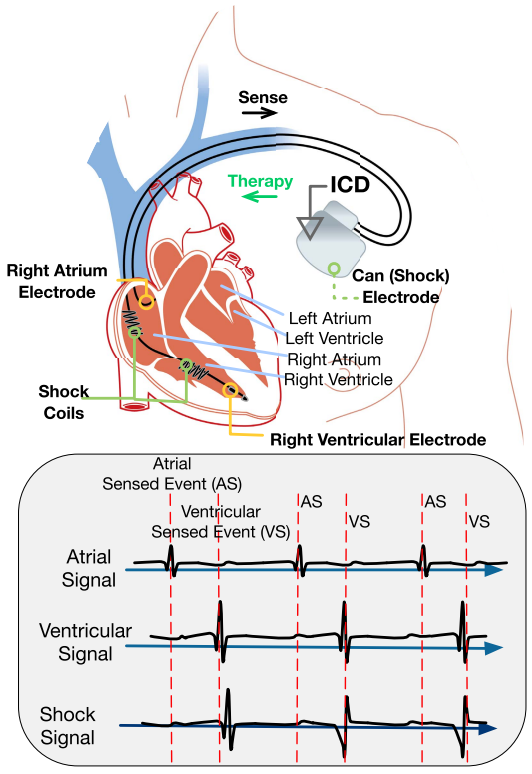

Fig. 1. *ICD and its connection to the heart.*

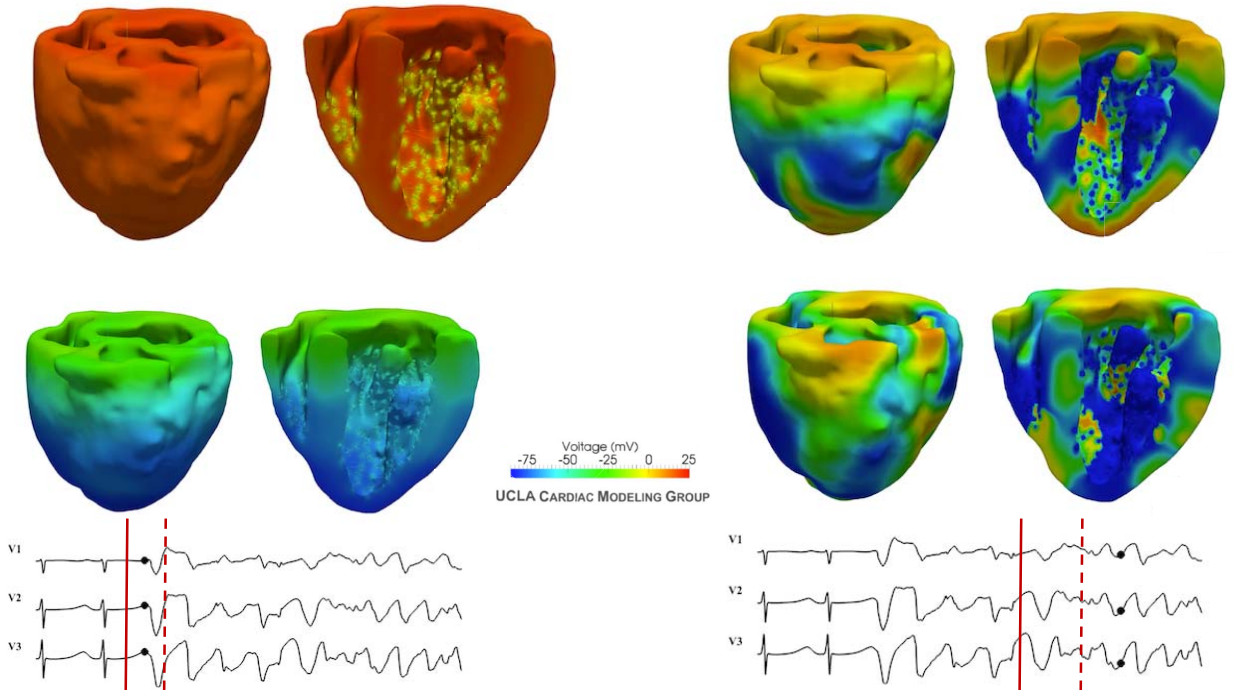

Fig. 2. *Electrical activity during normal sinus rhythm (NSR) and Ventricular Fibrillation (VF). The color scale runs from blue = rest state to red = excited (aka depolarized) state. In the top left, the ventricles are shown from two different angles, during a phase of NSR. The ventricles are fully excited. The bottom left panel shows a later phase of the same beat, where the ventricles are progressively relaxing, starting with the apex (the pointed tip of the heart). This orderly propagation ensures adequate muscle contraction and blood flow. Three surface ECGs are shown beneath the left column, with red bars indicating the timing of the two snapshots. Note the periodic pattern. The right column shows two snapshots during VF (earlier snapshot on top). Note the disorganized nature of the electrical activity, wavefront breakup, and the multiple regions of depolarization. Note also the change in the surface ECG from periodic and regular (early on) to disorganized. The AMA reads two such signals (obtained, however, intracardially and not from the surface) and tries to detect fibrillation. (Snapshots obtained from video of a simulation of the ventricles at UCLA, courtesy of Luigi Perotti [19].)*

A. Cardiac Electrophysiology

The heart has two upper chambers called the *atria* and two lower chambers called the *ventricles* (see Fig. 1). The synchronized contractions of atria and ventricles assure an adequate supply of oxygenated blood to the rest of the body. This contraction is driven by electrical activity in the heart, which originates in the right atrium, floods the atria first, then conducts down to the ventricles and floods those in turn. The cardiac muscle contracts as it is being traversed by the electrical wavefront, i.e., as it *depolarizes*. In a first approximation which is sufficient for understanding AMAs, we may consider that this contraction is an instantaneous event, and refer to it as an (atrial or ventricular) *beat*. This normal pattern of electrical activity is referred to as normal sinus rhythm (NSR), after the *sino-atrial node* where the electricity normally originates. Disturbances of NSR are referred to as *arrhythmias*. They can arise because of structural defects in the cardiac muscle, like a re-entrant circuit around which the electrical waveform circulates very fast, or because of irritable tissue that starts to depolarize faster than the sino-atrial node. Ventricular tachycardia (VT) is an example of an arrhythmia originating in the ventricles, in which the ventricles depolarize at a very high rate and effectively drive the rhythm. This high rate of depolarization does not give enough time for the muscle to contract and relax properly, which can result in insufficient blood supply. If the VT is sustained, or degenerates into Ventricular Fibrillation (VF) (Fig. 2), it is fatal within a minute. An abnormally fast heart rate that originates in the atria and/or the conduction system above the ventricles is referred to as a supra-ventricular tachycardia (SVT). An SVT causes patient discomfort but is not fatal in the short-term and does not require device treatment. Most fast arrhythmias fall under these two categories: VT or SVT.

B. Implantable Devices

Two types of implantable devices monitor a heart's rhythm continuously to detect abnormally fast arrhythmias, aka *tachycardias*. The first is implantable cardioverter defibrillators (ICDs). An ICD is inserted under the pectoral muscles, and has one or two leads that are directly implanted in the cardiac chambers, through which it measures local electrical activity; see Fig. 1. The measured signals are known as *electrograms*, or EGMs, and are termed "atrial" or "ventricular" depending on the chamber where they are measured.[1] See Fig. 3. An ICD uses EGMs to distinguish a wide range of tachycardias. If it detects a potentially fatal tachycardia, then it delivers therapy to the

[1]In this paper, we will ignore the so-called 'shock EGM' as it will not be used in describing arrhythmia monitoring algorithms.

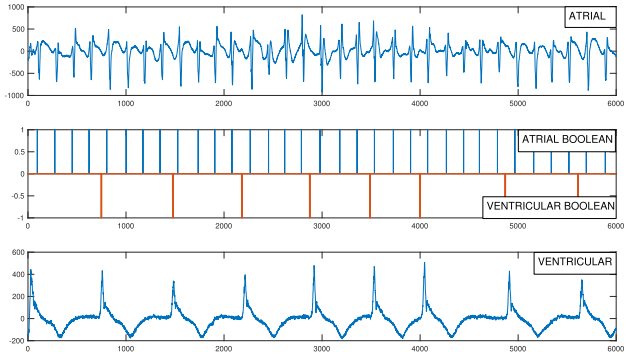

Fig. 3. *Electrograms (EGMs) (top and bottom panels) and corresponding Boolean beat signals (middle) during atrial tachycardia. Beats correspond to peaks in the EGMs.*

heart in the form of either low-energy pacing sequences or (possibly more than one) very high-energy shock. Either way, the goal of the therapy is to stop the current rhythm and allow a normal rhythm to start. VTs and SVTs can share similar heart rates and other characteristics, so an SVT can be misdiagnosed as a VT. This is problematic because shock therapy used to stop a VT can deliver between 30–60 Joules of energy at around 700 Volts in under 15ms [20], directly to the heart, which is very painful to the patient,[2] and has been shown to increase morbidity [21]. Therefore, one of the biggest challenges for ICDs is to discriminate between VF and sustained VT that typically requires a shock, and SVT that typically should not be shocked [22]. This paper will present one particular ICD AMA in detail in Section V.

The second type of device that monitors tachycardias is the insertable loop recorder (ILR) (also known as implantable cardiac monitor). An ILR is a small device (the smallest ILR on the market is smaller than a key) that is inserted subcutaneously, and monitors surface ECG signals. It uses these signals to compute a number of long- and short-term statistics of the rhythm, and in particular to detect atrial fibrillation (AF) episodes. AF is an abnormally fast and disorganized atrial rhythm that can lead to fainting spells, and which, in the long term, contributes to blood clot formation. These clots can cause a stroke upon reaching the brain. The ILR does not have any therapeutic functions, but only monitors the heart rhythm. As an example, Biotronik's BioMonitor [23] calculates and stores the following daily quantities, in a sliding window of 240 days where the oldest day drops out of the window. The quantities include: 1) the average daily heart rate; 2) the daily minimum average heart rate, where the averages are calculated over consecutive blocks of 10 min in the day; 3) daily heart rate variability, defined as the standard deviation of the sliding 5-min averages; and 4) the rate histogram, where each heartbeat is binned into bins of width 10 beats per-minute (bpm). In addition, the BioMonitor will take consecutive windows of n beats and count the number of cycle lengths that fall below a fixed value in each window.

Remote continuous monitoring has recently been shown to improve treatment outcomes [24] and to reduce time-to-treatment for patients with atrial tachycardia burden [25], so it is important to develop algorithms that can monitor over longer periods of time and/or compute more advanced statistics that can better detect the arrhythmia burden.

C. Device Measurement: From Real-Valued to Boolean Signal

Formally, an EGM is a uniformly sampled, discrete-time real-valued bounded signal. An EGM signal can be characterized by the *timing of beats* that produced it, and the *morphology of the signal itself*. To detect the beat timing (i.e., when the chamber is contracting), the peaks of the EGM are detected [26]. The output of peak detection is a discrete-time Boolean signal, where a 1 indicates a beat. See Fig. 3. Beat timing is crucial to an arrhythmia's detection, since it is used in all discriminators.

The "morphology" refers to the shape of the EGM. The so-called "shock" EGMs during an atrially driven rhythm look different from the shock EGMs during a ventricularly driven rhythm. The ICD uses this to help it determine whether the current arrhythmia is an SVT or VT. In this paper, and in order to keep the exposition simple, we will only work with the beat signal, i.e., the Boolean signal produced by peak detection on the local atrial and ventricular channels, as shown in Fig. 3.

III. STREAMING ALGORITHMS FOR ARRHYTHMIA DETECTION

An AMA is naturally viewed as *a pipeline of streaming algorithms*, where each node of the pipeline performs a streaming calculation on its input signal, and passes its output signal to the next node. So what is a streaming algorithm? And why view arrhythmia monitors as streaming algorithms? The main characteristics of a streaming algorithm are that it views its input as a sequence, or *stream*, of items from some data domain, arriving one at a time. It gets to process each item only once, after which it discards it and moves on to the next item in the input stream. After processing each item, the algorithm produces an output value (which might also be null). A streaming algorithm has limited memory available (much smaller than the length of the stream which, for practical purposes, may be regarded as infinite), and limited processing time. Section IV gives several examples of streaming calculations.

The following considerations, which govern the design and execution of an AMA, establish the suitability of the streaming model of calculation for AMA. First, an AMA's input is a uniformly sampled discrete-time electrical signal that arrives in real time, one sample at a time, and thus can be viewed as a stream. Second, when running on an

[2]Patients compare the shock to a "horse kicking you in the chest".

ICD, the AMA has a delay constraint. Namely, not much time must elapse between the onset of a fatal VT and the moment that the AMA detects it, because this delays the delivery of therapy. This requirement translates directly into a requirement of small processing time per item of the input signal, which is a key constraint on streaming algorithms. Third, ICDs and ILRs share a power consumption concern. Indeed, power is the *main* nonfunctional design factor for these devices. Even for today's ICDs, which can have a battery life between 7 and 11 years, an additional 3 months of battery life are still worth pursuing [27], since they can mean the difference between having to surgically replace the ICD or not. Because most ICD and ILR recipients are older patients with health complications [28], it is desirable to prolong battery life and reduce the likelihood of a replacement [27]. The power in an ICD is consumed by the monitoring algorithms, the shock therapy, and the pacing therapy. Although shocks are the single most power-hungry event, over an average device's lifetime, they will only consume 3% of the battery, and it is exceedingly rare that they consume more than 36% [29]. The rest is shared between pacing and monitoring. Thus, it is important to reduce the power cost of monitoring. For ILRs, because they do not have any therapeutic functions, most of the power is consumed by monitoring. Thus an AMA has a more general *small cost-per-item constraint*.

If AMAs are viewed as streaming algorithms, then it follows that they are best programmed using a *streaming programming language*. That is, a language that is expressly designed and optimized for describing streaming algorithms and automatically generating efficient code from the program description. Indeed, it is important to note the productivity gains achievable by using a domain specific language (DSL). It is generally agreed that programming in a DSL results in greater productivity for the development teams producing the software; see, e.g., [30] and [31], where development time reductions of 5–7x are routinely reported. During the *design exploration stage* when AMAs are developed, tweaked and compared, it is helpful to program in a language that allows high-level reasoning about the stream as the basic object of manipulation and easy capture of patterns in the stream.

The StreamQRE language [6], [18] permits such a declarative way of programming. StreamQRE (pronounced "stream query") allows the developer to create quantitative regular expressions (QREs), which are a quantitative extension of regular expressions. A QRE declares how the stream should be divided up (by matching against a regular expression) and which arbitrary operations should be executed on the matching pieces. Similarly to regular expressions, QREs can be combined using quantitative extensions of regular combinators to form more complex computations. QREs are described in detail in the next section. QREs also provide theoretical guarantees on the memory, time and energy consumed to process a data item by the resulting algorithm. Specifically, a QRE has per-item memory and time complexities and energy consumption that are independent of the length of the stream, and depend only on the size of the query. Thus, a QRE program automatically gives a baseline implementation with constant cost per data item. One also automatically gets a static upper bound on the per-item cost of a QRE. This allows a cost comparison to choose between similarly performing algorithms. Such early feedback on cost allows early design exploration, at a point in the design cycle where algorithmic changes are easy and can be correlated to cost decrease, and where it is well-established that the most gains are possible.

Of course, during design exploration, AMAs can also be programmed in a general purpose language like C++, and in a nonstreaming fashion, e.g., by keeping a sliding window big enough to store the entire signal segment of interest and repeating all computations with every new sample that enters the window. However, this requires the programmer to explicitly think of keeping state information and minimizing it, and to think of various sources of delay in her code and minimize those. Moreover, it is much harder to obtain upper bounds on cost (whether power, memory or processing time) of free-form code than the cost of QREs, which have sufficient structure to enable the above analysis. Finally, when it is possible, analysis of cost at code-level enables late-stage implementation changes whose effect on cost will typically be small compared to early-stage algorithmic changes.

