QMine: A Framework for Mining Quantitative Regular Expressions from System Traces

Pradeep K. Mahato
Department of Computer Science
The University of British Columbia
BC, Canada
pradeep.mahato@alumni.ubc.ca

Apurva Narayan
Department of Computer Science
The University of British Columbia
BC, Canada
apurva.narayan@ubc.ca

***Abstract*—Dynamic behavior of real-time systems and the ability to distinguish between normal and abnormal behavior is critical in safety-critical systems. Temporal patterns define the order of occurrence of events. Temporal properties help draw insights over system specifications. However, given the complexity of modern-day software in cyber-physical systems, the specifications are either not specified or loosely specified.**

We propose a framework for automating the task of mining temporal specifications from system traces with both events and quantitative values. Our framework, QMine, is an online property mining framework that extracts properties specified in the form of Quantitative Regular Expression (QRE) templates. QMine is shown to be sound and complete. Moreover, we evaluate our framework using real-world industry-standard traces such as Arrhythmia dataset.

***Index Terms*—Software and its engineering, Software maintenance tools, Specification Mining, Quantitative Regular Expression Mining, Real Time Systems**

I. Introduction

Modern-day software systems are complex and produce an exorbitant amount of data. In a recent study [1], it is found that every day, just embedded software is used in over 41.2 billion Internet of Things devices across the globe producing 5 quintillion bytes of data (that's 2.5 followed by 18 zeros). Software engineers, data scientists, and practitioners use this data for a variety of tasks such as automated reasoning, program comprehension, anomaly detection, intrusion detection, compliance checking, and so on. Moreover, the software on these systems is extremely complex and continuously evolving which prohibits and discourages the developers to explicitly provide software specifications for these systems [2]. Lack of proper specifications for software is a prime factor for increasing costs of software maintenance where developers spend a large amount of time to understand the behavior of these systems. Dynamic analysis is an area of software engineering that involves the processing of output from software to understand the underlying behavior rather than analyzing the source code (i.e. static analysis). Software engineering has evolved into a multidisciplinary field with increased penetration of data mining and machine learning techniques [3], [4].

The temporal behavior analysis of a program is an extensively studied topic [5]–[7]. It involves mining temporal patterns [8], automated program comprehension, and associated tasks such as anomaly detection, intrusion detection, and others. In most cases, temporal pattern mining techniques are aided by association rule mining for extracting relevant temporal properties [9]. Mined specifications are valuable since they can be used for a wide variety of activities in the software development life cycle (SDLC). These activities include software testing [10], automated program verification [11], anomaly detection [12], debugging [13], etc. Further, mined specifications can also assist in automated verification techniques because they provide an easy and user-friendly way to describe programs' specifications.

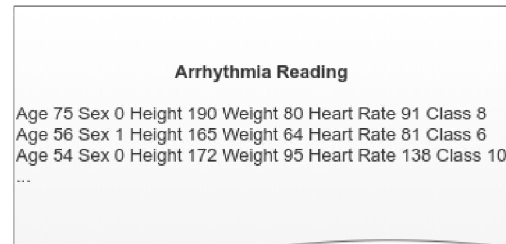

Fig. 1. Sample Arrhythmia Data with both quantitative and qualitative values

For instance, consider Figure 1, which presents preprocessed electrocardiogram or ECG readings for arrhythmia patients. It may be important for these traces to satisfy temporal property such as "Age, sex, and body dimensions are always followed by heartbeat and patient's class". Moreover, the heartbeat should be within certain statistical limits. Most research in temporal specification mining focuses on the ordering of events [14]–[18]. Such as, a *request* is always followed by a *response*. However, combining temporal properties with a quantitative stream has been explored using the formulation of quantitative regular expressions in QRE [19] and its applications in real-time monitoring systems such as StreamQRE [20], [21].

In this work, we propose a novel framework, QMine, for mining temporal properties in the form of quantitative regular expressions (QRE) from system trace. Our model takes a custom QRE specification template along with a trace log. The mining of QREs from system traces provides a framework to describe system behavior where both sequences of event occurrence and the quantitative stream of data are critical.

978-1-7281-8915-4/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS-C51114.2020.00070

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on January 14,2022 at 21:05:52 UTC from IEEE Xplore. Restrictions apply.

We begin with defining event, trace, and logs followed by our template formalism QRET. We then define each of our vital algorithms and provide guarantees over time and space. We use the association rule mining technique of *support* and *confidence* to evaluate the degree to which our QRET instances are satisfied.

