Quantitative Regular Expressions for Arrhythmia Detection

Houssam Abbas*, *Member, IEEE,* Alena Rodionova*, *Student Member, IEEE,* Konstantinos Mamouras, *Member, IEEE,* Ezio Bartocci, Scott A. Smolka *Member, IEEE,* and Radu Grosu, *Member, IEEE*

Abstract—Implantable medical devices are safety-critical systems whose incorrect operation can jeopardize a patient's health, and whose algorithms must meet tight platform constraints like memory consumption and runtime. In particular, we consider here the case of implantable cardioverter defibrillators, where *peak detection* algorithms and various others *discrimination algorithms* serve to distinguish fatal from non-fatal arrhythmias in a cardiac signal. Motivated by the need for powerful formal methods to reason about the performance of arrhythmia detection algorithms, we show how to specify all these algorithms using Quantitative Regular Expressions (QREs). QRE is a formal language to express complex numerical queries over data streams, with provable runtime and memory consumption guarantees. We show that QREs are more suitable than classical temporal logics to express in a concise and easy way a range of peak detectors (in both the time and wavelet domains) and various discriminators at the heart of today's arrhythmia detection devices. The proposed formalization also opens the way to formal analysis and rigorous testing of these detectors' correctness and performance, alleviating the regulatory burden on device developers when modifying their algorithms. We demonstrate the effectiveness of our approach by executing QRE-based monitors on real patient data on which they yield results on par with the results reported in the medical literature.

Index Terms—Peak Detection; Electrocardiograms; Arrythmia Discrimination; ICDs; Quantitative Regular Expressions

1 Introduction

In modern medical devices, signal processing (SP) algorithms are tightly integrated with decision making algorithms, so that the performance and correctness of the latter critically depends on that of the former. Thus, analyzing the device's decision making in isolation of its SP would provide an incomplete picture of the device's overall behavior.

We consider here the specific case of Implantable Cardioverter Defibrillators (ICDs) where a *Peak Detection* (PD) algorithm is first executed on the input voltage signal, known as an *electrogram* (see Fig. 1). The PD algorithm generates, as output, a timed boolean signal where a 1 represents a peak (local extremum) caused by a heartbeat. This boolean output signal is then used by the downstream *discrimination algorithms* to differentiate between fatal and non-fatal rhythms. Around 10% of an ICD's erroneous decisions are due to over-sensing (too many false peaks detected) and under-sensing (too many true peaks missed) [31]. This leads to an inaccurate estimate of both the heart rate and to an imprecise calculation of the timing relations between the contractions of the heart's chambers.

- *H. Abbas and A. Rodionova contributed equally*
- *Houssam Abbas and Alena Rodionova are with the Department of Electrical and Systems Engineering, The University of Pennsylvania, Philadelphia, PA, 19104. E-mail: {habbas,nellro}@seas.upenn.edu, seas.upenn.edu/∼habbas*
- *Konstantinos Mamouras is with the Department of Computer and Information Science, The University of Pennsylvania, Philadelphia, PA, 19104. E-mail: mamouras@seas.upenn.edu*
- *Ezio Bartocci and Radu Grosu are with the Faculty of Informatics, Technical University of Vienna, Vienna, Austria.*
- *Scott A. Smolka is with the Computer Science Department, Stony Brook University, Stony Brook, NY, U.S.A.*

Manuscript received April 19, 2018.

One of the main challenges is how to formally verify the properties of ICD algorithms for cardiac arrhythmia discrimination. In particular there is a *a need for a unified language to express and analyze both the PD and discrimination tasks that are currently at the heart of modern ICD technology.* The desired language should be *formal* (not ambiguous, with a well-defined semantics), expressive enough to be easy to use, and it should guarantee runtime and memory consumption bounds for the resulting programs. One common approach from the computing literature would be to view the ICD tasks as *specification-based monitoring problems* [10], [11]. Namely, one can try to express the PD and discrimination tasks as requirements in some temporal logics [12]: i.e., write a specification which is true exactly when the signal (in a given window) has a peak, and another one which is true exactly when the rhythm (in that window) displays an arrhythmia. Monitor synthesis then automatically translates these specifications into detection algorithms and code.

However, as shown in [4], this approach is impractical for the peak detection and discrimination tasks we are concerned with. Despite the increasingly sophisticated variety of temporal logics proposed in the literature [10], [15], [19], they turn out to be inadequate to express, in one concise formalism, both PD algorithms and arrhythmia discriminators. It is worth noting that PD is a very common signal processing primitive that is employed in many domains beyond medical applications (i.e., earthquake detection), and that arrhythmia discriminators are present in cardiac devices other than ICDs, such as Implantable Loop Recorders and pacemakers. Thus, the observed limitations of temporal logics extend beyond ICD algorithms.

PD and discrimination require performing a wide range

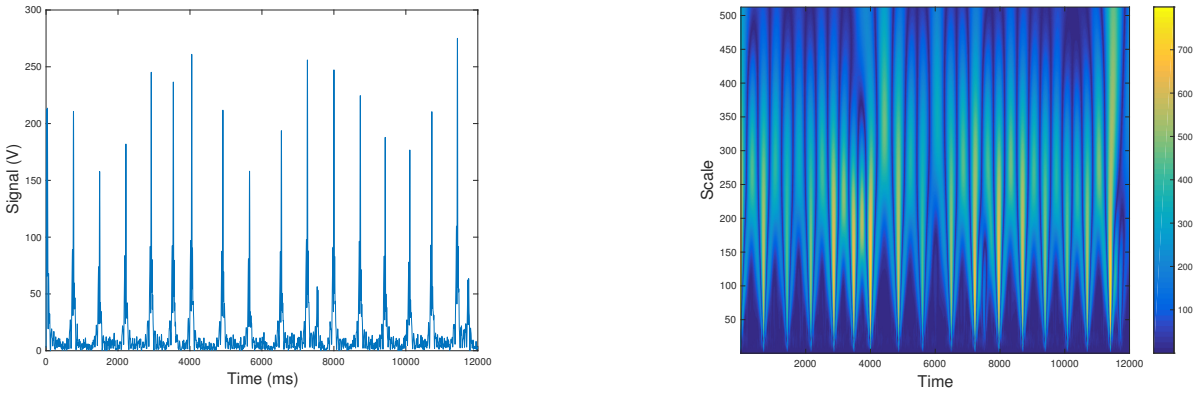

Fig. 1: Rectified EGM during normal rhythm (left) and its CWT spectrogram (right)

of numerical operations over data streams, where the data stream is the incoming cardiac voltage signal (electrogram) observed in real-time. For example, a commercial PD (shown in Section 5) defines a peak as a value that exceeds a certain time-varying threshold, and the threshold is periodically re-initialized as a percentage of the previous peak's value. As another example, the *Onset* discriminator compares the average heart rate in two successive windows. Thus, the desired formalism must enable value storage, time freezing, various arithmetic operations, and nested computations, *while remaining legible and succinct, and enabling compilation into efficient implementations*.

For this purpose, we propose here the use of Quantitative Regular Expressions (QREs) to program ICD tasks for arrhythmia detection. QREs, described in Section 3, are a *declarative* formal language based on classical regular expressions for specifying *numerical queries on data streams* [5]. QREs' ability to interleave user-defined computation at any nesting level of the underlying regular expression gives them significant expressiveness. QREs can also be evaluated in a runtime- and memory-efficient way, which is an important consideration for resource-constrained implanted medical devices.