On adopting a domain-specific language. In general, learning a new language incurs overhead for the engineers. This is true for *any* programming language, not only a DSL like StreamQRE, and the above-cited studies indicate the overall productivity gains that can be achieved after the initial learning curve. For instance, regular expressions are familiar to database developers who favor them over writing C code for querying databases. In our project, we had two teams: Team M, consisting of the first author and two other engineers, coded the AMA in Matlab. Matlab was chosen because Team M members are very familiar with it, it is an easy language to work in, and has a very rich development environment and IDE. After understanding the algorithm, it took Team M approximately three weeks to code it and check its operation (amount of time estimated from github commits). Team Q, consisting of the second and third author, took one week to code the same AMA in StreamQRE. Thus our experience validates the general point that using a DSL can unlock productivity gains.

In summary, the advantages of describing AMAs in a streaming language, and more specifically in StreamQRE, over describing them in a general purpose language, are as follows.

- A more natural way to reason about the algorithm's streaming operation, which highlights opportunities to reuse computation results.
- A declarative way to program the algorithm, which enables reasoning at the stream level and how it needs to be divided hierarchically and

processed, rather than get bogged down in item-level computations.
- An automatic implementation of the algorithm that guarantees bounded memory, runtime, and energy consumption per data item that is independent of the input signal length. The algorithm designer is relieved from having to explicitly maintain state.
- An automatic way to obtain an upper bound on the cost of a QRE as a function of the costs of the basic operations. This cost can model power consumption, for example.

IV. INTRODUCTION TO QREs

This section is an introduction to the language of QREs. First, we present the semantic model of streaming functions for describing stateful streaming transformations. Then, we introduce the language of QREs and define some derived constructs that will be used later to specify the arrhythmia detection algorithm. Finally, we discuss an efficient implementation of QREs as a Java library.

A. Streaming Functions

We introduce here the basic semantic objects for our language, called *streaming functions*, which are partial functions from sequences of input data items to an output value. Each streaming function has an associated *rate* that captures its domain; that is, as the function reads the input data stream, the rate characterizes the prefixes that trigger the production of the output. In our language, the rates are required to be *regular*, captured by symbolic regular expressions, which lead to decision procedures for constructing well-typed expressions.

As a motivating example, consider a stream that consists of integers and special separator symbols #

$$3 \quad -5 \quad 4 \quad 1 \quad -3 \quad \# \quad 7 \quad -2 \quad 9 \quad \# \quad 1 \quad -4.$$

Given such an input data stream, suppose we want to specify the transformation, illustrated as follows, that outputs at every occurrence of the # symbol the sum of all integers from the start of the stream

input :	3	−5	4	1	−3	#	7	−2	9	#	1	−4
output :						0				14.		

This transformation can be modeled by a streaming function, i.e., a partial function $f : D^* \rightharpoonup \mathbb{Z}$, where $D = \mathbb{Z} \cup \{\#\}$ is the set of input data items. For example, $f(3\,{-5}\,4\,1\,{-3}\,\#) = 0$ and $f(3\,{-5}\,4\,1\,{-3}\,\#\,7\,{-2}\,9\,\#) = 14$. The rate of f is the set of all finite sequences over D that end with #, which is denoted by the regular expression $D^* \cdot \#$. This rate is also captured by the equivalent expression $(\mathbb{Z}^* \cdot \#)^+$, where $\mathbb{Z}^* \cdot \#$ matches a block of integers terminated by a # symbol.

Suppose now that we want to process further the output stream produced by f in order to emit at every occurrence of a negative output of f the count of all negative outputs of f so far. This second processing state is described by a streaming function $g : \mathbb{Z}^* \rightharpoonup \mathbb{N}$, whose rate is denoted by the regular expression $\mathbb{Z}^* \cdot \mathbb{Z}^{<0}$, that counts the number of negative input elements and emits the count at every occurrence of a negative input item. We write $\mathbb{Z}^{<0}$ for the set of negative integers. The overall computation is described by the *streaming composition* $f \gg g$, which supplies the stream of outputs produced by f as the input stream to g.

B. Quantitative Regular Expressions

We will introduce now the language of QREs for representing stream transformations. For brevity, we also call these expressions *queries*. A query represents a streaming transformation whose domain is a regular set over the input data type.

To define queries, we first choose a typed signature which describes the basic data types and operations for manipulating them. We fix a collection of *basic types*, and we write $A, B, \ldots$ to range over them. This collection contains the type $\mathbb{B}$ of Boolean values, and the unit type $\mathbb{U}$ whose unique inhabitant is denoted by `def`. It is also closed under the Cartesian product operation $\times$ for forming pairs of values. Typical examples of basic types are the natural numbers $\mathbb{N}$, the integers $\mathbb{Z}$, the rationals $\mathbb{Q}$, and the real numbers $\mathbb{R}$. We write $a : A$ to mean that a is of type A. For example, we have $\texttt{def} : \mathbb{U}$.

We also fix a collection of *basic operations* on the basic types, for example the k-ary operation $op : A_1 \times \cdots \times A_k \to B$. The identity function on D is written as $\texttt{id}_D : D \to D$, and the operations $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ are the left and right projection, respectively. We assume that the collection of operations contains all identities and projections and is closed under pairing and function composition. To describe derived operations we use a variant of lambda notation that is similar to Java's lambda expressions [32]. That is, we write $(Ax) -> t(x)$ to mean $\lambda x{:}A.t(x)$, which is an (anonymous) function that takes an argument x of type A and returns the value $t(x)$. We write $(Ax, By, Cz) -> t(x, y, z)$ to mean $\lambda x{:}A, y{:}B, z{:}C.t(x, y, z)$. For example, the identity function on D is $(Dx) -> x$, the left projection on $A \times B$ is $(Ax, By) -> x$, the right projection on $A \times B$ is $(Ax, By) -> y$, and $(Dx) -> \texttt{def}$ is the unique function from D to $\mathbb{U}$. We will typically use lambda expressions in the context of queries from which the types of the input variables can be inferred, so we will omit the types as in $(x, y) -> x$.

For every basic type D, assume that we have fixed a collection of *atomic predicates*, so that the satisfiability of their Boolean combinations (built up using the Boolean operations: and, or, not) is decidable. We write $\varphi : D \to \mathbb{B}$ to indicate that φ is a predicate on D, and we denote by $\texttt{true}_D : D \to \mathbb{B}$ the predicate that is always true. The predicate $((\mathbb{Z}x) -> x > 0) : \mathbb{Z} \to \mathbb{B}$ is true of the strictly positive integers.

Example 4.1: We consider a Boolean ventricular heart signal, where the data items are values of type $\mathbb{B} = \{0, 1\}$. A value 1 indicates a ventricular contraction of the heart, and a value 0 indicates the absence of a contraction. The signal is sampled uniformly with a sampling rate of f Hz. The predicates $\neg$isV and isV test if a Boolean value is zero or one, respectively.

For a type D, we define the set of *symbolic regular expressions over* D [33], denoted $\mathtt{RE}\langle D\rangle$, with the grammar:

$$\begin{aligned} r ::= {} & \varphi \mid && \text{[predicate on } D\text{]} \\ & \varepsilon \mid && \text{[empty sequence]} \\ & r \sqcup r \mid && \text{[nondeterministic choice]} \\ & r \cdot r \mid && \text{[concatenation]} \\ & r^* && \text{[iteration]}. \end{aligned}$$

The concatenation symbol $\cdot$ is sometimes omitted, that is, we write rs instead of $r \cdot s$. The expression r^+ (iteration at least once) abbreviates $r \cdot r^*$. We write $r : \mathtt{RE}\langle D\rangle$ to indicate the r is a regular expression over D. Every expression $r : \mathtt{RE}\langle D\rangle$ is interpreted as a set $[\![r]\!] \subseteq D^*$ of finite sequences over D

$$[\![\varphi]\!] \triangleq \{d \in D \mid \varphi(d) \text{ is true}\}$$

and the rest of the regular construct have their usual interpretations. Two expressions are said to be *equivalent* if they denote the same language.

Example 4.2: The symbolic regular expression $(\neg\mathtt{isV})^* \cdot \mathtt{isV}$ denotes sequences of samples that contain a single ventricular beat (contraction) at the end.

A string can be matched efficiently against a regular expression if there is only one way in which it could match the expression. Intuitively, this reduces the number of possible matches that have to be kept track of. The notion of *unambiguity* for regular expressions [34] is a way of formalizing the requirement of uniqueness of parsing. The languages L_1, L_2 are said to be *unambiguously concatenable* if for every word $w \in L_1 \cdot L_2$ there are unique $w_1 \in L_1$ and $w_2 \in L_2$ with $w = w_1 w_2$. The language L is said to be *unambiguously iterable* if for every word $w \in L^*$ there is a unique integer $n \geq 0$ and unique $w_i \in L$ with $w = w_1 \cdots w_n$. The definitions of unambiguous concatenability and unambiguous iterability extend to regular expressions in the obvious way. Now, a regular expression is said to be *unambiguous* if it satisfies the following.

1) For every subexpression $e_1 \sqcup e_2$, e_1 and e_2 are *disjoint*.
2) For every subexpression $e_1 \cdot e_2$, e_1 and e_2 are *unambiguously concatenable*.
3) For every subexpression e^*, e is *unambiguously iterable*.

Example 4.3: Consider the finite alphabet $\Sigma = \{a, b\}$. The regular expression $r = (a \sqcup b)^* b (a \sqcup b)^*$ denotes the set of sequences with at least one occurrence of b. It is ambiguous, because the subexpressions $(a \sqcup b)^* b$ and $(a \sqcup b)^*$ are not unambiguously concatenable: the word $w = ababa$ matches r, but there are two different splits $w = ab \cdot aba$ and $w = abab \cdot a$ that witness the ambiguity of parsing. The regular expressions $a^* b (a \sqcup b)^*$ and $(a \sqcup b)^* b a^*$ are both equivalent to r, and they are unambiguous.

Checking whether a regular expression is unambiguous can be done in polynomial time. For the case of symbolic regular expressions this results still holds, under the assumption that satisfiability of the predicates can be decided in unit time [35].

After these preliminaries, we now define quantitative regular expressions, or queries, recursively. Informally, a query f is a symbolic regular expression, called the *rate* of f and written $\mathsf{R}(\mathtt{f})$, with a way to compute quantities over the strings that match the expression. The rate denotes the domain of the transformation that f represents. The definition of the query language has to be given simultaneously with the definition of rates (by mutual induction), since the query constructs have typing restrictions that involve the rates. We annotate a query f with a type $\mathtt{QRE}\langle D, C\rangle$ to denote that the input stream has elements of type D and the outputs are values of type C.

1) Atomic Queries: The basic building blocks of queries are expressions that describe the processing of a single data item. Suppose $\varphi : D \to \mathbb{B}$ is a predicate over the data item type D and $op : D \to C$ is an operation from D to the output type C. Then, the *atomic query* $\mathtt{atom}(\varphi, op) : \mathtt{QRE}\langle D, C\rangle$, with rate φ, is defined on single-item streams that satisfy the predicate φ. The output is the value of op on the input element.

Notation: It is very common for op to be the identity function, and φ to be the always-true predicate. So, we abbreviate the query $\mathtt{atom}(\varphi, \mathtt{id}_D)$ by $\mathtt{atom}(\varphi)$, and the query $\mathtt{atom}(\mathtt{true}_D)$ by $\mathtt{atom}()$.

Example 4.4: For the Boolean ventricular heart signal, the query that matches a single item that is a heartbeat and returns nothing is $\mathtt{f} = 0, 0, 0.65\mathtt{atom}(\mathtt{isV}, x{-}{>}\mathtt{def})$. The type of f is $\mathtt{QRE}\langle\mathbb{B}, \mathbb{U}\rangle$ and its rate is isV.

2) Empty Sequence: The query $\mathtt{eps}(c) : \mathtt{QRE}\langle D, C\rangle$, where c is a value of type C, is only defined on the empty sequence ε and it returns the output c.

3) Iteration: Suppose that we want to iterate a computation $\mathtt{f} : \mathtt{QRE}\langle D, A\rangle$ over consecutive subsequences of the input stream and aggregate all these output values sequentially using an initial value $c : B$ and an aggregation operation $op : B \times A \to B$. The iteration query

$$\mathtt{iter}(\mathtt{f}, c, op) : \mathtt{QRE}\langle D, B\rangle$$

describes this computation. More specifically, we split the input stream w into subsequences $w = w_1\, w_2 \ldots w_n$, where each w_i matches f. The output values $a_1\, a_2 \cdots a_n$ with $a_i = \mathtt{f}(w_i)$ are combined using the list iterator *left fold* with start value $c : B$ and aggregation operation op :

$B \times A \to B$. This can be formalized with the combinator

$$\text{fold} : B \times (B \times A \to B) \times A^* \to B$$

which takes an initial value $b : B$ and a stepping map $op : B \times A \to B$, and iterates through a sequence of values of A

$$\begin{aligned} \text{fold}(b, op, \varepsilon) &= b \\ \text{fold}(b, op, \gamma a) &= op(\text{fold}(b, op, \gamma), a) \end{aligned}$$

for all sequences $\gamma \in A^*$ and all values $a \in A$. For example, $\text{fold}(b, op, a_1 a_2) = op(op(b, a_1), a_2)$.

In order for $\texttt{iter}(\texttt{f}, c, op)$ to be well-defined as a function, every input stream w that matches $\texttt{iter}(\texttt{f}, c, op)$ must be uniquely decomposable into $w = w_1 w_2 \dots w_n$ with each w_i matching $\texttt{f}$. This requirement can be expressed equivalently as: the rate $\mathsf{R}(\texttt{f})$ is unambiguously iterable.

Example 4.5: For the Boolean heart signal, the query g below matches a sequence of data items that are not heartbeats and returns their count:

$$\begin{aligned} \texttt{f} : \texttt{QRE}\langle \mathbb{B}, \mathbb{B} \rangle &= \texttt{atom}(\neg\texttt{isV}) \\ \texttt{g} : \texttt{QRE}\langle \mathbb{B}, \mathbb{N} \rangle &= \texttt{iter}(\texttt{f}, 0, (x, y) \texttt{->} x + 1). \end{aligned}$$

The rate of f is $\neg$isV, and the rate of g is $(\neg\texttt{isV})^*$.

4) Combination and Application: Assume the queries f and g describe stream transformations with outputs of type A and B respectively that process the same set of input sequences, and op is a function of type $A \times B \to C$. Then

$$\texttt{combine}(\texttt{f}, \texttt{g}, op) : \texttt{QRE}\langle D, C \rangle$$

describes the computation where the input is processed according to both f and g in parallel and their results are combined using op. This computation is meaningful only when both f and g are defined on the input sequence. So, we demand w.l.o.g. that the rates of f and g are equivalent.

This binary combination construct generalizes to an arbitrary number of queries. For example, we write

$$\texttt{combine}(\texttt{f}, \texttt{g}, \texttt{h}, (x, y, z) \texttt{->} op(x, y, z))$$

for the ternary variant. In particular, we write $\texttt{apply}(\texttt{f}, op)$ for the case of one argument.