The major contribution of our work can be summarized as:

- Present a novel framework for mining a special class of formalism [Quantitative Regular Expression(QRE)] from system traces
- Provide algorithmic characterization of our approach w.r.t space and time
- Show that our approach is *sound*, *complete*, and *scalable*
- Evaluate and Validate the framework on industry strength systems

The remainder of the paper is organized as follows, Section II provides overview of relevant research, Section III provides necessary preliminaries for understanding our mining framework, Section IV provides details of our framework, Section VI presents experiments and evaluation of our approach on synthetic and real-world traces, finally we provide conclusion and future work in Section VII.

II. Literature Review

Mining specification/requirements of computer programs is a well-studied topic in computer science [22]. In software engineering, a myriad of specification mining techniques exists depending on the kind of specifications such as automata, temporal rules, sequence diagrams, and others [22]. They also vary depending on the type of input provided to the models where static analysis or model checking takes source code as input whereas, dynamic analysis techniques mine from the execution trace.

Mining likely temporal specifications of programs from system traces have become popular [23]. Critical and commonly occurring behavioral patterns are typically provided to the mining frameworks, which then mine system specifications of that form. The mining techniques identify a set of specifications that are satisfied by traces w.r.t. certain criteria.

Many programs lack formal temporal specifications, and mined specifications are therefore valuable since they can be used for a wide variety of activities in the software development life cycle (SDLC). These activities include software testing [10], automated program verification [11], anomaly detection [12], debugging [13], etc. Further, mined specifications can assist in automated verification techniques because they provide an easy and user-friendly way to describe programs' specifications. As argued by Ammons et. al. [14], automated verification techniques are unlikely to be widely adopted unless cheaper and easier ways of formulating specifications are developed.

The work is inspired by [19] where an abstract language of QRE is presented for specifying complex numerical queries over data streams and [24] where a framework is presented for mining timed regular expressions from system traces. StreamQRE is an application of QREs for monitoring and is presented in [20], [21]. It is an attempt for computing quantitative summaries from data streams in real-time. With increasing complexity and obstructions arising for real-time decision making in emerging IoT applications, the author of the paper provides a streaming framework that aims to solve this specific problem. StreamQRE aims in providing statistical analysis in real-time by processing system trace log.

TRE mining framework present in [24] mines for all possible patterns in a system-generated trace file. This provides both the flexibility and rigorousness to the user. But it is limited to event traces. In another work, authors mine properties in the form of nested words by using a restricted class of context-free grammar called the nested word automaton from system traces [25].

Generic LTL Specification Mining [2] presents a framework called Texada to mine for Linear Temporal Logic to understand software behavior. Texada either uses linear mining or map-based mining to summarize system behaviors. Once a temporal template is provided, Texada mines strictly matching the given template.

Our framework QMine can analyze all possible patterns for a user-defined expression while simultaneously recording quantitative measures for post-mining analyses. QMine records all the quantitative measurements for every individual pattern in the most simplified form possible eventually providing the possibility to query the mined properties for post-analysis.

III. Background

We begin by formally describing what we mean by a trace, a log, and an event. Subsequently, we define the formalism of quantitative regular expressions as used in our framework.

Definition 1 (Event). *The alphabet of events is a finite alphabet of strings.*

Definition 2 (Trace). *Group of events forms a trace.*

Definition 3 (QTrace). *A series of traces with quantitative values between them forms a trace file with mixture of events and quantitative values.*

Events are a representation of the system and user behavior. We use α_i to represent an event and Σ to represent global set of events. Therefore, a system with n unique events will have $< \alpha_1, \alpha_2, \cdots, \alpha_n > \in \Sigma$.

Most industrial and IoT systems generate traces that are in the form of log files and sensor data. Temporal property miners can be used to extract useful information from log files and subsequently statistical analysis can be performed on sensor data. Some research addresses both events in conjunction with quantitative values for monitoring systems behavior [20]. There is no framework reported to date that mines specifications/properties of systems that comprise of both events and quantitative values.

Quantitative regular expressions have been proposed by [19] as a high-level declarative language for specifying queries over data streams in a modular way. It has been shown that QREs have good theoretical properties and are efficient for streaming applications. Therefore, the most intuitive application for QREs is in the runtime monitoring of queries in streaming

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on January 14,2022 at 21:05:52 UTC from IEEE Xplore. Restrictions apply.

data. However, this framework does not mine for all possible patterns.

Our QMine framework addresses this issue of mining properties from system traces that combine both events and signals simultaneously. The framework mines the input pattern provided using a specific template as defined below.