In [4] we showed the versatility of QREs by encoding peak detection as QRE programs. In particular, we considered three different peak detectors:

1) A Wavelet Peak with Maxima (WPM) detector, which operates in the wavelet domain (Section 2),
2) A Wavelet Peaks with Blanking (WPB) detector [4], our own modification of WPM that sacrifices accuracy for efficiency, and
3) A Medtronic Peak (MDT) detector [4], which operates in the time domain, and is implemented in an Implantable Cardioverter Defibrillator (ICD) on the market today.

In this paper, we extend our preliminary work presented in [4] with the following results:

- We update the QRE programs of peak detectors using the StreamQRE framework recently introduced in [6] and implemented as a Java library in [2]
- We show how to program, within the same language, the discriminators used in single-chamber ICDs of St. Jude Medical as described in [29], thus demonstrating that QRE is suitable for both peak detection and discrimination
- We demonstrate the QREs programs for discrimination on real patient data.

We have also expanded the paper to contain a formal introduction to the QRE language.

Paper organization. Section 2 provides the necessary mathematical background to understand the peak characterization in the wavelet domain. Section 3 introduces the QRE language and basic constructs. The QRE encoding of the WPM peak detector is provided in Section 4. The operation of three detectors is illustrated by running them on real patient electrograms in Section 5. In Section 6 we present the St. Jude Medical arrhythmia discrimination algorithm, with its QRE implementation discussed in Section 7. We illustrate its operation in Section 8. In Section 9 we review the related literature, while in Section 10 we draw our conclusions and discuss future work.

2 Peaks in the Wavelet Domain

Rather than confine ourselves to one particular peak detector, we first summarize a general definition of *singularities* due to Mallat and Huang [25], which naturally gives rise to a peak detection algorithm. This definition operates in the wavelet domain, so a brief overview of the wavelet transform is first provided.

2.1 The Wavelet Transform

Let $\{\Psi_s\}_{s>0}$ be a family of functions, called *wavelets*, which are obtained by scaling and dilating a so-called *mother wavelet* $\psi(t)$: $\Psi_s(t) = \frac{1}{\sqrt{s}}\psi\left(\frac{t}{s}\right)$. The *wavelet transform* W_x of signal $x : \mathbb{R}_+ \to \mathbb{R}$ is the two-parameter function:

$$W_x(s,\, t) = \int_{-\infty}^{+\infty} x(\tau)\Psi_s(\tau - t)\, d\tau \qquad (1)$$

A common choice of ψ for peak detection is the n^{th} derivative of a Gaussian, that is: $\psi(t) = \frac{d^n}{dt^n}G_{\mu,\sigma}(t)$. Eq. (1) is known as a *Continuous Wavelet Transform* (CWT), and $W_x(s,t)$ is known as the *wavelet coefficient*.

Parameter s in the wavelet ψ_s is known as the *scale* of the analysis. A smaller value of s (in particular $s < 1$) *compresses* the mother wavelet, so that only values close

to $x(t)$ influence the value of $W_x(s,t)$ (see Eq. (1)). Thus, at smaller scales, the wavelet coefficient $W_x(s,t)$ captures *local* variations of x around t, and these can be thought of as being the higher-frequency variations, i.e., variations that occur over a small amount of time. At larger scales (in particular $s > 1$), the mother wavelet is *dilated*, so that $W_x(s,t)$ is affected by values of x far from t as well. Thus, at larger scales, the wavelet coefficient captures variations of x over large periods of time.

Fig. 1 shows a Normal Sinus Rhythm EGM and its CWT $|W_x(s,t)|$. The latter plot is known as a *spectrogram*. Brighter colors indicate larger values of coefficient magnitudes $|W_x(s,t)|$. It is possible to see that early in the signal, mid- to low-frequency content is present (bright colors mid- to top of spectrogram), followed by higher-frequency variation (brighter colors at smaller scales), and near the end of the signal, two frequencies are present: mid-range frequencies (the bright colors near the middle of the spectrogram), and very fast, low amplitude oscillations (the light blue near the bottom-right of the spectrogram).

2.2 Wavelet Characterization of Peaks

Consider the CWT $|W_x(s,t)|$ shown in Fig. 1. The coefficient magnitude $|W_x(s,t)|$ is a measure of signal power at (s,t). At larger scales, one obtains an analysis of the low-frequency variations of the signal, which are unlikely to be peaks, as the latter are characterized by a rapid change in signal value. At smaller scales, one obtains an analysis of high-frequency components of the signal, which will include both peaks and noise. These remarks can be put on solid mathematical footing [26, Ch. 6]. **Therefore, for peak detection one must start by querying CWT coefficients that occur at an appropriately chosen scale $\bar{s}$.**

At a fixed scale $\bar{s}$, $|W_x(\bar{s},\cdot)|$ is a function of time. The next task is to find the *local maxima* of $|W_x(\bar{s},t)|$ as t varies, because these are precisely the times when energy variations at scale $\bar{s}$ are locally concentrated. **Thus peak characterization further requires querying the local maxima at $\bar{s}$.**

Not all maxima are equally relevant: only those with value above a threshold, since these are indicative of signal variations with large energy concentrated at $\bar{s}$. **Therefore, we should only consider local maxima with a value above some threshold $\bar{p}$.**

Maxima in the wavelet spectrogram are not isolated: as shown in [26, Thm. 6.6], when the wavelet ψ is the n^{th} derivative of a Gaussian, the maxima belong to connected curves $s \mapsto \gamma(s)$ that are never interrupted as the scale decreases to 0. These *maxima lines* can be clearly seen in Fig. 1: they are the vertical lines of brighter color extending all the way to the bottom. Multiple maxima lines may converge to the same point $(0,t_c)$ in the spectrogram as $s \to 0$. As shown in [25], *singularities* in the signal always occur at the convergence times t_c. For our purposes, a singularity is a time when the signal undergoes an abrupt change (specifically, the signal is poorly approximated by an $(n+1)^{th}$-degree polynomial at that change-point). **These convergence times are then the peak times that we seek.**

Although theoretically, the maxima lines are connected, in practice, signal discretization and numerical errors will cause some interruptions. Therefore, rather than look for truly connected maxima lines, we only look for (ϵ,δ)-connected lines: given $\epsilon,\delta > 0$, an (ϵ,δ)-*connected curve* $\gamma(s)$ is one such that for any s, s' in its domain, $|s - s'| < \epsilon \implies |\gamma(s) - \gamma(s')| < \delta$.

A succinct description of this *Wavelet Peaks with Maxima* (WPM) is then:

– (Characterization *WPM*) Given positive reals $\bar{s}, \bar{p}, \epsilon, \delta > 0$, a peak is said to occur at time t_0 if there exists a (ϵ,δ)-connected curve $s \mapsto \gamma(s)$ in the (s,t)-plane such that $\gamma(0) = t_0$, $|W_x(s,\gamma(s))|$ is a local maximum along the t-axis for every s in $[0,\bar{s}]$, and $|W_x(\bar{s},\gamma(\bar{s}))| \geq \bar{p}$.

The choice of values $\bar{s}$, ϵ, δ and $\bar{p}$ depends on prior knowledge of the class of signals we are interested in. Such choices are pervasive in signal processing, as they reflect application domain knowledge.

3 A QRE Primer

The specification of discrimination and PD (Section 2) requires a language that: 1) has a rich set of numerical operations, 2) supports matching of complex patterns in the signal, and 3) enables the synthesis of time- and memory-efficient implementations. This led to the consideration of the StreamQRE language and execution engine [6], [27].