Example 4.6: For the Boolean heart signal, suppose g counts all heartbeats seen so far and h counts all data items. Then, the query k below computes the ratio of these values.

$$\begin{aligned} \texttt{f} : \texttt{QRE}\langle \mathbb{B}, \mathbb{N} \rangle &= \texttt{atom}(\texttt{true}_{\mathbb{B}}, x \texttt{->} \texttt{if } x \texttt{ then } 1 \texttt{ else } 0), \\ \texttt{g} : \texttt{QRE}\langle \mathbb{B}, \mathbb{N} \rangle &= \texttt{iter}(\texttt{f}, 0, (x, y) \texttt{->} x + y) \\ \texttt{h} : \texttt{QRE}\langle \mathbb{B}, \mathbb{N} \rangle &= \texttt{iter}(\texttt{atom}(), 0, (x, y) \texttt{->} x + 1) \\ \texttt{k} : \texttt{QRE}\langle \mathbb{B}, \mathbb{Q} \rangle &= \texttt{combine}(\texttt{g}, \texttt{h}, (x, y) \texttt{->} x/y) \end{aligned}$$

The rate of f is true and the rates of the queries g, h and k are all equal to $\texttt{true}^*$.

5) Quantitative Concatenation: Suppose that we want to perform two streaming computations in sequence: first execute the query $\texttt{f} : \texttt{QRE}\langle D, A \rangle$, then the query $\texttt{g} : \texttt{QRE}\langle D, B \rangle$, and finally combine the two results using the operation $op : A \times B \to C$. The query

$$\texttt{split}(\texttt{f}, \texttt{g}, op) : \texttt{QRE}\langle D, C \rangle$$

describes this computation. More specifically, we split the input into two parts $w = w_1 w_2$, process the first part w_1 according to f with output $\texttt{f}(w_1)$, process the second part w_2 according to g with output $\texttt{g}(w_2)$, and produce the final result $op(\texttt{f}(w_1), \texttt{g}(w_2))$ by applying op to the intermediate results.

In order for this construction to be well-defined as a function, every input w that matches $\texttt{split}(\texttt{f}, \texttt{g}, op)$ must be uniquely decomposable into $w = w_1 w_2$ with w_1 matching f and w_2 matching g. In other words, the rates of f and g must be unambiguously concatenable.

The binary split construct extends naturally to more than two arguments. For example, the ternary version would be $\texttt{split}(\texttt{f}, \texttt{g}, \texttt{h}, (x, y, z) \texttt{->} op(x, y, z))$.

Example 4.7: For the Boolean heart signal, suppose that g matches sequences that end with a heartbeat and h counts the size of sequences without any heartbeat. Then, the query k outputs the time that has elapsed since the last heartbeat.

$$\begin{aligned} \texttt{f} : \texttt{QRE}\langle \mathbb{B}, \texttt{Ut} \rangle &= \texttt{iter}(\texttt{atom}(), \texttt{def}, (x, y) \texttt{->} \texttt{def}) \\ \texttt{g} : \texttt{QRE}\langle \mathbb{B}, \texttt{Ut} \rangle &= \texttt{split}(\texttt{f}, \texttt{atom}(\texttt{isV}), (x, y) \texttt{->} \texttt{def}) \\ \texttt{h} : \texttt{QRE}\langle \mathbb{B}, \mathbb{N} \rangle &= \texttt{iter}(\texttt{atom}(\neg\texttt{isV}), 0, (x, y) \texttt{->} x + 1) \\ \texttt{k} : \texttt{QRE}\langle \mathbb{B}, \mathbb{N} \rangle &= \texttt{split}(\texttt{g}, \texttt{h}, (x, y) \texttt{->} y) \end{aligned}$$

The rate of f is $\texttt{true}^*$, that of g is $\texttt{true}^* \cdot \texttt{isV}$, the rate of h is $(\neg\texttt{isV})^*$, and the rate of k is $\texttt{true}^* \cdot \texttt{isV} \cdot (\neg\texttt{isV})^*$.

6) Streaming Composition: A natural operation for query languages over streaming data is streaming composition: given two streaming queries f and g, $\texttt{f} \gg \texttt{g}$ represents the computation in which the stream of outputs produced by f is supplied as the input stream to g. Such a composition is useful in setting up the query as a pipeline of several stages. We allow the operation $\gg$ to appear *only at the top-level* of a query. So, a general query is a pipeline of $\gg$-free queries. At the top level, no type checking needs to be done for the rates, so we do not define the function $\mathsf{R}()$ for queries $\texttt{f} \gg \texttt{g}$.

Example 4.8: For the Boolean heart signal, suppose we want to emit at every heartbeat the average heart rate over the entire stream. We will describe this computation as a two-stage pipeline. The first stage (query h as follows)

produces a sequence of natural numbers which correspond to the number of 0's between two consecutive 1's (heartbeats).

$$\begin{aligned}
\mathtt{f} : \mathtt{QRE}\langle \mathbb{B}, \mathbb{N}\rangle &= \mathtt{iter}(\mathtt{atom}(\neg \mathtt{isV}), 0, (x,y) \mathtt{->} x+1)\\
\mathtt{g} : \mathtt{QRE}\langle \mathbb{B}, \mathbb{N}\rangle &= \mathtt{split}(\mathtt{f}, \mathtt{atom}(\mathtt{isV}), (x,y) \mathtt{->} x)\\
\mathtt{h} : \mathtt{QRE}\langle \mathbb{B}, \mathbb{N}\rangle &= \mathtt{split}(\mathtt{iter}(\mathtt{g}, \mathtt{def}, (x,y) \mathtt{->} \mathtt{def}),\\
&\qquad \mathtt{g}, (x,y) \mathtt{->} y).
\end{aligned}$$

The rate of f is $(\neg \mathtt{isV})^*$, the rate of g is $(\neg \mathtt{isV})^* \cdot \mathtt{isV}$, and the rate of h is $((\neg \mathtt{isV})^* \cdot \mathtt{isV})^+$. The second stage (query n as follows) processes a stream of these numbers to compute the average heart rate in beats per minute.

$$\begin{aligned}
\mathtt{k} : \mathtt{QRE}\langle \mathbb{N}, \mathbb{N}\rangle &= \mathtt{iter}(\mathtt{atom}(), 0, (x,y) \mathtt{->} x+y)\\
\mathtt{l} : \mathtt{QRE}\langle \mathbb{N}, \mathbb{N}\rangle &= \mathtt{iter}(\mathtt{atom}(), 0, (x,y) \mathtt{->} x+1)\\
\mathtt{m} : \mathtt{QRE}\langle \mathbb{N}, \mathbb{Q}\rangle &= \mathtt{combine}(\mathtt{k}, \mathtt{l}, (x,y) \mathtt{->} x/y))\\
\mathtt{n} : \mathtt{QRE}\langle \mathbb{Q}, \mathbb{Q}\rangle &= \mathtt{apply}(\mathtt{m}, x \mathtt{->} (60 \cdot f)/x)
\end{aligned}$$

where f is the sampling rate in Hertz. The query m computes the average number of samples between two consecutive heartbeats. The top-level query is the pipeline $\mathtt{h} \gg \mathtt{n}$.

7) Global Choice: Given queries f and g of the same type with disjoint rates r and s, the query $\mathtt{or}(\mathtt{f}, \mathtt{g})$ applies either f or g to the input stream depending on which one is defined. The rate of $\mathtt{or}(\mathtt{f}, \mathtt{g})$ is the union $r \sqcup s$. This *choice* construction allows a case analysis based on a global regular property of the input stream.

Example 4.9: In our Boolean heart example, suppose we want to compute a statistic across days, where the contribution of each day is computed differently depending on whether or not an abnormally short interval between consecutive heartbeats occurred or not. Then, we can write a query summarizing the daily activity with a rate capturing normal days (the ones without any short interval) and a different query with a rate capturing abnormal days, and iterate over their disjoint union.

Consider the stream of interval lengths between consecutive heartbeats, i.e., the output stream of query h defined in Example 4.8. We assume that T is the threshold for an abnormally short interval between two consecutive heartbeats. Query h as follows computes the smallest interval length for sequences with at least one abnormally short interval:

$$\begin{aligned}
\mathtt{f} : \mathtt{QRE}\langle \mathbb{N}, \mathbb{Q}\rangle &= \mathtt{iter}(\mathtt{atom}(x \mathtt{->} x > T), \infty, \min)\\
\mathtt{g} : \mathtt{QRE}\langle \mathbb{N}, \mathbb{Q}\rangle &= \mathtt{iter}(\mathtt{atom}(), \infty, \min)\\
\mathtt{h} : \mathtt{QRE}\langle \mathbb{N}, \mathbb{Q}\rangle &= \mathtt{split}(\mathtt{f}, \mathtt{atom}(x \mathtt{->} x \leq T), \mathtt{g}, \min).
\end{aligned}$$

The rate of f is $(x > T)^*$, the rate of g is $\mathtt{true}^*$, and the rate of h is $(x > T)^* \cdot (x \leq T) \cdot \mathtt{true}^*$. Query m as follows computes the average interval length for sequences with no abnormally short interval:

$$\begin{aligned}
\mathtt{k} : \mathtt{QRE}\langle \mathbb{N}, \mathbb{N}\rangle &= \mathtt{iter}(\mathtt{atom}(x \mathtt{->} x \mathtt{>} T), 0, (x,y) \mathtt{->} x+y)\\
\mathtt{l} : \mathtt{QRE}\langle \mathbb{N}, \mathbb{N}\rangle &= \mathtt{iter}(\mathtt{atom}(x \mathtt{->} x \mathtt{>} T), 0, (x,y) \mathtt{->} x+1)\\
\mathtt{m} : \mathtt{QRE}\langle \mathbb{N}, \mathbb{Q}\rangle &= \mathtt{combine}(\mathtt{k}, \mathtt{l}, (x,y) \mathtt{->} x/y).
\end{aligned}$$

The rates of k, l and m are all equal to $(x > T)^*$. The top-level query is then $\mathtt{or}(\mathtt{h}, \mathtt{m})$.

C. Derived Constructs

The core language of Section IV-B is expressive enough to describe many common stream transformations. We present several derived constructs.

1) Matching Without Output: Suppose r is an unambiguous symbolic regular expression over the data item type D. The query $\mathtt{match}(r)$, whose rate is equal to r, does not produce any output when it matches. This is essentially the same as producing def as output for a match. The match construct can be encoded as follows:

$$\begin{aligned}
\mathtt{match}(\varphi) &\triangleq \mathtt{atom}(\varphi, x \mathtt{->} \mathtt{def})\\
\mathtt{match}(r_1 \sqcup r_2) &\triangleq \mathtt{or}(\mathtt{match}(r_1), \mathtt{match}(r_2))\\
\mathtt{match}(r_1 \cdot r_2) &\triangleq \mathtt{split}(\mathtt{match}(r_1), \mathtt{match}(r_2), (x,y) \mathtt{->} \mathtt{def})\\
\mathtt{match}(r^*) &\triangleq \mathtt{iter}(\mathtt{match}(r), \mathtt{def}, (x,y) \mathtt{->} \mathtt{def}).
\end{aligned}$$

An easy induction establishes that $\mathsf{R}(\mathtt{match}(r)) = r$.

2) "Until" Iteration: Suppose that ϕ and ψ are disjoint predicates on the input data type D, the function $op : C \times D \to C$ is an aggregation operation, and $c : C$ is the initial aggregate. The query $\mathtt{iterUntil}(\phi, \psi, c, op)$ aggregates a sequence of data items that satisfy ϕ and stops when an item that satisfies ψ is found. It is encoded as

$$\begin{aligned}
\mathtt{iterUntil}(\phi, \psi, c, op) \triangleq \mathtt{split}(&\mathtt{iter}(\mathtt{atom}(\phi), c, op),\\
&\mathtt{atom}(\psi), (x,y) \mathtt{->} x).
\end{aligned}$$

The query has type $\mathtt{QRE}\langle D, C\rangle$ and rate $\phi^* \cdot \psi$.

3) Stream Annotation: Suppose that the input stream has items of type D, f is a query of type $\mathtt{QRE}\langle D, C\rangle$, and we want to produce an output stream with items of type E in the following way: when the query f produces an output (upon consumption of the input stream) apply $op_2 : D \times C \to E$ to the last input element and its output to get the final result, and when the query f is undefined apply $op_1 : D \to E$ to the last input element. This computation is described by the query $\mathtt{annt}(\mathtt{f}, op_1, op_2) : \mathtt{QRE}\langle D, E\rangle$ with rate D^+. This annotation query can be encoded using the regular constructs of Section IV-B, but the encoding is complex and inefficient, so we provide a custom efficient implementation.

a) Tumbling windows: The term tumbling windows is used to describe the splitting of the stream into contiguous

nonoverlapping subsequences [36]. Suppose we want to describe the streaming function that iterates f at least once and reports the result given by f at every match. The following query expresses this behavior:

$$\texttt{iterLast(f)} \triangleq \texttt{split(match}(\mathsf{R}(\texttt{f})^*), \texttt{f}, (x, y) \dashrightarrow y).$$

The rate of iterLast(f) is equal to $\mathsf{R}(\texttt{f})^+$.

4) Efficient Sliding Windows: Suppose we want to apply the query $\texttt{f} : \texttt{QRE}\langle D, A\rangle$ to consecutive nonoverlapping parts of the input, and efficiently aggregate the intermediate results over a sliding window of size W. That is, the W most recent output values of f are aggregated to produce the final output. The aggregation is described by an initial aggregate $c : B$ and three functions: an *insertion* operation $\texttt{ins} : B \times A \to B$ describes how to add a new value of type A to the aggregate (of type B), the *removal* operation $\texttt{rmv} : B \times A \to B$ describes how to remove a value from the aggregate, and the *finalization* operation $op : B \to C$ computes the final result from the aggregate. This computation is described by the query

$$\texttt{wnd(f}, W, c, \texttt{ins}, \texttt{rmv}, op) : \texttt{QRE}\langle D, C\rangle$$

whose rate is equal to $\mathsf{R}(\texttt{f})^+$. This query can be encoded using the regular constructs of Section IV-B and an additional data type for FIFO queues (in order to maintain the buffer of values of type A that are currently in the active window).

D. Java Library of QREs

StreamQRE has been implemented as a Java library [18] in order to facilitate the easy integration with user-defined types and operations. The implementation covers all the core constructs of Section IV-B, and also provides optimizations for the derived constructs of Section IV-C (matching without output, "until" iteration, stream annotation, etc.).

Fig. 4 gives a simple example that illustrates how to program with the StreamQRE Java library. The query avg describes the computation of the average of a sequence of values of type Double. The method getEval, which stands for "get evaluator", is used to obtain an object that encapsulates the evaluation algorithm for the query. On this evaluator object, the methods start and next are used to initialize the algorithm and consume data items respectively.

```
// Process a single value: rate Double
QRe<Double, Double> f =
    Q.atomic(x -> true, x -> x);

// Sum of sequence of values: rate Double*
QRe<Double, Double> sum =
    Q.iter(f, 0.0, (x,y) -> x+y);

// Length of sequence of values: rate Double*
QRe<Double, Long> count =
    Q.iter(f, 0L, (x,y) -> x+1);

// Average of sequence of values: rate Double*
QRe<Double, Double> avg =
    Q.combine(sum, count, (x,y) -> x/y);

Iterator<Double> stream = ... // input stream

// evaluator for the query
Eval<Double, Double> e = avg.getEval();

// execution loop
Double output = e.start();
// e.start() returns null, if undefined
while (stream.hasNext()) {
    Double d = stream.next();
    output = e.next(d);
    // e.next(d) returns null, if undefined
}
```

Fig. 4. ***StreamQRE Library in Java: Computing the average of a sequence of values.***

V. ICD ARRHYTHMIA MONITORING ALGORITHM

We now describe in detail an Arrhythmia Monitoring Algorithm (AMA) found in one of the ICDs on the market today [17]. All ICD AMAs on the market today take the form of a decision tree, such as the one in Fig. 5. Each node in the tree is a *discriminator*, which computes one feature of the input signal and decides, on its basis, how to branch. Thus, each discriminator returns a decision, Yes or No. We chose to present this particular AMA because variants on its discriminators can be found in the AMAs of all devices on the market. For example, all devices measure average heart rate, compare atrial and ventricular rates, measure rate variability, onset of arrhythmia, etc. The differences are in how variability is defined (variance or sum of absolute differences, for example), the size of windows for computing quantities, the way they are combined in the decision tree, etc.