Definition 4 (QRET). *A QRET is a valid template if it contains at least two events* $(\alpha_i, \alpha_j) \in \Sigma$ *and a series of* m *quantitative values* $q_1 \cdots q_m$ *bounded by* α_i *and* α_j

More precisely, the most rudimentary form of *QRET* can be explained as QRET = $\alpha_1 \mathbb{R} \alpha_2$, where $\mathbb{R}$ is a set of real numbers.

QMine is a template-based mining framework. We use the term *event variable* to denote a placeholder for an event and M to represent a quantitative value. For instance, a QRET $0 \cdot M \cdot 1$ represents *"an event* 0 *is always followed by a quantitative value followed by an event* 1 *"*. It is important to note here that M represents the quantitative part of QRET that can overload a variety of mathematical operations. We use τ to denote the number of event variables. In the above example, τ is 2.

Definition 5 (Binding). Let V be a finite set of events in a QRET and Σ represent alphabet of events. Then a binding is a function $b : V \rightarrow \Sigma$. The number of unique event variables in QRET is the dimension of the template and is represented by τ.

Definition 6 (Support Potential) The support of a QRET instance π on a trace t is the number of time points of trace t which falsify π.

Definition 7 (Support) The support of a QRET instance π on a trace t is the number of time points of trace t which *do not* falsify π.

Definition 8 (Confidence) The confidence of QRET instance π on a trace t is the ratio of trace support to trace support potential.

The ranking component we use is a combination of support and confidence. The effectiveness of selecting a meaningful subset of specifications depends on picking a good set of thresholds.

Since the total number of mined QRET instances is often very large in real systems, we would ideally keep the confidence value at 100%. However, the motivation to reduce this threshold slightly is due to the presence of imperfect traces. Traces can be imperfect as a result of dropped events or execution of faulty programs. In such cases, dominant properties may not be perfectly satisfied in the collected traces. Reducing the threshold will thus include dominant properties from imperfect traces.

We examine different threshold values for both support and confidence and evaluate the best thresholds for reducing all feasible properties to adjust the dominant and interesting properties.

A. Quantitative Regular Expressions

The work on StreamQRE [20] presents a framework that monitors queries in the form of modular patterns and an operation to be performed on the quantitative values over streaming data provided in the form of QREs. Every pattern must be defined in terms of QRE combinators.

Our framework supports combinators ($atomic$, $iter$, $split$, and $choice$) from the work of Alur et. al. [19], [20]. QMine uses Ragel [26] for implementing the finite state machines. Ragel provides support for injecting user-defined action for every state change during transition within a finite state machine.

IV. Methodology

Figure 2 provides a high level working of QMine framework. The QMine framework requires two inputs: a real-world trace log in the QTrace format and a QRE template (QRET). These 2 inputs are fed to the *normalizer* module that pre-processes the trace as outlined in section IV-A, if needed. Subsequently, we have the *QMine*$_{\text{core}}$ module that

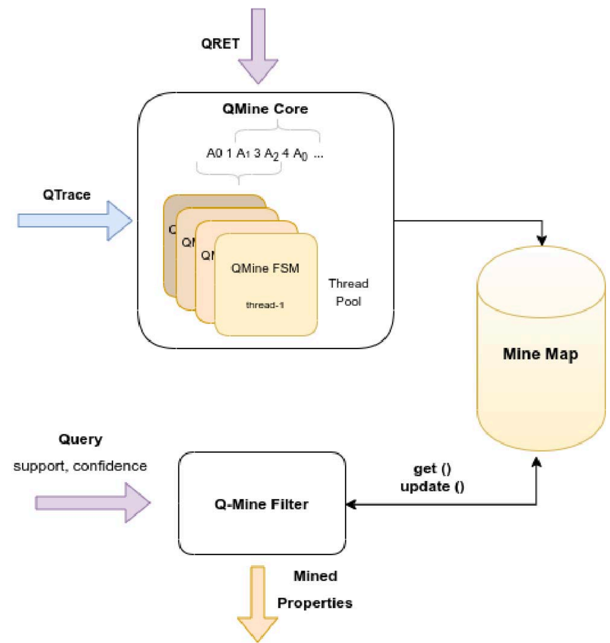

Fig. 2. QMine Worklfow

maintains the thread pool of the finite state machines for *QMine* instances. Each *QMine* finite state machine instance mines a single sub-sequences of *QRET*. Section IV-B provides further details of the *QMine*$_{\text{core}}$ algorithm. *QMine*$_{\text{core}}$ outputs a *mine-map* ξ that holds all instances of mined patterns and their related statistical details. The novelty and substantial advantage of *QMine* are in the fact that once all instances are mined and stored in ξ, one can then query the mined results for other patterns that prevent from going over the entire trace again. The mined results provide ease of querying for various post-execution result analysis.