The basic semantic objects of the QRE language are called *streaming functions*, and they describe the transformation of an input stream to an output stream. More specifically, a streaming function is a *partial function* $f : D^* \rightharpoonup C$ from finite sequences of input data items (of type D) to output values (of type C). A crucial notion is the *rate* of a streaming function that captures its domain. In other words, as the function reads the input data stream, a prefix of the input triggers the production of an output value exactly when the prefix matches the rate. In the StreamQRE language, the rates are required to be *regular*, and therefore can be captured by symbolic regular expressions. This leads to efficient decision procedures for constructing well-typed expressions.

3.1 Quantitative Regular Expressions

We will introduce now the language of *Quantitative Regular Expressions (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 `Bool` 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 [1]. That is, we write $(A\ x) \text{->} t(x)$ 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 $(A\ x,\ B\ y,\ C\ z) \text{->} t(x,y,z)$ to mean $\lambda x{:}A, y{:}B, z{:}C.t(x,y,z)$. For example, the identity function on D is $(D\ x) \text{->} x$, the left projection on $A \times B$ is $(A\ x,\ B\ y) \text{->} x$, the right projection on $A \times B$ is $(A\ x,\ B\ y) \text{->} y$, and $(D\ x) \text{->} \texttt{def}$ 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 $(x,y) \text{->} x$.

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 $\varphi : D \to \texttt{Bool}$ to indicate that φ is a predicate on D, and we denote by $\texttt{true}_D : D \to \texttt{Bool}$ the predicate that is always true. The predicate $((\mathbb{Z}\ x) \text{->} x > 0) : \mathbb{Z} \to \texttt{Bool}$ is true of the strictly positive integers.

Example 3.1. 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 $\neg\texttt{isV}$ and $\texttt{isV}$ test if a Boolean value is zero or one respectively. □

For a type D, we define the set of ***symbolic regular expressions over*** D [34], denoted $\texttt{RE}\langle D\rangle$, with the grammar:

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

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 $r : \texttt{RE}\langle D\rangle$ to indicate the r is a regular expression over D. Every expression $r : \texttt{RE}\langle D\rangle$ 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 3.2. The symbolic regular expression $(\neg\texttt{isV})^* \cdot \texttt{isV}$ denotes sequences of samples that contain a single ventricular beat (contraction) at the end. □

The notion of ***unambiguity*** for regular expressions [13] 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*.

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 [5].

The rate $\mathsf{R}(\texttt{f})$ of a query $\texttt{f}$ is a symbolic regular expression that denotes the domain of the transformation that $\texttt{f}$ 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 $\texttt{f}$ with a type $\texttt{QRE}\langle D, C\rangle$ to denote that the input stream has elements of type D and the outputs are values of type C.

Atomic queries. The basic building blocks of queries are expressions that describe the processing of a single data item. Suppose $\varphi : D \to \texttt{Bool}$ 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* $\texttt{atom}(\varphi, op) : \texttt{QRE}\langle D, C\rangle$, 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 $\texttt{atom}(\varphi, \texttt{id}_D)$ by $\texttt{atom}(\varphi)$, and the query $\texttt{atom}(\texttt{true}_D)$ by $\texttt{atom}()$.

Example 3.3. For the Boolean ventricular heart signal, the query that matches a single item that is a heartbeat and returns nothing is $\texttt{f} = \texttt{atom}(\texttt{isV}, x \text{->} \texttt{def})$. The type of $\texttt{f}$ is $\texttt{QRE}\langle \mathbb{B}, \mathbb{U}\rangle$ and its rate is $\texttt{isV}$. □

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

Iteration. Suppose that we want to iterate a computation $\texttt{f} : \texttt{QRE}\langle D, A\rangle$ 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

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

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 $\texttt{f}$. The output values $a_1\ a_2\ \cdots\ a_n$ with $a_i = \texttt{f}(w_i)$ 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:

$$\text{fold}(b, op, \varepsilon) = b \quad \text{fold}(b, op, \gamma a) = op(\text{fold}(b, op, \gamma), a)$$

for all sequences $\gamma \in A^*$ and all values $a \in A$. For example, $\text{fold}(b, op, a_1a_2) = op(op(b, a_1), a_2)$.

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 $w = w_1w_2 \dots w_n$ with each w_i matching f. This requirement can be expressed equivalently as: the rate $\mathsf{R}\texttt{f}$ is unambiguously iterable.

Combination and application. Assume the queries f and g describe stream transformations with outputs of type A and B respectively, and process the same set of input sequences of elements of type D, 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.

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. That is, we split the input into two parts $w = w_1w_2$, 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 $w = w_1w_2$ 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 $\texttt{split}(\texttt{f},\ \texttt{g},\ \texttt{h}, (x, y, z) \texttt{->}\ op(x, y, z))$.

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 R for queries $\texttt{f} \gg \texttt{g}$.

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

3.2 Derived constructs

The core language of Section 3.1 is expressive enough to describe many common stream transformations. We present below several derived constructs.

Matching without output. Suppose r is an unambiguous symbolic regular expression over the data item type D. The query $\texttt{match}(r)$, 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:

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

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

"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 $\texttt{iterUntil}(\phi, \psi, c, op)$ aggregates a sequence of data items that satisfy ϕ and stops when an item that satisfies ψ is found. It is encoded as:

$$\texttt{iterUntil}(\phi, \psi, c, op) \triangleq \texttt{split}(\texttt{iter}(\texttt{atom}(\phi), c, op), \texttt{atom}(\psi), (x, y) \texttt{->}\ x)$$

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

Stream Annotation. Suppose that the input stream has items of type D, f is a query of type $\texttt{QRE}\langle D, C\rangle$, 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 $\texttt{annt}(\texttt{f}, op_1, op_2) : \texttt{QRE}\langle D, E\rangle$ with rate D^+. This annotation query can be encoded using the regular constructs of Section 3.1, but the encoding is complex and inefficient, so we provide a custom efficient implemenation.

Tumbling windows. The term tumbling windows is used to describe the splitting of the stream into contiguous non-overlapping subsequences [23]. 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:

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

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

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

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

whose rate is equal to $\mathsf{R}(\texttt{f})^+$. This query can be encoded using the regular constructs of Section 3.1 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).

3.3 A Java Library of QREs

StreamQRE has been implemented as a Java library [2] in order to facilitate its integration with user-defined types and operations. The implementation covers all the core constructs of Section 3.1, and also provides optimizations for the derived constructs of Section 3.2 (matching without output, "until" iteration, stream annotation, etc.).

QREs can be compiled into efficient *evaluators* that process each data item in time (or memory) polynomial in the size of the QRE and proportional to the maximum time (or memory) needed to perform an *operation* on a set of cost terms, such as addition, least-squares, etc. The operations are selected from a set of operations *defined by the user*. *It is important to be aware that the choice of operations constitutes a trade-off between expressiveness (what can be computed) and complexity (more complicated operations cost more).* See [5] for restrictions placed on the predicates and the symbolic regular expressions.

The declarative nature of QREs will be important when writing complex algorithms, without having to explicitly maintain state and low-level data flows. But as with any new language, QREs require some care in their usage.

4 QRE Implementation of a Peak Detector

We now describe the QRE program that implements the Wavelet Peak Maxima (WPM) peak detector of Section 2.2. The emphasis is on the fact that this algorithm can be described in a declarative fashion using QREs, without resorting to explicitly storing state, etc.