A. Discriminators

Recall that the input to the AMA is a discrete-time Boolean signal, which is obtained by running a peak detector on the discrete-time real-valued EGM signal. The peak detector outputs a 1 at peaks, and 0 otherwise. The signals we work with in this paper have a sampling period of 1 ms. Formally, let $\mathbb{B} = \{0, 1\}$. At every time $t \in \mathbb{N}$, the AMA receives a data item s of the following form:

$$s = (V, A, t) \in D := \mathbb{B} \times \mathbb{B} \times \mathbb{N} \tag{1}$$

where $V = 1$ indicates there is a ventricular beat at time t (and $V = 0$ indicates that there is not), similarly for A. We will find the need to refer to the ventricular Boolean signal separately, and we write $\mathbf{V} \in \mathbb{B}^*$ to denote it. It will also be called the ventricular *channel*. Similarly, $\mathbf{A} \in \mathbb{B}^*$ is the

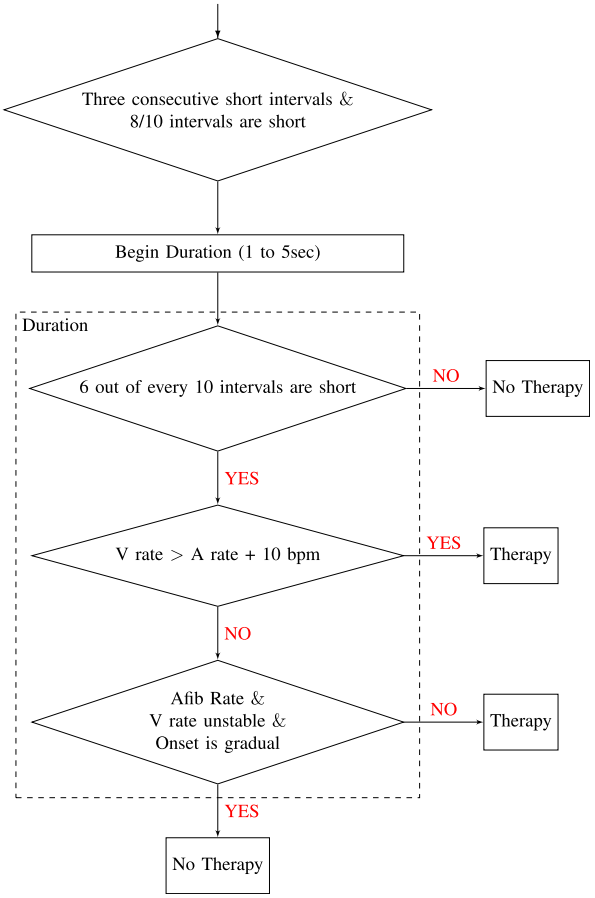

Fig. 5. ***Boston Scientific discrimination algorithm.***

atrial channel. See Fig. 6. Given an item s, the function call `s.V` returns its first element, similarly for `s.A` and `s.t`.

An (atrial or ventricular) *interval* in a given channel is the interval of time between two consecutive beats. Its length is denoted by I and is always an integer measured in milliseconds (ms). The average (atrial or ventricular) *rate* is the inverse of the average interval length.

The decision tree of the AMA we describe is shown in Fig. 5. It is made up of the following discriminators.

1) Three Consecutive Short Intervals: Three consecutive short intervals are required to initiate rhythm analysis, as they indicate a potentially accelerating rhythm. Therefore, this discriminator checks if three consecutive intervals are shorter than some prespecified threshold T_{csi}. Referring to Fig. 6

$$\text{CSI} := (I_5 < T_{\text{csi}}) \wedge (I_6 < T_{\text{csi}}) \wedge (I_7 < T_{\text{csi}}). \quad (2)$$

2) 8/10 Short Intervals: A rhythm that becomes fast for a few beats then slows down again is not fatal and so should not cause therapy to be delivered. To estimate whether the current rhythm is sustained, this discriminator checks whether 8 out of 10 intervals are shorter than some threshold $T_{8/10}$. Referring to Fig. 6

$$\text{Short8outof10} := |\{I_k : 5 \le k \le 14, I_k < T_{8/10}\}| \ge 8. \quad (3)$$

3) Sudden Onset: VF, which is fatal, usually occurs suddenly, whereas a tachycardia that accelerates gradually is usually nonfatal. The onset discriminator quantifies the suddenness of tachycardia onset as follows. It operates in two steps, which process a window of $2m$ intervals. To help explain this discriminator using Fig. 6, we will assume $m = 4$. In the first step, it detects the ventricular beat in the first four intervals $(I_1, \ldots, I_4)$ at which the interval length decreased the most. This is the *pivot* beat. If the amount of decrease is greater than some threshold, Step I declares Onset. In the second step, the algorithm computes the differences between the average of four prepivot beats $((I_1 + \cdots + I_4)/4 := \mu)$ and each of four post-pivot beats

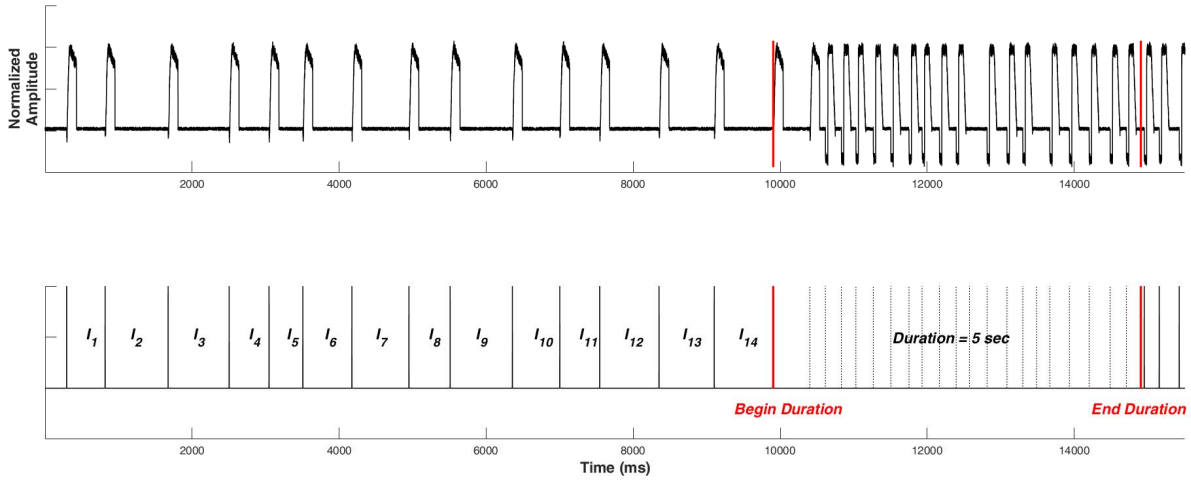

Fig. 6. ***Input stream from one channel. Measured electrogram (top figure) and corresponding Boolean stream (bottom figure). In the Boolean stream, spikes represent beats, and I_k is an interval of time between beats. Duration is a fixed time period, here set to 5 s.***

$(I_5, \ldots, I_8)$, i.e., it computes $d_5 = \mu - I_5, \ldots, d_8 = \mu - I_8$. If at least three of these four differences $d_5, \ldots, d_8$ is greater than a threshold, Step II declares Onset. If both stages declare Onset, the discriminator declares Sudden Onset. In our implementation, we simplify things by taking the pivot to be the middle beat in the window of $2m = 8$ intervals. So Sudden Onset is computed as

$$\text{SO-StepI} := I_{post-pivot} < \alpha \cdot I_{pre-pivot} \quad (4)$$

$$\text{SO-StepII} := |\{d_k : d_k > T_{o2}\}| \geq 3 \quad (5)$$

$$\text{SuddenOnset} := \text{SO-StepI} \wedge \text{SO-StepII}.$$

When both Three Consecutive Short Intervals and 8/10 Short Intervals match, then a *Duration* is started. A Duration is a fixed-length time period (e.g., 5 s) during which the algorithm will continue to monitor the rhythm to see whether the arrhythmia is sustained, or it slows down and dies out. In the latter case, no therapy is delivered. See Fig. 6. During Duration, the following four discriminators are evaluated.

4) A/V Rate Comparison: If the ventricles have more beats than the atria, this is an almost sure sign that the arrhythmia is ventricular in origin (i.e., the ventricles are driving the atria and not the other way around). This discriminator compares the average ventricular heart rate r_V with the average atrial heart rate r_A, where the average is computed over the last ten intervals in the duration window

$$\text{AVRate} := r_V > r_A + 10\text{bpm}$$

5) Sliding 6/10: Sometimes an arrhythmia terminates on its own, which is preferable to having the device terminate it with a shock. This discriminator continuously checks whether six out of every ten intervals are short; if any 10 intervals fails this check, the discriminator declares No Therapy

$$\text{Sliding6outof10} := \text{For every 10 intervals } I_1, \ldots, I_{10} \quad |\{I_k : I_k < T_{6/10}\}| \geq 6.$$

6) Stability: VF is an unstable rhythm, meaning that the interval lengths during fibrillation vary greatly. The Stability discriminator defines rhythm stability as being the variance in ventricular intervals' lengths during Duration. If variance is below a threshold T_{stab}, then the rhythm is deemed stable. With $\bar{I}$ denoting the average interval length

$$\text{Stability} := \frac{1}{n}\sum_{k=1}^{n}(I_k - \bar{I})^2 \leq T_{\text{stab}}.$$

7) AFib Rate: AF is an atrially driven rhythm with a high rate, and is one possible source of misclassification for the AMA. To circumvent this issue, this discriminator measures the atrial rate throughout the Duration. As long as at least 4/10 intervals are shorter than the AF threshold T_{af}, this discriminator decides that the current rhythm is in fact AF and therapy should be withheld

$$\text{SlidingAFib} := \text{For every 10 interval lengths } I_1, \ldots, I_{10} \quad |\{I_k : I_k < T_{af}\}| \geq 4.$$

VI. QRE IMPLEMENTATION OF THE ARRHYTMIA MONITORING ALGORITHM

The QRE implementation of the BSC algorithm of Section V is divided into four main stages. The first two stages annotate the input signal with additional information: the lengths of the intervals between heartbeats, and some sliding-window statistics over them. The annotated stream is passed to the later stages in order to compute the discriminators for deciding whether therapy should be delivered or not. We give a high-level overview of each stage in Section VI-A, as well as more detailed descriptions and QREs implementations in Sections VI-B–VI-E.

A. Overview of Implementation Stages

All discriminators described in Section V use the interval lengths between consecutive heartbeats. In order to simplify the later computations, it is useful to annotate the stream with this extra information so that it is readily available in the next processing steps. Similarly, there are some sliding-window statistics that are required for the discriminators "A/V Rate Comparison," "Sliding 6/10" and "AFib Rate." These quantities require looking at the ten previous intervals to be computed. The specification of the algorithm is much easier if this information is already present in the stream, which obviates the need to look back ten intervals into the past. This motivates our design choice to always annotate the stream with these useful sliding-window statistics.

The ICD's AMA receives beats from the atrium and the ventricle. The input stream consists of data items that are of the form shown in 1). The implementation is a multi-stage pipeline, where each stage is a QRE. Each stage feeds its output stream to the following stage for further annotation and processing. They are as follows

- Stage 0: preprocessing stage which annotates the input stream $\mathbf{s}$ with the lengths of the ventricular and atrial intervals. See Fig. 8. The output from this stage will be used in all subsequent stages. Call the output stream of this stage $\mathbf{s_0}$.
- Stage 1: augments its input stream $\mathbf{s_0}$ with two pieces of information. The first is the total duration of every window of 10 consecutive intervals, in both channels. This will be used for the A/V Rate Comparison discriminator. The second piece of information is the number of short[3] intervals in every window of ten consecutive intervals, in both channels. This will

[3]For example, those that are shorter than a predefined threshold $T_{6/10}$.

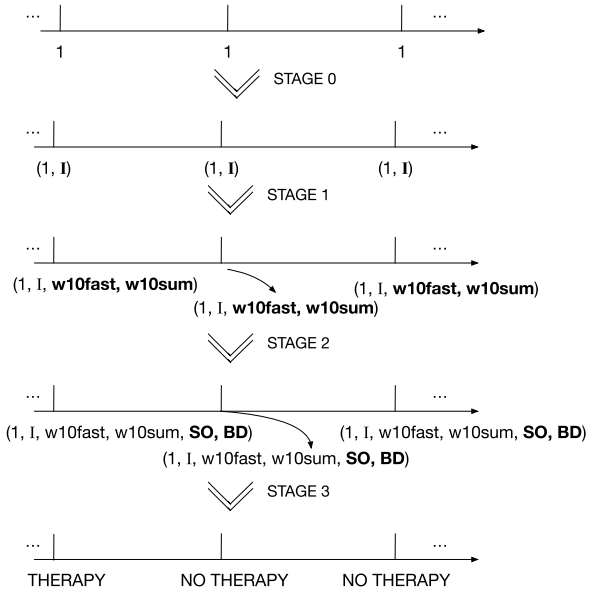

Fig. 7. *The overall detection algorithm is shown for the ventricular channel and with the timestamp sequence omitted. The top stream gives the input Boolean signal. Streams below it are annotated with the information in bold font. I = Interval Length, w10fast = number of last ten intervals that are short, w10sum = sum of last ten interval lengths, SO = Sudden Onset flag, BD = Begin Duration flag.*

be used for the Sliding 6/10 and AFib Rate Comparison discriminators. See Fig. 9 for the computation of both quantities on the V channel. Call the output stream of this stage $\mathbf{s_1}$.

- Stage 2: detects the beginning of Duration, the period of time during which the rhythm is monitored for a fixed amount of time to confirm whether a suspected arrhythmia is indeed sustained and ventricular in origin. For Duration to be declared and monitored, both the Three Consecutive Short Intervals and 8/10 Short Intervals discriminators must return Yes. If Duration is initiated as a result, the input stream $\mathbf{s_1}$ is annotated with a BD marker to indicate the start of Duration. See Fig. 10. This stage also computes the Onset discriminator and annotates the stream with flag SO $= 1$ if it is met. Call the output stream of this stage $\mathbf{s_2}$.
- Stage 3: final stage, has input stream $\mathbf{s_2}$. It computes all discriminators in Duration: Stability, Sliding 6/10, AV Rate Comparison, and AFib Rate. Based on all these and the value of Onset, the stage makes a final decision of Therapy or No Therapy. See Fig. 10.