The *QMine_filter* module allows users to analyze for any pattern of interest with ability to overload a statistical operation (currently QMine support μ (mean), σ (variance), `min`, and `max`) of interest to the engineer. Let us denote the pattern template of interest as *QRET*. The event mapping dictionary

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on January 14,2022 at 21:05:52 UTC from IEEE Xplore. Restrictions apply.

η and *mine-map* ξ are available through the *normalizer* and the *QMine*$_{\text{core}}$ module.

Any log trace would typically consist of three types of events: (1) Event A marking the start of the pattern (2) measurement M, a set of numerical values (3) Event B marking the end of the pattern, and similar subsequent occurrences.

To illustrate our approach lets us consider two regular expressions R_1 and R_2 as pre-condition and post-condition patterns of interest with a quantitative measurement between the two expressions as M. Hence, QRET for such a scenario can be represented as $(R_1 \cdot M \cdot R_2)$. Let us consider both R_1 and R_2 have a single variable then our expression becomes $0 \cdot M \cdot 1$ where 0 and 1 are placeholders for events in the trace alphabet Σ.

Overall, QMine framework has four modules which define its working. These can be listed as below:

- *Normalizer*
- *QMine*$_{core}$
- *QMine Filter*
- *QRE Parser*

In the following sections we shall describe in the detail the four modules of the QMine framework.

A. Normalizer

The main goal of the normalizer is preprocessing of the input trace. This is required in case the set of events Σ is not available priori to mining properties from the traces and if the trace is not in the QTrace format. However, in most cases the alphabet of the system under consideration is available.

B. QMine Algorithm

At the heart of our QMine property mining framework is $QMine_{core}$ driven by the QMine algorithm. This algorithm takes in the system trace in the QTrace format and QRE template to mine specifications that conform to the provided QRET. Our algorithm uses Ragel [26] to implement finite state machines and a multi-threading approach to achieve speedup. Algorithm 1 provides a pseudo-code of the module.

Algorithm 1: QMine(QRET, QTrace, Σ, ξ)

```
  Input: QRET, QTrace, alphabet Σ, mine-map ξ
  Result: Mined properties for QTrace
1 Generate all permutations of QRET from Σ
2 Initialize Finite State Acceptor (FSA)
3 Set all threads with ξ and the FSA
4 Divide QTrace segments per thread
5 foreach segment ∈ QTrace segments do
6 |   mineInternal (segment, ξ, FSA)
7 end
```

The main objective of the Algorithm 1 is to mine all occurrences of all patterns specified by a template in a given QTrace. We provide our QTrace, alphabet set Σ, and QRET as input. We initialize the thread pool vector along with finite state machines by generating all possible permutations of QRET using the binding function. We subsequently initialize our threads and maintain their counts `th-count`. We begin with recursively extracting QTrace segments that match the structure of the given *QRET* template and assign them to a thread for processing. The algorithm iterates over each QTrace segment where each thread matches the segment with the FSA to find if it reaches an accepting state or an error state. Each thread returns a value `TRUE` if its underlying finite state machine reaches an accepting state. The thread that returns `TRUE`, is removed from our thread pool, and the *mine-map* is updated to reflect the results. The *mine-map* ξ, is available to each thread and upon reaching the accepting state of the finite state acceptor all the tracked quantitative values are inserted in ξ for further statistical operations as desired.

All threads maintain mutual exclusion with respect to the *mine-map*, ξ. Consider a thread t, where $t \in \langle t_0, t_1, \cdots t_{\textit{th-count}-1} \rangle$. All thread execute instructions in their respective memory space except for operations on *mine-map* ξ. The update operation on ξ only happens in case a full match is found for a pattern in the trace segment processed by the thread. The framework ensures that each sub-sequence is processed by one and only one thread at a time. This guarantees that more than one thread's finite-state acceptor never reaches an accepting state. Therefore, there would be only one update operation on ξ at any time. Thus a race condition over *mine-map* ξ is guaranteed to not happen.

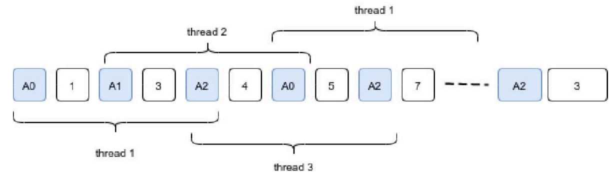

Fig. 3. Thread distribution in QMine

Figure 3 pictorially displays the allocation of segments of QTrace per thread. Each thread handles only a single segment at a time.