A numerical implementation of a Continuous Wavelet Transform (CWT) returns a discrete set of coefficients. Let $s_1 < s_2 < \ldots < s_n$ be the analysis scales and let $t_1, t_2, \ldots$ be the signal sampling times. Recall that a QRE views its input as a stream of incoming data items. A data item for WPM is $d = (s_i, t_j, |W_x(s_i, t_j)|) \in D := (\mathbb{R}_+)^3$. We use $d.s$ to refer to the first component of d, and $d.|W_x(s,t)|$ to refers to its last component. The input stream $w \in D^*$ is defined by the values from the spectrogram organized in a column-by-column fashion starting from the highest scale:

$$w = \underbrace{(s_n, t_1, |W_x(s_n, t_1)|), \ldots, (s_1, t_1, |W_x(s_1, t_1)|)}_{w_{t_1}} \ldots$$
$$\ldots \underbrace{(s_n, t_m, |W_x(s_n, t_m)|), \ldots, (s_1, t_m, |W_x(s_1, t_m)|)}_{w_{t_m}}$$

Let s_σ, $1 \le \sigma \le n$, the the scale that equals $\bar{s}$. Since the scales $s_i > s_\sigma$ are not relevant for peak detection (their frequency is too low), they should be discarded from w. Now, for each scale s_i, $i \le \sigma$, we would like to find those local maxima of $|W_x(s_i, \cdot)|$ that are larger than threshold p_i[1]. We build the QRE `peakWPM` bottom-up as follows. In what follows, $i = 1, \ldots, \sigma$. See Fig. 2.

- QRE $\texttt{selectCoef}_i$ selects the wavelet coefficient magnitude at scale s_i from the incoming spectrogram column w_t. It must first wait for the entire colum to arrive in a streaming fashion, so it matches n data items (recall there are n items in a column – see Fig. 2) and returns as cost $d.|W_x(s_i,t)|$.

$$\texttt{selectCoef}_i := (d_n d_{n-1} \ldots d_1 ?\, d_i.|W_x(s_i,t)|)$$
$$\texttt{selectCoef}_i := \texttt{split}(\texttt{match}(\texttt{d}^{n-i}),\ \texttt{d},\ \texttt{match}(\texttt{d}^{i-1}),\ (x,y,z) \texttt{->}\ y.|W_x(s_i,t)|)$$

- QRE $\texttt{repeatSelectCoef}_i$ applies $\texttt{selectCoef}_i$ to the latest column w_t. To do so, it executes $\texttt{selectCoef}_i$ on the last column, and ignores all columns that preceded it. It returns the selected coefficient $|W_x(s_i,t)|$ from the last column.

$$\texttt{repeatSelectCoef}_i := \texttt{iter}_1(\texttt{selectCoef}_i, 0, (x,y)\texttt{->}y)$$

- QRE $\texttt{localMax}_i$ matches a string of real numbers of length at least 3: $r_1 ... r_{k-2} r_{k-1} r_k$. It returns 1 if the value of r_{k-1} is larger than r_k and r_{k-2}, and is above some pre-defined threshold p_i; otherwise, it returns 0. This will be used to detect local maxima in the spectrogram in a moving-window fashion. In detail:

$$\texttt{localMax}_i := \texttt{wnd}(\texttt{atom}(), 3, 0, \texttt{ins}, \texttt{rmv}, \texttt{LM}_3)$$

  The query $\texttt{localMax}_i$ executes $\texttt{atom}()$ over a sliding window of size 3, and performs the operation $\texttt{LM}_3$ over these three values. Operation $\texttt{LM}_3$ simply returns 1 if the middle value is a local maximum that is above p_i, and returns zero, otherwise.
- The query $\texttt{oneMax}_i$ feeds outputs of QRE $\texttt{repeatSelectCoef}_i$ to the QRE $\texttt{localMax}_i$.

$$\texttt{oneMax}_i := \texttt{repeatSelectCoef}_i \gg \texttt{localMax}_i$$

  Thus, $\texttt{oneMax}_i$ "sees" a string of coefficient magnitudes $|W_x(s_i,t_1)|, |W_x(s_i,t_2)|, \ldots$ generated by (streaming) $\texttt{repeatSelectCoef}_i$, and produces a 1 at the times of local maxima in this string.
- QRE $\texttt{peakTimes}_i$ collects the times of local maxima at scale s_i into one set.

$$\texttt{peakTimes}_i := \texttt{oneMax}_i \gg \texttt{unionTimes}$$

  It does so by passing the string of 1s and 0s produced by $\texttt{oneMax}_i$ to `unionTimes`. The latter counts the number of 0s separating the 1s and puts that in a set $\mathcal{M}_i$. Therefore, after k columns w_t have been seen, set $\mathcal{M}_i$ contains all local maxima at scale s_i which are above p_i in those k columns.
- QRE `peakWPM` is the final QRE. It combines results obtained from scales s_σ down to s_1:

$$\texttt{peakWPM} := \texttt{combine}(\texttt{peakTimes}_\sigma, ..., \texttt{peakTimes}_1, conn_\delta)$$

1. $p_\sigma = \bar{p}$, $p_{i<\sigma} = 0$, since we threshold only the spectrogram values at scale $\bar{s}$. After this initial thresholding, tracing of maxima lines returns the peaks.

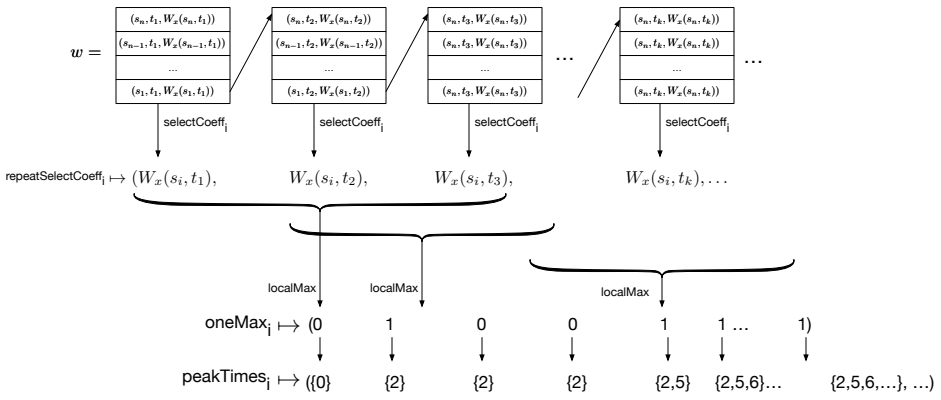

Fig. 2: QRE `peakWPM`

Operator $conn_\delta$[2] checks if the local maxima times for each scale (produced by $\texttt{peakTimes}_i$) are within a δ of the maxima at the previous scale.

In summary, the complete QRE `peakWPM` is given top-down by:

$$\begin{aligned}
\texttt{peakWPM} &:= \texttt{combine}(\texttt{peakTimes}_\sigma, ..., \\
&\qquad \texttt{peakTimes}_1, conn_\delta) \\
\texttt{peakTimes}_i &:= \texttt{oneMax}_i \gg \texttt{unionTimes} \\
\texttt{oneMax}_i &:= \texttt{repeatSelectCoef}_i \gg \texttt{localMax}_i \\
\texttt{localMax}_i &:= \texttt{wnd}(\texttt{atom}(), 3, 0, \texttt{ins}, \texttt{rmv}, \texttt{LM}_3) \\
\texttt{repeatSelectCoef}_i &:= \texttt{iter}_1(\texttt{selectCoef}_i, 0, (x, y) \text{->} y) \\
\texttt{selectCoef}_i &:= \texttt{split}(\texttt{match}(\texttt{d}^{n-i}),\ \texttt{d},\ \texttt{match}(\texttt{d}^{i-1}), \\
&\qquad (x, y, z) \text{->} y.|W_x(s_i, t)|)
\end{aligned}$$

5 Experimental Results of Peak Detector

We show the results of running peak detector `peakWPM`, described in Section 4, on a dataset of real patient electrograms. For comparison purposes, we also programmed two peak detectors in QRE: `peakMDT`, which is available in a commercial ICD [30], and `peakWPB` (*Wavelet Peaks with Blanking*), which is a simplified variant of `peakWPM`. For details , see [4].