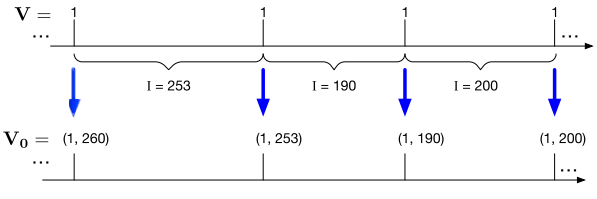

Fig. 8. *Stage 0 annotates both channels V and A with interval lengths, i.e., the number of 0s between 1s. Here it is shown operating on the ventricular channel.*

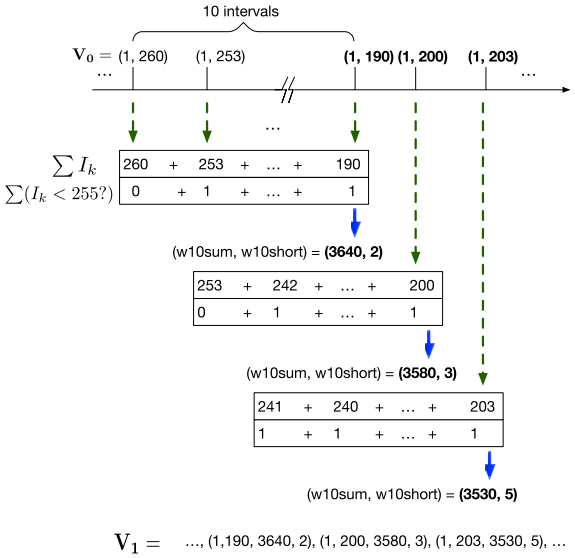

Fig. 9. *Stage 1, shown acting on the V channel, augments V_0 with the total duration counter w10sum and the short intervals counter w10short, computed over the last ten intervals. Here, the threshold $T_{6/10}$=255.*

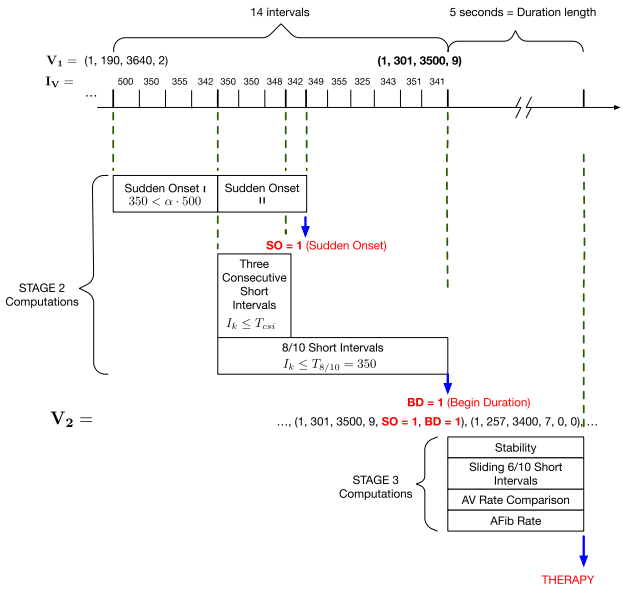

Fig. 10. *Stages 2 and 3. The rectangles show the computed discriminators, and their width covers the part of the input stream used in the computation.For example, "8/10 Short Intervals" uses the ten intervals above its box, while Stability uses all intervals in the Duration window. Downward blue arrows indicate when a quantity is computed; e.g., the BD marker is computed every 14 intervals. SO and BD are added to stream V_1 to obtain stream V_2. The A channel is not shown, though it enters in the calculation of AV Rate Comparison discriminator.*

Remark: Because a QRE describes a streaming algorithm, each of the above stages operates continuously and issues an output with every new data item (including $\perp$ if the string so far does not match). So for example, it is possible for Stage 2 to declare the start of Duration several times in a row, i.e., to output BD $= 1$ several times. See Fig. 14 for an example. The first Duration to end in a Therapy decision in Stage 3 will cause therapy to be scheduled, and the other Durations in progress are aborted. On the other hand, if the first Duration does not end in therapy, the subsequent ones continue to be monitored to their conclusion. Thus, one important consequence of this streaming implementation is that it is possible for the QRE to track *multiple simultaneous* potential arrhythmias. In this way, no potentially fatal arrhythmia is missed.

We will explain now each stage in detail, and present the precise implementation in the StreamQRE language. Recall the QRE constructs of Section IV and the type of the input data items (1). Some computations are performed in the same way both on the atrial and the ventricular channel. In such cases we will only give the queries that describe the processing of the ventricular channel for the sake of brevity.

B. Stage 0: Annotate Interval Lengths

This stage annotates the stream with heartbeat interval lengths, that is, the lengths of the sequences between two consecutive heartbeats. So, the length of an interval of the form $100\cdots001$ is the number of 0s between the 1s. This computation is performed both for the ventricular and atrial channel. The regular expression that describes a signal that has a single heartbeat at the end is 0^*1. The query for computing the ventricular interval lengths is the following:

$$\begin{aligned}
\texttt{lincr} &= (x,y) \texttt{->} x+1, \text{ of type } \mathbb{N}\times\mathbb{N}\to\mathbb{N}\\
\texttt{intV} &= \texttt{iterUntil}(\neg\texttt{isV}, \texttt{isV}, 0, \texttt{lincr})\\
\texttt{allIntV} &= \texttt{iterLast}(\texttt{intV}) \quad \text{// rate } ((\neg\texttt{isV})^*\cdot\texttt{isV})^+\\
\texttt{annt0V} &= \texttt{annt}(\texttt{allIntV}, x \texttt{->} x, (x,c) \texttt{->} x[I_V := c])\\
\texttt{stage0} &= \texttt{annt0V} \gg \texttt{annt0A}.
\end{aligned}$$

The query `intV` iterates over the 0s of the ventricular channel (predicate $\neg$`isV`) while incrementing a counter until it encounters a 1 (predicate `isV`). The query `allIntV` iterates `intV` over consecutive nonoverlapping subsequences, thus processing all ventricular intervals. The query `annt0V` annotates the input elements with the interval values I_V calculated by `allIntV`, and `annt0A` does the same with the atrial channel. See Fig. 8. Therefore, the output stream $\mathbf{s_0}$ from this stage consists of data items of the following form:

$$s_0 = (V, I_V, A, I_A, t) \in D_0 = (\mathbb{B}\times\mathbb{N})^2\times\mathbb{N}. \tag{6}$$

C. Stage 1: Sudden Onset and Short Intervals

The input stream for this stage consists of items of the form shown in (6). In this state, we first calculate the sum of interval lengths over a sliding window that consists of ten intervals, and we annotate the stream with this information (see Fig. 9)

$$\begin{aligned}
\texttt{blockV} &= \texttt{split}(\texttt{match}((\neg\texttt{isV})^*), \texttt{isV}, (x,y) \texttt{->} y.I_V)\\
\texttt{wndSumV} &= \texttt{wnd}(\texttt{blockV}, 10, 0, (x,y) \texttt{->} x+y)\\
\texttt{stg1SumV} &= \texttt{annt}(\texttt{wndSumV}, x \texttt{->} x, (x,c) \texttt{->} x[SumV := c]).
\end{aligned}$$

The query `blockV` matches 0^*1 in the ventricular channel and returns the length of the interval that ends with the matched 1. The query `wndSumV` executes `blockV` over a sliding window of size 10 and accumulates the interval lengths by summing them up. The query `stg1SumV` annotates the stream with all these sliding-window sums.

In the second part of this stage we also calculate the number of short ventricular intervals over a sliding window of size 10, where "short" is defined as being of length less than $T_{6/10}$

$$\begin{aligned}
\texttt{shortV} &= \texttt{apply}(\texttt{blockV},\\
&\qquad x \texttt{->} \texttt{if } (x \le T_{6/10}) \texttt{ then } 1 \texttt{ else } 0)\\
\texttt{wndShortV} &= \texttt{wnd}(\texttt{shortV},\ 10,\ 0,\ (x,y) \texttt{->} x+y)\\
\texttt{stg1ShortV} &= \texttt{annt}(\texttt{wndShortV}, x \texttt{->} x,\\
&\qquad (x,y) \texttt{->} x[ShrtV := c]).
\end{aligned}$$

The query `shortV` applies a thresholding operator to the output of `blockV`. As before, `shortV` is run in a sliding-window fashion using the `wnd` construct, and the output is annotated onto the stream using `annt`.

The same two computations are performed on the atrial channel, but with a different threshold, T_{afib}, for `stg1ShortA`. The final query for this state is the streaming composition of the above channel-specific computations

$$\begin{aligned}
\texttt{stage1} = \texttt{stg1SumV} &\gg \texttt{stg1ShortV} \gg \texttt{stg1SumA}\\
&\gg \texttt{stg1ShortA}.
\end{aligned}$$

The output stream $\mathbf{s_1}$ of this stage consists of items (with rearrangement) of the following form:

$$\begin{aligned}
s_1 &= (V, I_V, SumV, ShrtV, A, I_A, SumA, ShrtA, t)\\
&\in D_1 = (\mathbb{B}\times\mathbb{N}^3)^2\times\mathbb{N}.
\end{aligned} \tag{7}$$

D. Stage 2: Sudden Onset and Begin Duration

This stage computes the Sudden Onset discriminator and Begin Duration (BD) marker at every ventricular beat. In order to do this, the last 14 ventricular intervals I_1, $I_2,\ldots,I_{14}$ have to be considered, as shown in Fig. 10.

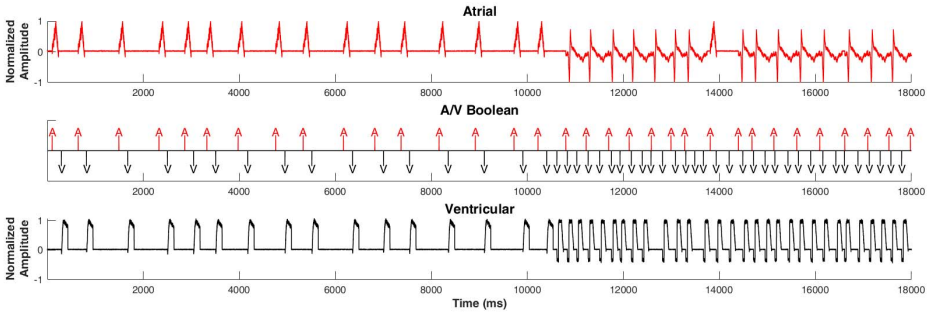

Fig. 11. *EGM during a VF. Top panel shows the atrial EGM. Bottom panel shows the ventricular EGM. The middle panel shows the sensed Boolean signal that is part of the input stream s to the AMA. Spikes above the x-axis indicate atrial beats, and spikes below it are the ventricular beats.*

- The first four intervals I_1, I_2, I_3 and I_4 are used for Step I of "Sudden Onset," defined in (4).
- The next four intervals I_5, I_6, I_7 and I_8 are used for Step II of "Sudden Onset," defined in (5).
- The intervals I_5, I_6, I_7 are used for the "Three Consecutive Intervals" discriminator, defined in (2).
- The last ten intervals I_5, I_6,..., I_{14} are used for the "8/10 Short Intervals" discriminator, defined in (3).

This stage splits the stream into consecutive intervals, and evaluates all the relevant discriminators over the last 14 intervals using the operation $opStage2 : \mathbb{N}^{14} \rightarrow \mathbb{B} \times \mathbb{B}$. The input to $opStage2$ is a vector of 14 ventricular interval lengths, and the output is a pair of Boolean values: the first component indicates the presence of "Sudden Onset" (SO), and the second component indicates the presence of BD.

$$\begin{aligned} \texttt{sobd} &= \texttt{split}(\texttt{blockV}, \ldots, \texttt{blockV}, \\ &\qquad (x_1, \ldots, x_{14}) -> opStage2(x_1, \ldots, x_{14})) \\ \texttt{wndsobd} &= \texttt{split}(\texttt{match}(\mathsf{R}(\texttt{blockV})^*), \texttt{sobd}, \pi_2) \\ \texttt{stage2} &= \texttt{annt}(\texttt{wndsobd}, x -> x, \\ &\qquad (x, \langle c_1, c_2 \rangle) -> x[SO := c_1, BD := c_2]). \end{aligned}$$

The query $\texttt{sobd} : \texttt{QRE}\langle D_1, \mathbb{B}^2 \rangle$ matches 14 consecutive ventricular intervals, and applies the function *opStage2* to their lengths in order to compute the Boolean flags for "Sudden Onset" and "Begin Duration." This computation is executed in a sliding-window fashion and the output is used to annotate the stream. The output stream $\mathbf{s_2}$ from Stage 2 contains data items of the following form:

$$s_2 = (s_1, SO, BD) \in D_2 = D_1 \times \mathbb{B}^2.$$

E. Stage 3: Therapy Decision

This stage uses the four discriminators shown in Fig. 10 to make the final decision whether to apply therapy or not. Whenever BD is detected by the previous stage, the algorithm considers the window of N data items following BD, and the discriminators are computed using the information contained within this window. For example, if the Duration window is programmed to be 5 s, and the sampling rate is 256 Hz, then the window contains $N = 5 \times 256 = 1280$ items. The query

$$\texttt{stage3} = \texttt{wnd}(\texttt{atom}(), N, 0, \texttt{ins}, \texttt{rmv}, \texttt{discr})$$

describes a sliding-window computation that maintains a buffer with all ventricular and atrial beats of the duration period. The function `ins` adds a new item to the buffer, the function `rmv` removes an expiring item from the buffer, and the operation `discr` computes the discriminators and the final therapy decision using only the items contained in the buffer.

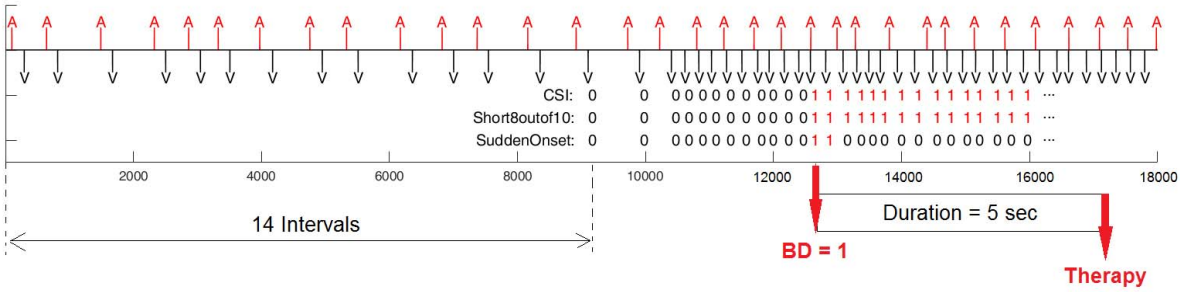

Fig. 12. *Boolean beat stream from Fig. 11 and the streaming output of QRE* `stage2` *(which calculates CSI, Short8outof10 and SuddenOnset).*

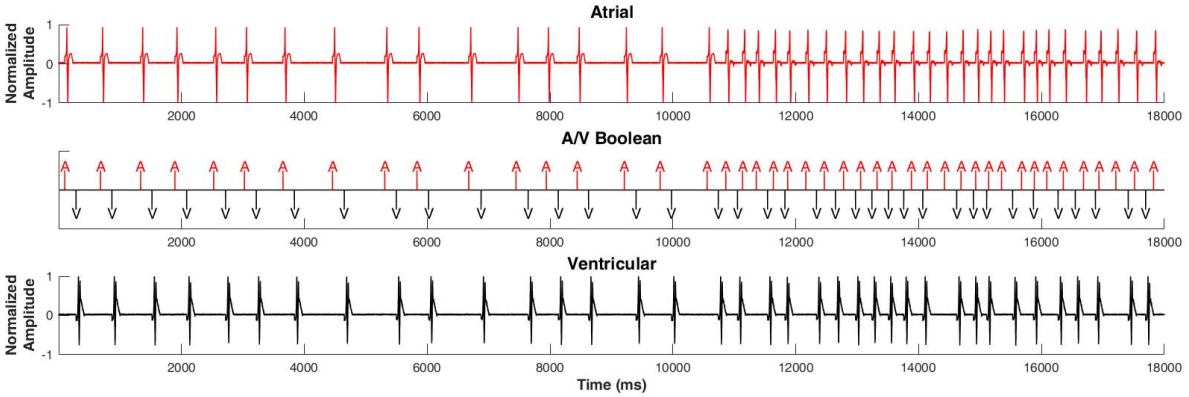

Fig. 13. ***AF EGMs and their Boolean beat streams.***

F. Overall AMA Query

The top-level query for this AMA is the streaming composition of all stages (see Fig. 7)

$$\texttt{AMA} = \texttt{stage0} \gg \texttt{stage1} \gg \texttt{stage2} \gg \texttt{stage3}.$$

VII. ILLUSTRATIVE EXAMPLES

A. Sample Executions

Two examples will serve to illustrate the details of the query execution. Fig. 11 shows a VF EGM signal along with the corresponding Boolean beat stream. The results of running `stage2` on this signal are presented in Fig. 12. At times 12 572 ms and 12 811 ms the start of Duration is detected (BD $= 1$). At the end of the first initiated Duration (at 17572 ms), the A/V Rate Comparison discriminator and Sliding 6/10 discriminator are satisfied and the AMA outputs Therapy. This is consistent with the decision tree in Fig 5.