The implementation uses an early version of the StreamQRE Java library [27]. Comparing the runtime and memory consumption of different algorithms (including algorithms programmed in QRE) in a consistent and reliable manner requires running a compiled version of the program on a particular hardware platform. No such compiler is available at the moment, so we don't report such performance numbers.

The results in this section should not be interpreted as establishing the superiority of one peak detector over another, as this is not this paper's objective. Rather, the objective is to motivate the use of a formal language for programming peak detectors, and other tasks in ICDs. Namely, by highlighting the challenges involved in peak detection for cardiac signals, this section establishes the need for a formal understanding of their operation on classes of arrhythmias, and thus the need for a formal description of peak detectors.

Fig. 3 presents one rectified EGM signal of a Ventricular Tachycardia (VT). Circles (indicating detected time of peak) show the result of running `peakWPM` (red circles) and `peakWPB` (black circles). These results were obtained for $\bar{s} = 80$, $BL = 150$, and different values of $\bar{p}$. The first setting of $\bar{p}$ (Fig. 3 (a)) for both detectors was chosen to yield the best performance. This is akin to the way cardiologists set the parameters of commercial ICDs: they observe the signal, then set the parameters. We refer to this as the *nominal setting*. Ground-truth is obtained by having a cardiologist examine the signal and annotate the true peaks.

We first observe that the peaks detected by `peakWPM` match the ground-truth; i.e., the nominal performance of `peakWPM` yields perfect detection. This is not the case with `peakWPB`. Next, one can notice that the time precision of detected peaks with `peakWPM` is higher than with `peakWPB`. Note also that the results of `peakWPM` are stable for various parameters settings. Improper thresholds $\bar{p}$ or scales $\bar{s}$ degrade the results only slightly (compare locations of red circles on Fig. 3 (a) with Fig. 3 (b)). By contrast, `peakWPB` detects additional false peaks (compare black circles in Figs. 3 (a) and (b)).

Fig. 4 (left) shows WPM (red circles) running on a Ventricular Fibrillation (VF) rhythm, which is a potentially fatal disorganized rhythm. Again, we note that WPM finds the peaks.

Detector `peakMDT` works almost perfectly with nominal parameters settings on any Normal Sinus Rhythm (NSR) signal (see Fig. 4 right). NSR is the "normal" heart rhythm. Using the same nominal parameters on more disorganized EGM signals with higher variability in amplitude, such as VF, does not produce good results; see the black circles in Fig. 4 left.

6 Arrhythmia Detection in QRE

The outcome of peak detection is a discrete-time, uniformly-sampled, time-stamped Boolean signal, where a 1 indicates

2. Operator $conn_\delta$ can be defined recursively as follows: $conn_\delta(X, Y) = \{y \in Y : \exists x \in X : |x - y| \leq \delta\}$, $conn_\delta(X_k, .., X_1) = conn_\delta(conn_\delta(X_k, .., X_2), X_1)$

(a) $\bar{p} = 400$ (b) $\bar{p} = 150$

Fig. 3: `peakWPM`-detected peaks (red circles) and `peakWPB`-detected peaks (black circles) on a VT rhythm.

Fig. 4: WPM and `peakMDT` running on a VF rhythm (left) and `peakMDT` running on an NSR rhythm (right).

an electrical event (a depolarization), and a 0 indicates the absence thereof. This signal is input to the next stage of the ICD, which is the *detection algorithm*. This is an algorithm that tries to detect whether the current rhythm is a ventricular tachycardia (which can be fatal), or not. In this section we describe a detection algorithm used in the single-chamber ICDs of St. Jude Medical as described in [29][3] It is representative of detection algorithms of other manufacturers which all have similar components and functions.

All ICD detection algorithm take the form of a decision tree, where the nodes are independent *discriminators*. Each discriminator computes one particular feature of the input boolean signal, based on which it decides how to branch in the tree. The decision tree of the St. Jude Medical algorithm is presented in a Figure 5; we will refer to it henceforth as SJM. SJM is made of the following discriminators [35].

6.1 Initial Rhythm Classification

SJM first compares the current interval (time in milliseconds between two consecutive 1s) and the running average of the most recent four intervals to a predefined threshold of 350ms. See Figure 6.

If both these quantities are less than the threshold, the current interval is marked as 'Tach'. If both are slower than the threshold, the interval is marked as 'Sinus'. Otherwise, it is marked as 'Undefined'.

6.2 Sudden Onset

Sudden onset leverages the clinical observation that VT occurrence is usually sudden, in contrast to the gradual onset of SVT (during which the rhythm usually accelerates gradually). The Onset discriminator quantifies the suddenness of tachycardia onset as follows. First, it computes the 9 most recent running average $A_1, \ldots, A_9$, where each running average is the average of 4 intervals. A_9 is the most recent

3. Single-chamber ICDs only measure the signal in the right ventricle.

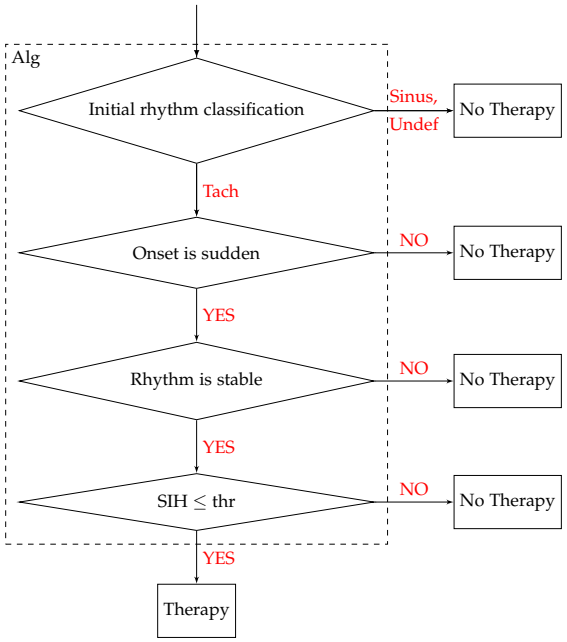

Fig. 5: St. Jude Medical discrimination algorithm

(current) average. Then it evaluates the following absolute differences: $d_1 = |A_9 - A_1|, ..., d_4 = |A_9 - A_7|$. See Figure 7. If any of these differences is greater than a physician-set threshold, the onset is considered to be sudden. Otherwise, the onset is gradual.

6.3 Rhythm Stability

Intervals during VT usually display low variability, a property called 'stability' in the medical literature. The rhythm stability discriminator quantifies rhythm stability by computing the difference between the second longest and the second shortest intervals in the last 10 intervals. If the difference is larger than a pre-programmed threshold, this

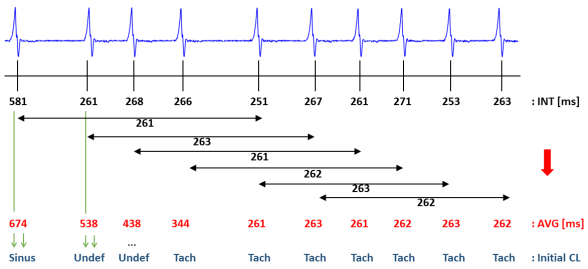

Fig. 6: Evaluation of the running average and the initial rhythm classification discriminator. Running average is evaluated over the current interval and three preceding intervals. The threshold for the initial classification discriminator is 350ms.

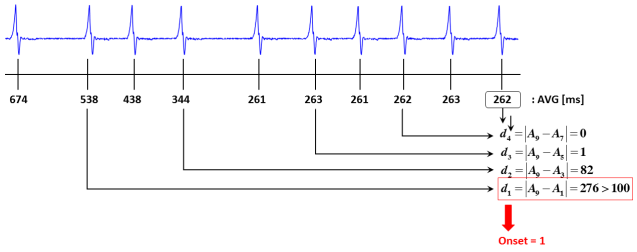

Fig. 7: Evaluation of the sudden onset discriminator. Threshold default value is 100ms.

indicates an unstable rate. Otherwise, the rhythm is considered stable.

6.4 Sinus Interval History (SIH)

Sinus interval history prevents an inappropriate therapy in case of stable and sudden SVT with dropped beats. SIH records the number of 'Sinus'-labeled intervals within a detection window of 10 intervals. If this counter is greater than or equal to a threshold, no therapy is delivered.

7 QRE Implementation of Detection Algorithm

The QRE implementation of SJM (Section 6) is divided into four main stages. The first three stages annotate the input stream with interval lengths, running interval averages and rhythm labels (See section 6.1). Stage 4 computes all discriminators needed for a final decision: Therapy or No Therapy.

The input to the detection algorithm is a discrete-time boolean signal. Formally, let $\mathbb{B} = \{0, 1\}$ be the set of boolean values. At every time $t \in \mathbb{N}$, the detection algorithm receives a data item s of the following form $s = (V, t) \in D := \mathbb{B} \times \mathbb{N}$, where $V = 1$ indicates there is a beat at time t, and $V = 0$ indicates that there is not.

We will explain now each stage in detail, and present the precise implementation in the StreamQRE language.

Stage 1: Annotate with interval length. An *interval* is the amount of time that elapses between two consecutive beats. Thus, it is the number of 0s between consecutive 1s in the input stream. The corresponding QRE is given by:

$$\begin{aligned}
\texttt{lincr} &:= (x, y) \texttt{->} x + 1, \text{ of type } \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
\texttt{intV} &:= \texttt{iterUntil}(\neg\texttt{isV}, \texttt{isV}, 0, \texttt{lincr}) \\
\texttt{allIntV} &:= \texttt{iterLast}(\texttt{intV}) \texttt{ //rate } ((\neg\texttt{isV})^* \cdot \texttt{isV})^+ \\
\texttt{stage1} &:= \texttt{annt}(\texttt{allIntV}, x \texttt{->} x, (x, c) \texttt{->} x.I := c)
\end{aligned}$$

The query `intV` iterates over 0s (predicate $\neg$`isV`) until it finds 1 (predicate `isV`). Each matched 0 increments the counter (`lincr`), starting from initial value 0. QRE `allIntV` process all consecutive intervals in the input stream by iterating `intV`. The query `stage1` annotates the input items with the interval length values I calculated by the query `allIntV`. The output stream s_1 from this stage consists of data items of the following form $(V, t, I) \in D_1 := \mathbb{B} \times \mathbb{N} \times \mathbb{N}$.

Stage 2: Annotate with average interval length. This stage annotates its input stream with running average interval values over a window of 4 intervals (see Figure 7):

$$\begin{aligned}
\texttt{blockV} &:= \texttt{split}(\texttt{match}((\neg\texttt{isV})^*), \texttt{isV}, (x, y) \texttt{->} y) \\
\texttt{wndAvg} &:= \texttt{wnd}(\texttt{blockV}, 4, 0, \texttt{avg}) \\
\texttt{stage2} &:= \texttt{annt}(\texttt{wndAvg}, x \texttt{->} x, (x, c) \texttt{->} x.avg := c)
\end{aligned}$$

The query `blockV` matches 0^*1 and returns the last data item that matches $V = 1$. Thq QRE `wndAvg` executes `blockV` in a sliding window of length 4 and computes the average value. The query `stage2` annotates the stream with all these sliding-window averages. The output stream s_2 from this stage consists of data items of the following form: $(V, t, I, avg) \in D_2 := \mathbb{B} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Q}$.

Stage 3: Annotate with label. This stage annotates the stream s_2 with the rhythm classification label from $\mathbb{S} = \{Tach, Sinus, Undef\}$. For details, see Section 6.1. The query for computing the label is given by:

$$\begin{aligned}
\texttt{fLabel} := x \texttt{->} & \\
&\texttt{if}(x.avg \leq thr \wedge x.I \leq thr), \texttt{return } Tach \\
&\texttt{elseif}(x.avg > thr \wedge x.I > thr), \texttt{return } Sinus \\
&\texttt{else, return } Undef \\
\texttt{label} &:= \texttt{apply}(\texttt{iterLast}(\texttt{blockV}), \texttt{fLabel}) \\
\texttt{stage3} &:= \texttt{annt}(\texttt{label}, x \texttt{->} x, (x, c) \texttt{->} x.L := c)
\end{aligned}$$

`fLabel` is a basic operation to compute the label for each data item. The query `label` applies `fLabel` to every output of `blockV` (defined earlier). The query `stage3` annotates the input stream for this stage with obtained labels. The output stream s_3 from this stage consists of data items of the following form: $(V, t, I, avg, L) \in D_3 := \mathbb{B} \times \mathbb{N}^2 \times \mathbb{Q} \times \mathbb{S}$.

Stage 4: Discriminators. This stage computes the discriminators Initial Rhythm Classification, Sudden Onset, Rhythm Stability and SIH, and combines them together according to the discrimination algorithm from Fig. 5:

$$\begin{aligned}
\texttt{initClasf} &:= \texttt{iter}_1(\texttt{blockV}, 0, (x, y) \texttt{->} y.L == Tach) \\
\texttt{onset} &:= \texttt{wnd}(\texttt{blockV}, 9, 0, \texttt{ins}, \texttt{rmv}, \texttt{fOnset}) \\
\texttt{stability} &:= \texttt{wnd}(\texttt{blockV}, 10, 0, \texttt{ins}, \texttt{rmv}, \texttt{fStability}) \\
\texttt{sih} &:= \texttt{wnd}(\texttt{blockV}, 10, 0, \texttt{ins}, \texttt{rmv}, \texttt{fSih}) \\
\texttt{stage4} &:= \texttt{combine}(\texttt{initClasf}, \texttt{onset}, \texttt{stability}, \\
&\qquad \texttt{sih}, (x, y, z, u) \texttt{->} x \wedge y \wedge z \wedge u)
\end{aligned}$$

where the construct `wnd` describes a sliding-window computation that maintains a buffer with items defined by QRE `blockV`. The function `ins` adds a new item to the buffer, and the function `rmv` removes an expiring item from the buffer. Functions `fOnset`, `fStability`, `fSih` compute the corresponding discriminators described in Section 6 using only the items contained in the buffer. The QRE `stage4` combines the defined above queries by logical operator $\wedge$.

Overall St Jude Discrimination Algorithm. The top-level query for St. Jude Medical discrimination algorithm is the streaming composition of all stages:

$$\texttt{discrStJude} := \texttt{stage1} \gg \texttt{stage2} \gg \texttt{stage3} \gg \texttt{stage4}$$

8 Experimental results of discrimination algorithm

We validated that our QRE implementation of SJM indeed follows the description in [29] in two ways[4]: by comparing its output to that of a second, Matlab, implementation on the same set of signals, and by checking that its accuracy (defined below) is within the expected range.