Fig. 14 shows an AF signal. The algorithm never outputs therapy. Before time 15 529 ms the rhythm is not determined to be fast (Three Consecutive Short Intervals and 8/10 Short Intervals are never satisfied together). The first time when the fast rhythm is detected is at 15 529 ms. Therefore, the first BD $= 1$ flag happens at time 15 529 ms and Duration starts. At the end of this Duration (20 529 ms), A/V Rate Comparison is not satisfied. Moreover, the rhythm is determined to be unstable with gradual onset and AFib Rate condition is satisfied. Therefore, no therapy is delivered at this point. The same thing occurs for the next ventricular beat time point (15 867 ms), and no therapy is detected again.

B. Validation of the QRE Implementation

To validate the correctness of our QRE implementations, we created three versions of the AMA in Fig. 5. These three versions will also be used in the power analysis of Section VIII. The baseline version, presented in Section V, includes all discriminators and has a Duration length of 5 s. The second version does not use the Sudden Onset discriminator. This discriminator is

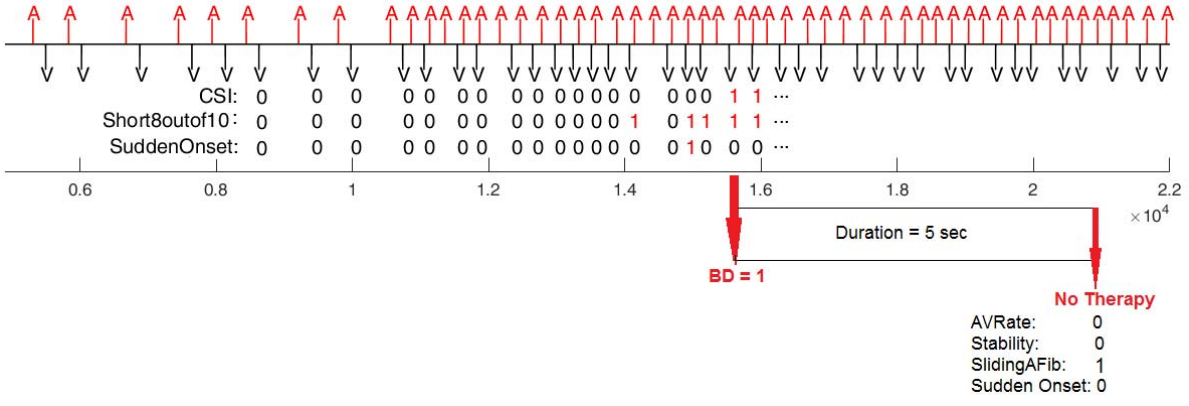

Fig. 14. ***Boolean beat stream from Fig. 13 and the streaming output of QRE*** `stage2`.

off by default when the device ships. The third version reduces Duration length to 1 s. Accuracy is measured using the Specificity and Sensitivity of detection, defined, respectively, as

$$\text{Specificity} = \frac{\#\text{ correctly detected SVTs}}{\#\text{ true SVTs}} \times 100\%$$

$$\text{Sensitivity} = \frac{\#\text{ correctly detected VTs}}{\#\text{ true VTs}} \times 100\%$$

where the denominators are the number of true SVTs and VT, respectively.

The three versions were run on a database of 960 EGMs, equally divided into 480 SVTs and 480 VTs. The beat timing in the EGMs (in other words, the Boolean stream **s**) was generated by the heart model of [37] and [38]. Briefly, this model can simulate beat generation and propagation at different rates, from different locations in the heart; e.g., it can simulate a normal sinus rhythm (NSR) which originates in the sino-atrial node and conducts down, or a fast ventricular rhythm that starts in the ventricles and conducts up to the atria. The model can also simulate different conduction pathways and conduction delays between locations. In this manner, it is capable of simulating a wide range of VTs and SVTs. These simulated arrhythmia episodes are automatically labelled by the model so that we know whether they should be treated by the device or not, thus allowing us to compute specificity and sensitivity.

The validity of the simulated beat stream is guaranteed in three complementary ways: 1) the model implements well-known clinical principles of arrhythmia generation, such as re-entrant circuits [22], and the implementation has been reviewed by two cardiologists; 2) key output stream characteristics, like the rate, are guaranteed to fall in the clinically observed ranges; and finally, 3) a representative sample of model outputs has been validated as correct by two cardiologists.

Table 1 shows the results of running these three versions on the signals database. It also includes throughput, which is the number of data items processed per second. First, we note that the Sensitivity of all three algorithms is 100%, which matches the reported sensitivity of ICDs in the literature. Indeed, missing a true VT or VF can have a debilitating or fatal effect on the patient, so the algorithms are programmed to err on the side of safety and guarantee 100% sensitivity. Second, we note that turning off Sudden Onset has a negligible effect on Specificity, which justifies its being turned off by default in real devices. Finally, shortening Duration further decreases Specificity, as expected: when Duration is shorter, the algorithm is leaving less time for the arrhythmia to terminate on its own, and is taking a Therapy decision for signals that should not be treated.

VIII. UPPER BOUNDS ON QRE COST

Power consumption is an important consideration when designing the software and hardware of an implantable medical device. Replacing an implantable device requires surgery, and most ICD and ILR recipients are older patients with various health issues [28], so reducing the likelihood of a replacement by prolonging battery life is highly desirable [27].

It is generally true that the higher the abstraction level at which power consumption is estimated, then the easier it is to correlate algorithmic changes to power changes and the more questions can be answered analytically. However, the estimates are then less accurate in absolute terms. Conversely, at a lower abstraction level, the power model is more accurate, but is much harder to correlate to algorithmic changes, especially if it is tied to a particular target processor.

In this section, we provide a way to compute an *upper bound* on the energy consumed by a QRE per data item. The per-item consumption is the appropriate unit of measurement since a stream can be arbitrarily long. Being an upper bound, it allows the algorithm developers to compare design options very early on based on worst case cost, and hardware engineers to provision battery capacity and electronics that are suitable for the expected worst case energy draw. The upper bound is obtained by first measuring the per-item energy consumption of all the predicates and *op*s that appear in the QRE. These will be referred to as the *basic costs*. Then the QRE evaluator itself is used to combine these basic costs into the worst case cost of the query, without any further measurements. It is possible to do this for programs written in the StreamQRE language because of the well-understood syntactical restrictions it imposes, in particular, the restriction that computation results cannot be used in predicates. Note that these analyses apply trivially to any other additive cost, such as processing time, and not just power.

The upper-bound energy analysis described in the previous paragraph is meant to provide only a crude estimate of energy consumption for early design space exploration. It is not meant to replace a more fine-grained analysis (such as a WCET analysis) that takes the hardware and the input data into account. Such a high-precision analysis is useful for fine-tuning the performance of a production implementation, but a more rough analysis is still useful in the early design stage.

Table 1 Database-Averaged Detection Accuracy for Three Versions of AMA. Throughput Measured on Standard Desktop With Intel i5 Processor Running Ubuntu

	Algorithm		
Measurements:	Baseline	No Onset	Duration = 1s
Throughput [items/sec]	674.602	714.206	914.746
Sensitivity	100%	100%	100%
Specificity	92.5%	93.13%	88.54%

A. An Upper Bound Based on the Evaluator

We first need to understand roughly how the QRE evaluator works. The evaluator is the algorithm that evaluates a QRE on a given stream. For a query q, the evaluator first invokes a query-specific `start` routine to initialize the internal data structures appropriately. With every new data item that arrives, the evaluator invokes a query-specific `next` routine to process it. Moreover, `next` might have to pass the item to subqueries: e.g., $\texttt{split}(\texttt{f},\texttt{g})$ will pass the item to g every time the string seen so far matches f. In such a case, `next` will need to invoke the `start` method of g. Therefore, the cost of processing a data item is the cost of calling the QRE's `next` *routine*.

1) From Basic Cost to QRE Cost: Let $cost(\varphi)$ and $cost(op)$ be the cost of evaluating the predicate φ and operation op respectively. It is assumed that these costs are data-independent, which is true for the queries that appear in `AMA`. Let $start(\texttt{q})$ and $next(\texttt{q})$ be functions that return the cost of executing `start` and `next` methods of query q. The per-item cost of a QRE q can be upper-bounded using the following recursion on its structure:

$$
\begin{aligned}
&\texttt{q} = \texttt{atom}(\varphi, \texttt{op}):\\
&\quad start(\texttt{q}) = 0\\
&\quad next(\texttt{q}) = cost(\varphi) + cost(\texttt{op})\\
&\texttt{q} = \texttt{split}(\texttt{f}, \texttt{g}, \texttt{op}):\\
&\quad start(\texttt{q}) = start(\texttt{f}) + start(\texttt{g}) + cost(\texttt{op})\\
&\quad next(\texttt{q}) = next(\texttt{f}) + next(\texttt{g}) + start(\texttt{g}) + cost(\texttt{op})\\
&\texttt{q} = \texttt{iter}(\texttt{f}, init, \texttt{op}, \texttt{out}):\\
&\quad start(\texttt{q}) = start(\texttt{f}) + cost(\texttt{out})\\
&\quad next(\texttt{q}) = next(\texttt{f}) + cost(\texttt{op}) + start(\texttt{f}) + cost(\texttt{out})\\
&\texttt{q} = \texttt{iterLast}(\texttt{f}):\\
&\quad start(\texttt{q}) = start(\texttt{f})\\
&\quad next(\texttt{q}) = next(\texttt{f}) + start(\texttt{f})\\
&\texttt{q} = \texttt{iterUntil}(\varphi, \psi, init, \texttt{op}):\\
&\quad start(\texttt{q}) = 0\\
&\quad next(\texttt{q}) = cost(\varphi) + cost(\psi) + cost(\texttt{op})\\
&\texttt{q} = \texttt{wnd}(\texttt{f}, size, init, \texttt{ins}, \texttt{rmv}, \texttt{out}):\\
&\quad start(\texttt{q}) = start(\texttt{f})\\
&\quad next(\texttt{q}) = next(\texttt{f}) + cost(\texttt{ins}) + cost(\texttt{rmv}) +\\
&\qquad\qquad cost(\texttt{out}) + start(\texttt{f})\\
&\texttt{q} = \texttt{annt}(\texttt{f}, \texttt{op}_1, \texttt{op}_2):\\
&\quad start(\texttt{q}) = start(\texttt{f})\\
&\quad next(\texttt{q}) = next(\texttt{f}) + \max(cost(\texttt{op}_1), cost(\texttt{op}_2))\\
&\texttt{q} = \texttt{f} \gg \texttt{g}:\\
&\quad start(\texttt{q}) = start(\texttt{f}) + start(\texttt{g}) + next(\texttt{g})\\
&\quad next(\texttt{q}) = next(\texttt{f}) + next(\texttt{g}).
\end{aligned}
$$

To understand this recursion, consider the case $\texttt{q} = \texttt{atom}(\varphi, op)$. Starting the evaluator does not cost anything in this case. When the data item arrives and it matches φ, then op is executed and we pay $cost(\varphi) + cost(op)$. Otherwise, we only pay $cost(\varphi)$. Thus an upper-bound on cost is $cost(\varphi) + cost(op)$, as indicated.

For a more involved example, consider the case $\texttt{q} = \texttt{split}(\texttt{f}, \texttt{g}, op)$. `start`-ing q involves `start`-ing f and g, and we pay the corresponding costs. If both of them match the empty string, then we also pay $cost(op)$. So worst case cost of `start` is as shown. When a data item arrives, it is passed to both f and g: f might match the string in multiple positions, and it is not possible to know ahead of time which will be the right split point, so the string is always fed to f, and we pay $next(\texttt{f})$. If the string seen so far matches f then the item is also passed to g to see if the string suffix will match it, and we pay $start(\texttt{g})$. g might also be in the middle of matching a previous suffix (remember the evaluator maintains all possible matches). In that case, it will also process the new item using its `next` routine, and we pay $next(\texttt{g})$. Finally, if both f and g match, then $op([\![\texttt{f}]\!]w, [\![\texttt{g}]\!]w)$ is evaluated and we pay $cost(op)$. Thus in the worst case, the cost of $\texttt{next}(\texttt{q})$ is $next(\texttt{f}) + next(\texttt{g}) + start(\texttt{g}) + cost(op)$, as shown.

2) Measuring the Basic Costs: To start the above recursion, we need knowledge of $cost(\varphi)$ and $cost(op)$. For example, consider query `stage3` defined in Section VI-E and its associated costs:

$$
\begin{aligned}
\texttt{stage3} &= \texttt{wnd}(\texttt{atom}(), N, 0, \texttt{ins}, \texttt{rmv}, \texttt{discr})\\
start(\texttt{stage3}) &= start(\texttt{atom}()) = 0\\
next(\texttt{stage3}) &= next(\texttt{atom}()) + cost(\texttt{ins}) + cost(\texttt{rmv})\\
&\quad + cost(\texttt{discr}) + start(\texttt{atom}())\\
&= cost(x \dashrightarrow True) + cost(x \dashrightarrow x)\\
&\quad + cost(\texttt{ins}) + cost(\texttt{rmv}) + cost(\texttt{discr}).
\end{aligned}
$$

Therefore, it is necessary to measure the following:

$$
\begin{aligned}
C_1 &= cost(x \dashrightarrow True)\\
C_2 &= cost(x \dashrightarrow x)\\
C_3 &= cost(\texttt{ins}) + cost(\texttt{rmv}) + cost(\texttt{discr}).
\end{aligned}
$$

The costs of predicates and *op*s can be measured using jRAPL [39] for example. jRAPL provides a mean to measure the energy consumption of any snippet of Java code by enclosing it between `getEnergyStats` function calls. The `getEnergyStats` function accesses Machine-Specific Registers (MSRs) that store the energy consumed since a predefined datum. Thus, we can measure the energy consumed by a given piece of code by comparing the register contents before and after invoking that code,

Table 2 jRAPL-Reported Values for Basic Costs (Obtained by Averaging Over 20M Execution of Operation (= 1 Experiment), and Over 125 Experiments After 25-Experiment Warm Up

	Basic operation	
Measurements:	x-> True	x -> x
DRAM [J·e−5]	0.000003244	0.00004134
CPU [J·e−5]	0.0000045418	0.0001023086
Package [J·e−5]	0.00001155	0.0002180939
Total [J·e−5]	0.000019336	0.000361745

Table 3 jRAPL-Reported Energy Values for $C_3 = cost(\texttt{ins}) + cost(\texttt{rmv}) + cost(\texttt{discr})$, for Three Versions of AMA. Obtained as Average of 100 Experiments After 25-Experiment Warm Up, Each Experiment Having 1M Runs

Measurements:	Baseline	No Onset	Duration = 1s
DRAM [J·e−5]	0.55755	0.54573	0.182398
CPU [J·e−5]	0.549273	0.502224	0.23123
Package [J·e−5]	2.2693036	2.21987	0.726192
Total [J·e−5]	3.376137	3.267839	1.13983

e.g. as shown in the following:

```
EnergyCheckUtils ec = new
  EnergyCheckUtils();
double[] before = ec.getEnergyStats();
long duration =
  Queries.execute(streamlength, stream,
  myquery); //nano-sec
double[] after = ec.getEnergyStats();
double[] energy = after - before;
System.out.println("Consumed energy = "
  + energy).
```

Internally, jRAPL is a Java wrapper around the RAPL library. Running average power limit (RAPL) is a suite of low-level interfaces to the MSRs with the ability to monitor and control energy and power consumption of different hardware levels, widely supported in Intel architectures. RAPL allows energy/power consumption to be reported separately from the CPU core, package (L3 cache, on-chip GPUs, and interconnects), and DRAM.

For the example of `stage3`, Table 2 shows the energy values C_1 and C_2 reported by jRAPL. These operations are extremely cheap and their measurement can be nondeterministically affected by irrelevant processes running on the hardware (like page swaps), compiler optimizations (like discarding of unused outputs, which is why in the code listing above we print out `duration`). To account for this variability, we compute cost by running the same operation 20M times and averaging the energy over the runs. We call this an experiment. We run 125 such experiments in a row, and discard the first 25 experiments to take into account background noise caused by the warm up, and average the last 100 experiments. The final reported number is then the energy per predicate or op.

Table 3 shows the energy value C_3, when running as part of the three versions of AMA described in Section VII-B: the Baseline algorithm, version with no Sudden Onset discriminator, and version with a Duration of 1sec. The energy consumption of `discr` depends on which algorithm it is running in because, for example, a shorter Duration implies that `discr` is operating on fewer items, while no Onset means that the value of Sudden Onset is not used in the decision making of `discr`.

Equipped with these numbers we can upper-bound the per-item energy consumption of `stage3` by $C_1 + C_2 + C_3$. On this basis, the Duration= 1s version is the cheapest in the worst case, and Baseline is the most expensive. On the other hand, No Onset has a per-item cost which is only slightly smaller than that of Baseline. This can be explained by the fact that Baseline only performs one extra AND relative to No Onset, which is a cheap operation (and even that is sometimes not executed, depending on the ordering of arguments). The fact that disabling Onset does not yield meaningful energy savings suggests that for patients that might benefit from Sudden Onset discrimination (like patients who have low frequency of SVTs), it can be enabled without any loss in device longevity.

B. Measured Energy Consumption of Entire Algorithm

We also measured the energy consumption of the QREs on a typical workload, namely, when processing the signals in the EGM database. The three versions of `AMA` and the signals database were described in Section VII. The energy is measured again using jRAPL. Because the AMA is a sufficiently costly operation whose energy measurements will not vary significantly between repeated runs, each experiment consists of a single run of the QRE on the database of signals. We still run and discard some initial warm-up experiments.

Table 4 reports the per-item energy consumption, averaged over the signals in the database. The energy numbers match expectations: the baseline version consumes the most energy. Version No Onset is second most expensive, because eliminating Sudden Onset reduces the costs of Stage 2 (which computes the Onset decision SO—see Fig. 10 and QRE `stage2`), and Stage 3 (which uses the SO value in an AND statement). Finally, Shortening

Table 4 Database-Averaged jRAPL-Reported Energy for Three Versions of Entire Algorithm. Obtained as Average of 40 Experiments After Warm-Up of 10 Experiments, Each Experiment Having 1 Run

	Algorithm		
Measurements:	Baseline	No Onset	Duration = 1s
DRAM [J·e−5]	1.7177	1.5836	1.1813
CPU [J·e−5]	2.7648	2.0992	1.6067
Package [J·e−5]	6.7009	5.9251	4.4322
Total [J·e−5]	11.1834	9.6078	7.2203

Duration saves the most energy, since it implies shorter computations for four discriminators.

C. Validating Cost Rankings

The above analysis depends on the assumption that the ranking of queries on the basis of the upper bound will, by and large, match the ranking we obtain for them on the basis of actual measured energy consumption. To test this assumption, we wrote a program that generates arbitrary queries (StreamQRE programs) by arbitrarily composing the `split`, `iter` and `combine` combinators. Using this program we generated 2000+ queries. For each query we generated a random set of input strings over which to measure the actual (per-item) consumed energy using jRAPL. The basic operations were made sufficiently expensive to eliminate measurement variability due to background noise. For each QRE, we also computed the upper bounds. The 2000+ queries were ordered by upper bounds and by measurements.

To compare the two rankings, we use Spearman's Rank-Order Correlation, a standard statistic that measures the strength and direction of a monotonic relationship between two ordinal variables. A correlation close to 1 indicates a strong correlation, i.e., that the two *rankings* are indeed highly correlated. The two rankings of 2000+ QREs had a correlation of 0.959, which validates our claim about the usefulness of the upper bounds for these types of programs.

IX. RELATED WORK

A. Medical Device Algorithms

Most of the literature on formal methods for medical device algorithms focuses on verifying and testing the functionality of the algorithm, see [40]–[43] for examples in the specific context of implantable cardiac devices. These concerns are orthogonal to ours; the focus of this paper is the description of a *programming language* that is suitable for arrhythmia monitoring, and the *meta-functional* characteristics it automatically guarantees. It is worth nothing that the U.S. Food and Drug Administration (FDA), which regulates medical devices in the U.S., does not mandate particular types of validation, such as model checking [44]. Rather, it describes in generic terms the kind of evidence that should be provided. For example, it stipulates that "Software quality assurance needs to focus on preventing the introduction of defects into the software development process," and that "software developers should use a mixture of methods and techniques to prevent software errors and to detect software errors that do occur." [44, Sec. 4.2].

The FDA Guidance does not explicitly address meta-functional properties. Works in quantitative verification, such as [45] and [46], model the heart and pacemaker to verify statistically or through simulations whether some quantitative properties are satisfied. This contrasts with our approach which is model-free and provides cost upper bounds based on the QRE code itself, not a model of it. An application of QREs to arrhythmia monitoring appeared in [26] where a peak detector is coded in an early variant of the language.

QREs are a DSL; they are meant for programming queries on arbitrary data streams, with strong theoretical foundations [5] and a flexible programming environment [6], [18]. DSLs have been developed for medical device development, albeit these are usually meant for the creation of the entire device, including hardware, and focus on capturing object-oriented aspects of the domain (i.e., identifying the main objects in the domain and modeling them and their relations); e.g., [47] develops a graphical language for modeling blood separator machines, along with code generators and lock-step simulators of the model and its generated code. No work has appeared in the literature on a DSL for ICDs or ILR algorithms, and more generally, rhythm monitoring algorithms.

B. Streaming Languages

There is a large body of work on streaming database languages and systems such as Aurora [48], Borealis [49], STREAM [50], and StreamInsight [51], [52]. The query language supported by these systems (for example, CQL [53]) is typically a version of SQL with additional constructs for splitting the stream into finite windows (e.g., tumbling or sliding windows, count-based or time-based). This allows for rich relational queries, including set-aggregations (e.g. sum, maximum, minimum, average, count) and joins over multiple data streams. Such SQL-based languages are, however, limited in their ability to express properties and computations that rely on the sequence of the events such as: sequence-based pattern-matching, and numerical computation based on list-iteration when the order of the data items is significant. There are streaming engines such as IBM's Stream Processing Language (SPL) [54], [55], ReactiveX [56], Esper [57] and Flink [58], which support user-defined types and operations, and allow for both relational and stateful sequential computation. However, none of these engines provides support for decomposing the stream in a regular fashion and performing incremental computations that reflect the structure of the parse tree, which is a central feature of the QRE language. LOLA [59] allows arbitrary computations on streams and incremental computation of statistics, but does not support regular decomposition of the stream to define the computation domains. Finally, Timed Regular Expressions [60] allow the specification of time windows during which the timed string must match a regular expression. As such they are a specification language rather than a programming language and do not support the rich computations and quantitative combinators that QREs support.

C. Power Estimation

Since QREs are aimed at high-level programming and the cost analysis is aimed at early design exploration, we do no review the vast literature on low-level power estimation techniques (anything below C program level), nor do we review analyses that focus on the impact of particular hardware choices like [61]. Such analyses occur later in the design cycle and require the availability of low-level artifacts like circuits. The interested reader can consult [62] for a recent review of such techniques.

In [62]–[64], a *functional level power model* is used for estimating the power consumption of a C program without compiling it. It requires partitioning the target processor into functional units, and estimating some key parameters like the cache miss rate, external data memory access rate and the processing rate. It also depends on the user providing low-level execution details like the data mapping. This target-specific code-level analysis complements our presented bounds, which occur earlier in the design cycle and are at the algorithm level.

The approach in [65] estimates battery dissipation. It treats the processor as a black box and instead decomposes *the program* into types of basic instructions, similar to what we did to obtain the upper bounds in Section VIII. However, the basic instructions in [65] are at the instruction-set level, like integer and floating point loads and stores. And while we exploit the fact that we have a uniform evaluation algorithm for any QRE to infer bounds on the entire program's cost, the authors in [65] must establish empirically, for a given processor and program, that the program's cost is the weighted sum of the dissipations for basic instructions.

A static analysis of energy consumption of XC programs is presented in [66].[4] After building an ISA-level power model using hardware measurements of a test suite, the XC program is translated to Horn clauses in the Ciao programming language [67]. The Ciao preprocessor can then bound the power consumption as a function of input data sizes. This technique was later extended to use a power model at the level of the compiler's intermediate representation rather than the ISA level [68]. This approach applies to programs that can be translated into a logic program. Another static analysis technique [69] uses integer linear programming to compute the worst case energy consumption, given estimates of dynamic and leakage power contributions of basic blocks in a program's control flow graph. This is inspired by well-known Worst Case Execution Time estimation techniques.

X. CONCLUSION

This paper has argued that arrhythmia monitoring algorithms are best viewed as streaming algorithms, and that they are best programmed in the StreamQRE language. Unlike traditional streaming applications where throughput is a prime concern, here energy consumption is the primary design factor. A program written in StreamQRE automatically gets a baseline implementation with a constant memory, processing time and energy consumption per item. Moreover, the QRE evaluator automatically provides upper bounds on the per-item cost of the query, which can be used early in the design cycle to guide the choice of algorithm, and to decide whether some discriminators are worth having at all. We showed how the StreamQRE Java Library can be used to program and evaluate a query and to obtain cost upper bounds, and how these bounds correlate to actual power measurements. We believe this approach to exploring and programming arrhythmia monitors, and other medical device algorithms, has the potential to greatly alleviate the device development burden. In particular, it opens the possibility of designing ILR algorithms that collect statistics over longer time durations than is currently done. Other applications that might benefit from StreamQRE include glucose monitoring [70], [71], where a mobile device periodically or continuously measures a diabetic's blood glucose and performs various filtering operations to predict hypo- or hyperglycemic episodes.

The theoretical basis of StreamQRE raises the possibility of performing static (formal) analysis of its performance. It is already possible, for queries written in a subset of the language, to answer questions such as "Does the worst case peak power consumed by the algorithm exceed some threshold?," "Could the long-term average power consumed by the algorithm exceed some threshold?," and "Does algorithm A consume less peak/average power than algorithm B?" Answers to these questions impact the choice of electronics that must withstand the peak power draw, and the capacity of the device battery.

On the tools side, two projects are worth exploring: first, implementing the decision procedures that perform the above-described static analysis. Second, creating a compiler that compiles a QRE into C or assembly code targeting a given hardware platform. This would complete the path from algorithm to code to implementation, and would allow a reliable comparison of upper-bounds to actual energy consumption of the compiled code. In niche areas, expert coders might be able to squeeze more performance per Watt from hand-written code than a compiler could from automatically generated code. However, it is to be expected in the long run that medical devices will follow the arc of semiconductors, where automation has gradually outperformed humans, or has yielded such productivity gains that small performance losses are more than made up for by the reduced time-to-market, reproducibility and scalability of the design process, and automatic guarantees of correctness and performance. ■

[4]XC is a high-level C-based programming language.

REFERENCES

[1] E. A. Lee, "What's ahead for embedded software?" *Computer*, vol. 33, no. 9, pp. 18–26, 2000.

[2] T. Henzinger and J. Sifakis, "The embedded systems design challenge," in *Proc. 14th Int. Symp. Formal Methods*, 2006, pp. 1–15.

[3] A. Sangiovanni-Vincentelli, "Quo vadis, SLD? Reasoning about the trends and challenges of system level design," *Proc. IEEE*, vol. 95, no. 3, pp. 467–506, Mar. 2007.

[4] R. Alur, *Principles of Cyber-Physical Systems*. Cambridge, MA, USA: MIT Press, 2015.

[5] R. Alur, D. Fisman, and M. Raghothaman, "Regular programming for quantitative properties of data streams," in *Proc. 25th Eur. Symp. Program. Lang. Syst.*, 2016, pp. 15–40.

[6] K. Mamouras, M. Raghothaman, R. Alur, Z. G. Ives, and S. Khanna, "StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data," in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Des. Implement.*, 2017, pp. 693–708.

[7] S. Chintapalli, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2016, pp. 1789–1792.

[8] P. Tucker, K. Tufte, V. Papadimos, and D. Maier (2002). "*NEXMark: A benchmark for queries over data streams*." [Online]. Available: http://datalab.cs.pdx.edu/niagara/NEXMark/

[9] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo, "Quantitative network monitoring with NetQRE," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2017, pp. 99–112.

[10] I. Lee, "Challenges and research directions in medical cyber–physical systems," *Proc. IEEE*, vol. 100, no. 1, pp. 75–90, Jan. 2012.

[11] P. Ye, E. Entcheva, R. Grosu, and S. A. Smolka, "Efficient modeling of excitable cells using hybrid automata," in *Proc. Comput. Methods Syst. Biol.*, 2005, pp. 216–227.

[12] Z. Jiang, M. Pajic, S. Moarref, R. Alur, and R. Mangharam, "Modeling and verification of a dual chamber implantable pacemaker," in *Proc. 18th Int. Conf. Tools Algorithms Construct. Anal. Syst.* Springer, 2012, pp. 188–203.

[13] T. Chen, M. Diciolla, M. Kwiatkowska, and A. Mereacre, "Quantitative verification of implantable cardiac pacemakers over hybrid heart models," *Inf. Comput.*, vol. 236, pp. 87–101, Aug. 2014.

[14] R. Alur and D. L. Dill, "A theory of timed automata," *Theory Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[15] R. Alur, "The algorithmic analysis of hybrid systems," *Theor. Comput. Sci.*, vol. 138, no. 1, pp. 3–34, 1995.

[16] G. Behrmann, A. David, K. Larsen, P. Pettersson, and W. Yi, "Developing UPPAAL over 15 years," *Softw. Pract. Exper.*, vol. 41, no. 2, pp. 133–142, 2011.

[17] *The Compass—Technical Guide to Boston Scientific Cardiac Rhythm Management Products*, Boston Sci. Corp., Marlborough, MA, USA, 2007.

[18] StreamQRE Library. [Online]. Available:http://www.seas.upenn.edu/∼amouras/StreamQRE/StreamQRE.jar

[19] S. Krishnamoorthi, "Simulation methods and validation criteria for modeling cardiac ventricular electrophysiology," *PLoS ONE*, vol. 9, no. 12, p. e114494, Dec. 2014.

[20] A. Thammanomai, M. O. Sweeney, and S. R. Eisenberg, "A comparison of the output characteristics of several implantable cardioverter-defibrillators," *Heart Rhythm*, vol. 3, no. 9, pp. 1053–1059, 2017, doi: 10.1016/j.hrthm.2006.05.006.