First, we implemented SJM in Matlab and compared the output of the Matlab implementation with that of `discrStJude` by running them both on a database of 1920 EGM signals, described in greater details below. The two implementations returned exactly the same decisions on all signals. To illustrate the details of the query execution, consider the example in Fig. 8. It shows an EGM signal from the database with the corresponding boolean stream generated by the peak detector. The results of running `stage1`, `stage2` and `stage3` on this signal are shown as INT[ms], AVG[ms] and LABEL labels streams, respectively. All discriminators of `stage4` are depicted as a red box. At time $12,390$ms all four discriminators returned $true$, and the discrimination algorithm therefore outputs Therapy.

The database of signals we used in this evaluation consists of 1920 EGMs, equally divided between 960 VTs and 960 SVTs. The input stream was generated by the heart model of [3], [22], whose outputs had been validated for realism by cardiologists [22]. Using a heart model allows us to generate different types of VTs and SVTs, thus exposing the QRE implementation to a wider range of rhythms than is possible by using a database of natural signals. It also enables us to generate more signals for free, whereas collecting EGM signals from patients is very laborious. Finally, and specifically for the purpose of measuring the accuracy of our implementation of SJM, the model-generated signals we use come with the true timing of events (i.e., the boolean stream of events is also given by the model), which allows an accurate measure of the arrhythmia detector's accuracy.

The second way we validated our implementation of SJM is by running `discrStJude` on the database of signals and measuring its Specificity and Sensitivity, defined respectively as

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

4. Note we are *not* validating that the *algorithm itself* is 'correct' in some sense. That is a separate question for future research. Our concern here is to validate that we *implemented* the algorithm described in [29] correctly.

TABLE 1: Database-averaged detection accuracy.

Sensitivity	Specificity
92% (884/960)	96% (921/960)

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

Table 1 shows the results. The numbers fall within the expected range, namely above 90% for both.

9 Related Work

Signal Temporal Logic (STL) [24] is a popular specification language employed to express real-time properties over real-valued signals. The use of STL has been proposed in several application domains [10] including the monitoring of medical signals [9], [16]. In [15], STL was extended with a signal value *freeze operator* enabling the specification in the time-domain of complex oscillatory patterns (i.e., damping oscillations). However, this extension does not allow the detection of oscillations within a specified frequency range.

If we consider the spectrogram of a signal as a 2D map (from time and scale to amplitude), one may think to apply a spatial-temporal logic such as SpaTeL [21], Signal Spatio-Temporal Logic (SSTL) [28] or STREL [8] on spectrograms. However, their underlying spatial models, graph structures for SSTL and STREL and quadtrees for SpaTeL, are not suitable for this purpose.

There has been also an effort to propose logics for describing both frequency and temporal properties, including Time-Frequency Logic (TFL) in [19] and the approach described in [17]. However, TFL is not expressive enough for peak detection, because it lacks the necessary mechanisms to quantify over variables or to freeze their values.

Timed regular expressions [7], [32], [33] extend regular expressions by clocks and are expressively equivalent to timed automata, but they are insufficient to specify the tasks described in this paper. The work proposed in [20] on measuring signals with timed patterns does not improve the situation for our case study, since it does not provide, neither in the specification nor in the measurement, the notion of local minima/maxima that is fundamental for peak detection. Furthermore, the operator of measure is separated by the specification of the pattern to match.

Stream runtime verification languages (SRVs) [14], [18], such as LOLA [18], require explicit encoding of all the relations between input and output streams. This would result in a very cumbersome activity while encoding the complex tasks of this paper. Moreover, unlike Boolean SRVs [14], QREs allow multiple unrestricted data types in the intermediary computations.

10 Conclusions and Future Work

The tasks of discrimination and peak detection, fundamental to arrhythmia-discrimination algorithms, are easily and succinctly expressible in QREs. One obvious limitation of QREs is that they only allow regular matching, though this is somewhat mitigated by the ability to chain QREs (though the streaming combinator $\gg$) to achieve more complex

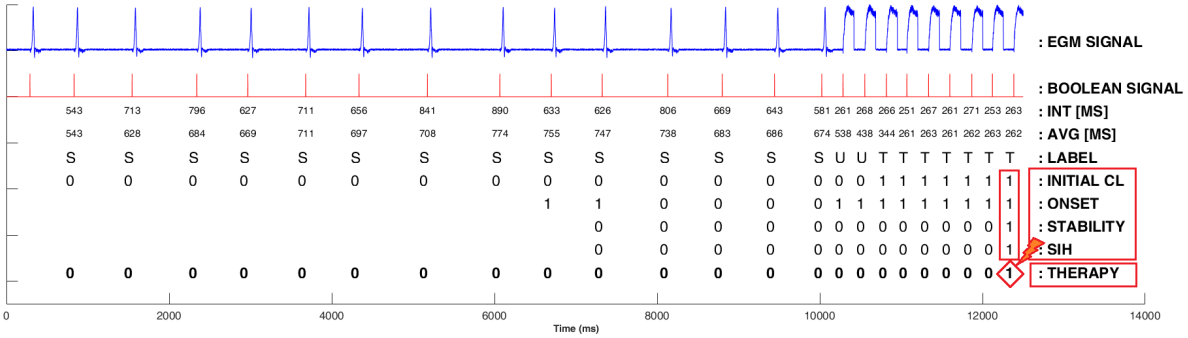

Fig. 8: Boolean signal obtained from the EGM signal and the streaming output of QRE implementation of St Jude Discrimination algorithm presented in Sections 6-7.

tasks. One advantage of programming in QREs is that it automatically provides us with a base implementation, whose time and memory complexity is independent of the stream length.

As future work, it will be interesting to compile a QRE into C or assembly code to measure and compare actual performance on a given hardware platform. Also, just like an RE has an equivalent machine model (DFA), a QRE has an equivalent machine model in terms of a deterministic finite-state transducer [5]. This points to an analysis of a QRE's correctness and efficiency beyond testing.

Acknowledgments

This work is supported in part by AFOSR Grant FA9550-14-1-0261 and NSF Grants IIS-1447549, CNS-1446832, CNS-1446664, CNS-1445770, and CNS-1445770. E.B. and R.G. acknowledge the partial support of the Austrian National Research Network S 11405-N23 and S 11412-N23 (RiSE/SHiNE) of the Austrian Science Fund (FWF) and the ICT COST Action IC1402 Runtime Verification beyond Monitoring (ARVI).

References

[1] Java's lambda expressions. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

[2] StreamQRE library. http://www.seas.upenn.edu/~mamouras/StreamQRE/StreamQRE.jar.

[3] 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 *2016 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 85–92, Oct 2016.

[4] H. Abbas, A. Rodionova, E. Bartocci, S. A. Smolka, and R. Grosu. Quantitative regular expressions for arrhythmia detection algorithms. In *Proc. of CMSB 2017: the 15th International Conference on Computational Methods in Systems Biology*, volume 10545 of *LNCS*, pages 23–39. Springer, 2017.

[5] R. Alur, D. Fisman, and Raghothaman. Regular programming for quantitative properties of data streams. In *Proc. of ESOP 2016: the 25th European Symposium on Programming Languages and Systems*, volume 9632 of *LNCS*, pages 15–40. Springer, 2016.

[6] R. Alur and K. Mamouras. An introduction to the StreamQRE language. *Dependable Software Systems Engineering*, 50:1, 2017.

[7] E. Asarin, P. Caspi, and O. Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, Mar. 2002.

[8] E. Bartocci, L. Bortolussi, M. Loreti, and L. Nenzi. Monitoring mobile and spatially distributed cyber-physical systems. In *Proc. of MEMOCODE 2017: the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pages 146–155. ACM, 2017.