[21] M. Rosenqvist, T. Beyer, M. Block, K. D. Dulk, J. Minten, and F. Lindemans, "Adverse events with transvenous implantable cardioverter-defibrillators: A prospective multicenter study," *Circulation*, vol. 98, no. 7, pp. 663–670, 1998.

[22] K. Ellenbogen, G. N. Kay, C.-P. Lau, and B. L. Wilkoff, *Clinical Cardiac Pacing, Defibrillation and Resynchronization Therapy E-Book*. Amsterdam, The Netherlands: Elsevier, 2011.

[23] *BioMonitor Technical Manual*, Biotronik, Berlin, Germany, 2015.

[24] J. M. Nasir, "Predicting determinants of atrial fibrillation or flutter for therapy elucidation in patients at risk for thromboembolic events (PREDATE AF) study," *Heart Rhythm*, vol. 14, no. 7, pp. 955–961, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1547527117304927

[25] W. Amara, "Early detection and treatment of atrial arrhythmias alleviates the arrhythmic burden in paced patients: The SETAM study," *Pacing Clin. Electrophysiol.*, vol. 40, no. 5, pp. 527–536, 2017, doi: 10.1111/pace.13062.

[26] H. Abbas, A. Rodionova, E. Bartocci, S. A. Smolka, and R. Grosu, "Quantitative regular expressions for arrhythmia detection algorithms," in *Computational Methods in Systems Biology*. Cham, Switzerland: Springer, 2017, pp. 23–39, doi: 10.1007/978-3-319-67471-1_2.

[27] R. D. Berger, "A novel strategy to mitigate ICD shock-related pain," in *Heart Rhythm Scientific Sessions*. 2017.

[28] F. A. Masoudi, "Longitudinal study of implantable cardioverter-defibrillators," *Circulat., Cardiovascular Quality Outcomes*, vol. 5, no. 6, pp. e78–e85, 2012. [Online]. Available: http://circoutcomes.ahajournals.org/content/5/6/e78

[29] G. Boriani, "Battery drain in daily practice and medium-term projections on longevity of cardioverter-defibrillators: An analysis from a remote monitoring database," *EP Europace*, vol. 18, no. 9, pp. 1366–1373, 2016, doi: 10.1093/europace/euv436.

[30] J. Kärnä, J.-P. Tolvanen, and S. Kelly, "Evaluating the use of domain-specific modeling in practice," in *Proc. OOPSLA Workshop Domain-Specific Modeling*, 2009. [Online]. Available: http://www.dsmforum.org/events/dsm09/papers/karna.pdf

[31] J.-P. Tolvanen, N. Brouwers, R. Hendriksen, G. Kahraman, and J. Kouwer, "Industrial use of domain-specific modeling: Panel summary," in *Proc. Domain Specific Modeling Workshop*, Amsterdam, Netherlands, 2016.

[32] *Java's Lambda Expressions*. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

[33] M. Veanes, P. De Halleux, and N. Tillmann, "Rex: Symbolic regular expression explorer," in *Proc. 3rd Int. Conf. Softw. Testing, Verification Validation (ICST)*, Apr. 2010, pp. 498–507.

[34] R. Book, S. Even, S. Greibach, and G. Ott, "Ambiguity in graphs and expressions," *IEEE Trans. Comput.*, vol. C-20, no. 2, pp. 149–153, Feb. 1971.

[35] R. Alur, D. Fisman, and M. Raghothaman, "Regular programming for quantitative properties of data streams," in *Proc. 25th Eur. Symp. Program. (ESOP)*, 2016, pp. 15–40, doi: 10.1007/978-3-662-49498-1_2.

[36] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "Semantics and evaluation techniques for window aggregates in data streams," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2005, pp. 311–322.

[37] H. Abbas, Z. Jiang, K. J. Jang, M. Beccani, J. Liangy, and R. Mangharam, "High-level modeling for computer-aided clinical trials of medical devices," in *Proc. IEEE Int. High Level Des. Validation Test Workshop (HLDVT)*, Oct. 2016, pp. 85–92.

[38] Z. Jiang, "In-silico pre-clinical trials for implantable cardioverter defibrillators," in *Proc. 38th Annu. Int. Conf. IEEE Eng. Med. Biol. Soc. (EMBC)*, Aug. 2016, pp. 169–172.

[39] K. Liu, G. Pinto, and Y. D. Liu, "Data-oriented characterization of application-level energy optimization," in *Proc. Fundam. Approaches Softw. Eng.*, Apr. 2015, doi: 10.1007/978-3-662-46675-9_21.

[40] Z. Jiang, M. Pajic, R. Alur, and R. Mangharam, "Closed-loop verification of medical devices with model abstraction and refinement," *Int. J. Softw. Tools Technol. Transf.*, vol. 16, no. 2, pp. 191–213, 2014, doi: 10.1007/s10009-013-0289-7.

[41] H. Abbas, K. J. Jiang, Z. Jiang, and R. Mangharam, "Towards model checking of implantable cardioverter defibrillators," in *Proc. 19th Int. Conf. Hybrid Syst., Comput. Control*, New York, NY, USA, 2016, pp. 87–92, doi: 10.1145/2883817.2883841.

[42] L. A. Tuan, M. C. Zheng, and Q. T. Tho, "Modeling and verification of safety critical systems: A case study on pacemaker," in *Proc. 4th Int. Conf. Secure Softw. Integr. Rel. Improvement*, Jun. 2010, pp. 23–32.

[43] S. Andalam, H. Ramanna, A. Malik, P. Roop, N. Patel, and M. L. Trew, "Hybrid automata models of cardiac ventricular electrophysiology for real-time computational applications," in *Proc. 38th Annu. Int. Conf. IEEE Eng. Med. Biol. Soc. (EMBC)*, Aug. 2016, pp. 5595–5598.

[44] *General Principles of Software Validation; Final Guidance for Industry and FDA Staff*, document, Center for Devices Radiological Health, 2002.

[45] T. Chen, M. Diciolla, M. Kwiatkowska, and A. Mereacre, "Quantitative verification of implantable cardiac pacemakers over hybrid heart models," *Inf. Comput.*, vol. 236, pp. 87–101, Aug. 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0890540114000157

[46] C. Barker, M. Kwiatkowska, A. Mereacre, N. Paoletti, and A. Patane, "Hardware-in-the-loop simulation and energy optimization of cardiac pacemakers," in *Proc. 37th Annu. Int. Conf. IEEE Eng. Med. Biol. Soc. (EMBC)*, Aug. 2015, pp. 7188–7191.

[47] J.-P. Tolvanen, V. Djukić, and A. Popovic, "Metamodeling for medical devices: Code generation, model-debugging and run-time synchronization," *Proceedia Comput. Sci.*, vol. 63, pp. 539–544, Jan. 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S187705091502517X

[48] D. J. Abadi, "Aurora: A new model and architecture for data stream management," *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003.

[49] D. J. Abadi, "The design of the Borealis stream processing engine," in *Proc. 2nd Biennial Conf. Innov. Data Syst. Res. (CIDR)*, 2005, pp. 277–289. [Online]. Available: http://cidrdb.org/cidr2005/papers/P23.pdf

[50] A. Arasu, "STREAM: The Stanford data stream management system," Stanford InfoLab, Tech. Rep. 2004-20, 2004. [Online]. Available: http://ilpubs.stanford.edu:8090/641/

[51] R. S. Barga, J. Goldstein, M. Ali, and M. Hong, "Consistent streaming through time: A vision for event stream processing," in *Proc. 3rd Biennial Conf. Innov. Data Syst. Res. (CIDR)*, 2007, pp. 363–374. [Online]. Available: http://cidrdb.org/cidr2007/papers/cidr07p42.pdf

[52] M. Ali, B. Chandramouli, J. Goldstein, and R. Schindlauer, "The extensibility framework in Microsoft Streaminsight," in *Proc. 27th IEEE Int. Conf. Data Eng. (ICDE)*, Apr. 2011, pp. 1242–1253, doi: 10.1109/ICDE.2011.5767878.

[53] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: Semantic foundations and query execution," *VLDB J.*, vol. 15, no. 2, pp. 121–142, 2006, doi: 10.1007/s00778-004-0147-z.

[54] M. Hirzel, "IBM streams processing language: Analyzing big data in motion," *IBM J. Res. Develop.*, vol. 57, nos. 3–4, pp. 7:1–7:11, 2013.

[55] M. Vaziri, O. Tardieu, R. Rabbah, P. Suter, and M. Hirzel, "Stream processing with a spreadsheet," in *Proc. 28th Eur. Conf. Object-Oriented Program. (ECOOP)*, 2014, pp. 360–384, doi: 10.1007/978-3-662-44202-9_15.

[56] ReactiveX: An API for Asynchronous Programming With Observable Streams. [Online]. Available: http://reactivex.io/

[57] Esper for Java. [Online]. Available: http://www.espertech.com/esper/

[58] Apache Flink: Scalable Batch and Stream Data Processing. [Online]. Available: https://flink.apache.org/

[59] B. D'Angelo, "Lola: Runtime monitoring of synchronous systems," in *Proc. 12th Int. Symp. Temporal Represent. Reasoning (TIME)*, Jun. 2005, pp. 166–174.

[60] E. Asarin, P. Caspi, and O. Maler, "Timed regular expressions," *J. ACM*, vol. 49, no. 2, pp. 172–206, Mar. 2002, doi: 10.1145/506147.506151.
[61] J. L. Ayala and M. López-Vallejo, "Integrating functional and power simulation in embedded systems design," *J. Embedded Comput.*, vol. 1, no. 3, pp. 325–340, Aug. 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1233748.1233752
[62] M. E. A. Ibrahim, M. Rupp, and H. A. H. Fahmy, "A precise high-level power consumption model for embedded systems software," *EURASIP J. Embedded Syst.*, vol. 2011, pp. 1:1–1:14, Jan. 2011, doi: 10.1155/2011/480805.
[63] E. Senn, N. Julien, J. Laurent, and E. Martin, "Power consumption estimation of a C program for data-intensive applications," in *Proc. 12th Int. Workshop Integr. Circuit Design Power Timing Modeling, Optim. Simulation (PATMOS)*. London, U.K.: Springer-Verlag, 2002, pp. 332–341. [Online]. Available: http://dl.acm.org/citation.cfm?id=646949.712708
[64] H. Blume, D. Becker, L. Rotenberg, M. Botteck, J. Brakensiek, and T. G. Noll, "Hybrid functional- and instruction-level power modeling for embedded and heterogeneous processor architectures," *J. Syst. Archit.*, vol. 53, no. 10, pp. 689–702, Oct. 2007, doi: 10.1016/j.sysarc.2007.01.002.
[65] C. Krintz, Y. Wen, and R. Wolski, "Application-level prediction of battery dissipation," in *Proc. Int. Symp. Low Power Electron. Design*, Aug. 2004, pp. 224–229.
[66] U. Liqat, "Energy consumption analysis of programs based on XMOS ISA-level models," in *Logic-Based Program Synthesis and Transformation*. Cham, Switzerland: Springer, 2014, pp. 72–90, doi: 10.1007/978-3-319-14125-1_5.
[67] The Ciao System. Accessed: Jun. 17, 2017. [Online]. Available: https://ciao-lang.org/
[68] K. Georgiou, S. Kerrison, Z. Chamski, and K. Eder, "Energy transparency for deeply embedded programs," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 1, pp. 8:1–8:26, Mar. 2017. [Online]. Available: http://doi.acm.org/10.1145/3046679
[69] R. Jayaseelan, T. Mitra, and X. Li, "Estimating the worst-case energy consumption of embedded software," in *Proc. 12th IEEE Real-Time Embedded Technol. Appl. Symp.*, Apr. 2006, pp. 81–90.
[70] B. W. Bequette, "Continuous glucose monitoring: Real-time algorithms for calibration, filtering, and alarms," *J. Diabetes Sci. Technol.*, vol. 4, no. 2, pp. 404–418, 2010, doi: 10.1177/193229681000400222.
[71] Y. Leal, "Enhanced algorithm for glucose estimation using the continuous glucose monitoring system," *Med. Sci. Monitor*, vol. 16, no. 6, pp. MT51–MT58, 2010.

ABOUT THE AUTHORS

Houssam Abbas (Member, IEEE) received the B.E. degree in computer engineering from the American University of Beirut, Beirut, Lebanon, and the M.S. and Ph.D. degrees in electrical engineering from Arizona State University, Tempe, AZ, USA.

He was a Design Automation Engineer in the SoC Verification group at Intel from 2006 to 2014. He is currently a Postdoctoral Fellow in the Department of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, PA, USA. His research interests are in the verification, control and conformance testing of cyber–physical systems. His current research includes the verification and performance analysis of life-supporting medical devices, the verification and control of autonomous vehicles with a view towards certifying such systems, and anytime computation and control.

Rajeev Alur (Fellow, IEEE) received the B.S. degree in computer science from IIT Kanpur, India, in 1987, and the Ph.D. degree in computer science from Stanford University, Stanford, CA, USA, in 1991.

He is the Zisman Family Professor of Computer and Information Science at the University of Pennsylvania, Philadelphia, PA, USA. Before joining the University of Pennsylvania in 1997, he was with the Computing Science Research Center, Bell Labs. His research is focused on formal methods for system design, and spans theoretical computer science, software verification and synthesis, and cyber-physical systems. He is the author of the textbook *Principles of Cyber-Physical Systems* (Cambridge, MA, USA: MIT Press, 2015).

Dr. Alur is a Fellow of the American Association for the Advancement of Science (AAAS), a Fellow of the Association for Computing Machinery (ACM), an Alfred P. Sloan Faculty Fellow, and a Simons Investigator. He was awarded the inaugural CAV (Computer-Aided Verification) award in 2008, the ACM/IEEE Logic in Computer Science (LICS) Test-of-Time award in 2010, and the inaugural Alonzo Church award by ACM SIGLOG/EATCS/EACSL in 2016 for his work on timed automata.

Konstantinos Mamouras (Member, IEEE) completed undergraduate studies in electrical and computer engineering at the National Technical University of Athens, Greece, and received the M.Sc. degree in computer science from the Imperial College London, London, U.K., and the Ph.D. degree in computer science from Cornell University, Ithaca, NY, USA.

He is currently a Postdoctoral Researcher in the Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA. His research interests lie in the area of programming languages for data stream processing, and logical approaches for program verification.

Rahul Mangharam (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, USA, in 2000, 2002, and 2008, respectively.

He is an Associate Professor in the Department of Electrical and Systems Engineering as well as the Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA. He is the Director of the Real-Time and Embedded Systems Lab and Mobility21 DoT National University Transportation Center. His current interests are in the application of formal methods, controls and machine learning for safe and efficient life-critical systems in medical devices, energy markets and autonomous systems.

Dr. Mangharam received the U.S. Presidential Early Career Awards for Scientists and Engineers in 2016, DoE CLEANTECH Prize (Regional) in 2016, National Science Foundation CAREER Award in 2014, IEEE Benjamin Franklin Key Award in 2013, Intel Early Faculty Career Award in 2012 and was selected by the National Academy of Engineering for the 2012 and 2017 Frontiers of Engineering.

Alena Rodionova (Member, IEEE) received the B.S. and M.S. degrees in mathematics from the Siberian Federal University, Russia, in 2012 and 2014, respectively. She is currently working toward the Ph.D. degree in the Department of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, PA, USA.

Before joining the University of Pennsylvania in 2017, she was with the Cyber-Physical Systems Group, TU Wien. Her research is focused on formal analysis and verification of safety-critical systems such as medical devices, and risk assessment, verification and control of cyber-physical systems such as autonomous vehicles.