[9] E. Bartocci, L. Bortolussi, and G. Sanguinetti. Data-driven statistical learning of temporal logic properties. In *Proc. of FORMATS 2014*, volume 8711 of *LNCS*, pages 23–37. Springer, 2014.

[10] E. Bartocci, J. V. Deshmukh, A. Donzé, G. E. Fainekos, O. Maler, D. Nickovic, and S. Sankaranarayanan. Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 135–175. Springer, 2018.

[11] E. Bartocci and Y. Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*. Springer, 2018.

[12] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. Introduction to runtime verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 1–33. Springer, 2018.

[13] R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, C-20(2):149–153, 1971.

[14] L. Bozzelli and C. Sánchez. Foundations of boolean stream runtime verification. In *Proc. of RV 2014: the 5th International Conference on Runtime Verification*, volume 8734 of *LNCS*, pages 64–79, 2014.

[15] L. Brim, P. Dluhos, D. Safránek, and T. Vejpustek. STL-*: Extending signal temporal logic with signal-value freezing operator. *Inf. Comput.*, 236:52–67, 2014.

[16] S. Bufo, E. Bartocci, G. Sanguinetti, M. Borelli, U. Lucangelo, and L. Bortolussi. Temporal logic based monitoring of assisted ventilation in intensive care patients. In *Proc. of ISoLA*, volume 8803 of *LNCS*, pages 391–403. Springer, 2014.

[17] A. Chakarov, S. Sankaranarayanan, and G. Fainekos. Combining time and frequency domain specifications for periodic signals. In *Runtime Verification: Second International Conference, San Francisco, CA, USA*, pages 294–309, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[18] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. LOLA: Runtime monitoring of synchronous systems. In *Proceedings of the 12th International Symposium of Temporal Representation and Reasoning (TIME 2005)*, pages 166–174. IEEE Computer Society Press, 2005.

[19] A. Donzé, O. Maler, E. Bartocci, D. Nickovic, R. Grosu, and S. A. Smolka. On Temporal Logic and Signal Processing. In *Proc. of ATVA 2012, Thiruvananthapuram, India, October 3-6*, volume 7561 of *LNCS*, pages 92–106, 2012.

[20] T. Ferrère, O. Maler, D. Nickovic, and D. Ulus. Measuring with timed patterns. In *Proc. of CAV 2015: the 27th International Conference on Computer Aided Verification, Part II*, volume 9207 of *LNCS*, pages 322–337. Springer, 2015.

[21] I. Haghighi, A. Jones, Z. Kong, E. Bartocci, R. Grosu, and C. Belta. Spatel: A novel spatial-temporal logic and its applications to networked systems. In *Proc. of HSCC '15: the 18th International Conference on Hybrid Systems: Computation and Control*, pages 189–198. ACM, 2015.

[22] Z. Jiang, H. Abbas, K. J. Jang, M. Beccani, J. Liang, S. Dixit, and R. Mangharam. In-silico pre-clinical trials for implantable cardioverter defibrillators. In *2016 38th Annual International Conference*

of the IEEE Engineering in Medicine and Biology Society (EMBC), pages 169–172, Aug 2016.
[23] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 311–322, 2005.
[24] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *Proc. of FORMATS/FTRTFT 2004*, volume 3253 of *LNCS*, pages 152–166, 2004.
[25] S. Mallat and W. L. Hwang. Singularity detection and processing with wavelets. *IEEE Transactions on Information Theory*, 38(2):617–643, March 1992.
[26] S. G. Mallat. *A Wavelet Tour of Signal Processing, Third Edition: The Sparse Way*. Academic Press, 2008.
[27] K. Mamouras, M. Raghothaman, R. Alur, Z. Ives, and S. Khanna. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In *Proc. 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 693–708, 2017.
[28] L. Nenzi, L. Bortolussi, V. Ciancia, M. Loreti, and M. Mieke. Qualitative and quantitative monitoring of spatio-temporal properties. In *Proc. of RV 2015: the 6th International Conference on Runtime Verification*, volume 9333 of *LNCS*, pages 21–37. Springer, 2015.
[29] St. Jude Medical. Bradycardia and tachycardia devices, merlin patient care system, help manual. *St. Jude Medical Product Manuals*, 2017.
[30] R. X. Stroobandt, S. S. Barold, and A. F. Sinnaeve. *Implantable Cardioverter - Defibrillators Step by Step*. Wiley, 2009.
[31] C. D. Swerdlow, S. J. Asirvatham, K. A. Ellenbogen, and P. A. Friedman. Troubleshooting implanted cardioverter defibrillator sensing problems I. *Circulation: Arrhythmia and Electrophysiology*, 7(6):1237–1261, 2014.
[32] D. Ulus. Montre: A tool for monitoring timed regular expressions. In *Proc. of CAV 2017: the 29th International Conference on Computer Aided Verification*, volume 10426 of *LNCS*, pages 329–335. Springer, 2017.
[33] D. Ulus, T. Ferrère, E. Asarin, and O. Maler. Timed pattern matching. In *Proc. of FORMATS 2014: the 12th International Conference on Formal Modeling and Analysis of Timed Systems*, volume 8711 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2014.
[34] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST '10)*, pages 498–507. IEEE, 2010.
[35] J. Zdarek and C. W. Israel. Detection and discrimination of tachycardia in ICDs manufactured by St. Jude Medical. *Herzschrittmachertherapie + Elektrophysiologie*, 27(3):226–239, Sep 2016.

Houssam Abbas (M'01) received his B.E. in Computer Engineering from the American University of Beirut (Lebanon) and his M.S. and Ph.D. in Electrical engineering from Arizona State University. He was a Design Automation engineer in the SoC Verification group at Intel from 2006 to 2014. Houssam is currently a postdoctoral fellow in the Department of Electrical and Systems Engineering at the University of Pennsylvania. His research interests are in the verification, control and conformance testing of Cyber-Physical Systems. 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.

Alëna Rodionova received her B.S. and M.S. in Mathematics from the Siberian Federal University (Russia) in 2012 and 2014, respectively. Before joining the University of Pennsylvania in 2017, she was with the Cyber-Physical Systems Group at TU Wien. Alena is currently a doctoral student in the Department of Electrical and Systems Engineering at the University of Pennsylvania. 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.

Konstantinos Mamouras completed his undergraduate studies in Electrical and Computer Engineering at the National Technical University of Athens, obtained an M.Sc. in Computer Science from Imperial College London, and a Ph.D. in Computer Science from Cornell University. He is currently a postdoctoral researcher in the Department of Computer and Information Science at the University of Pennsylvania. His research interests lie in the area of programming languages for data stream processing, and logical approaches for program verification.

Ezio Bartocci is a tenure-track assistant professor at the Department of Computer Engineering at TU Wien. The primary focus of his research is to develop formal methods, computational tools and techniques that support the modeling and the automated analysis of complex computational systems, including software systems, cyber-physical systems and biological systems.

Scott A. Smolka is SUNY Distinguished Professor of Computer Science at Stony Brook University and a Fellow of the European Association of Theoretical Computer Science. His research interests include the formal modeling and analysis of complex systems. He is the lead PI and director of CyberCardia, the NSF CPS Frontier project on the formal analysis of cardiac medical devices.

Radu Grosu is a full Professor, and the Head of the Cyber-Physical Systems Division of the Faculty of Informatics of the Technische Universität Wien. He is also a Research Professor at the Department of Computer Science of the State University of New York at Stony Brook, USA. His research interests include the modeling, analysis and control of cyber-physical systems and of biological systems. The applications focus includes smart mobility, Industry 4.0, smart buildings, smart agriculture, smart health care, smart cities, IoT, cardiac and neural networks, and genetic regulatory networks.