Multi-threading provides parallelism and speed up to our mining framework. Our scheme analyzes the available resources on the computer system and accordingly modulates the number of threads required for program execution.

Algorithm 2 shows the working of an individual thread for each QTrace segment. The algorithm takes a trace segment, *mine-map*, and finite-state acceptor as the input. Finite-state acceptor processes the trace segment and if `FINISH` state is reached then *mine-map* is updated to reflect the count of the mined pattern. The *mine-map* gets a new pattern added to the list if it does not exist in the list of already mined patterns. Our approach ensures that all patterns are mined and accounted for in the system.

C. QMine Filter

Once the entire trace has been processed by the QMine framework, the summary of the patterns is represented in *mine-map*. One can develop a whole range of queries that can be

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on January 14,2022 at 21:05:52 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2: mineInternal(segment, ξ, FSA)

Input: QTrace Segment, ξ, FSA
Result: Mined property satisfied by a QTrace segment

```
1  action EVENT
2      record patterns for QTrace Segment match
3  action NUM
4      push digit to ξ
5  action FINISH
6      match overall pattern
7      push ξ into global shared space & increment count
8      return TRUE // Accepting state
9  action ERROR
10     remove inserted values from ξ
11     return FALSE // Error State
```

answered by this abstraction, *mine-map*, of the system trace. For instance, we can now perform a variety of mathematical operations on the quantitative values in each instance of the mined pattern. In this work, we show how a filter can evaluate an average μ and standard deviation σ for each pattern. The QMine filter can easily be extended to a variety of metrics that can help summarize the system trace under consideration. QMine filter also acts as a ranking module for mined properties where it uses the combination of support and confidence provided by the user to extract dominant properties.

Algorithm 3: evaluate(Query,ξ)

Result: List of dominant patterns with μ, σ
Input: Query Pattern, ξ, Support, Confidence

```
1  Filter patterns with Support & Confidence threshold
2  if Query Pattern ∈ ξ with Support & Confidence then
       /* compute μ and σ                          */
3  end
```

Algorithm 3 takes as input a user-defined pattern instance in the QRET format. The query searches through the *mine-map* in linear time and provides instance counts, μ and σ for each of the quantitative variables in the QRET (or queried instance) with pre-specified or custom values for support and confidence rules. Moreover, the negation of a variable in QRET is handled during post-processing by querying for a negated instance in the QMine's filter module.

D. QRE Parser

QREs are a declarative way of specifying quantitative regular expressions in a hierarchical way as introduced by Alur et. al. [21]. We provide an interface the allows engineers and developers to provide patterns and QRETs in a hierarchical format using the formalism provided by [19], [20]. The goal is to provide options to users to input their queries in either format, however keeping the mining framework generic for quantitative data streams. Our parser can convert hierarchical QREs to QRET format and vice-versa. This provides a wider advanced audience with familiarity with QREs and the ones without it to use the tool for their analysis of complex systems.

We present some basic parsing examples to show the relationships between QRE operators defined in Section III-A and QRET structure. Some basic keywords are:

- **E** : E corresponds to an event placeholder
- **M** : M corresponds to quantitative value
- **iterOp** : iterOp corresponds to iterate operation.

$$iterOp : (x, y) \rightarrow x + y$$

- **splitOp** : splitOp corresponds to concatenation operation.

$$splitOp : (x, y) \rightarrow z$$

Example 1. Create template $R_1 \cdot M \cdot R_2$ where $M \leftarrow [0-9]+$

Solution: In the given template, R_1 and R_1 are pre and post condition for pattern of interest with a quantitative measurement M in between them. This can be converted to the QRE formulation based and expressed as

$$R_1 : QRE\langle QTrace, \tau\rangle = atom(R_1 \leftarrow R_1.type = E) \quad (1)$$

$$M : QRE\langle QTrace, \tau\rangle = atom(M \leftarrow M.type = M) \quad (2)$$

$$R_2 : QRE\langle QTrace, \tau\rangle = atom(R_2 \leftarrow R_2.type = E) \quad (3)$$

Using the atomic expression from Equation 1, 2 and 3, we can perform split as

$$split_1 : QRE\langle QTrace, \tau_1\rangle = split(R_1, M, splitOp) \quad (4)$$

Thus using Equation 4 and associative property, we finally deduce our template as

$$\begin{aligned} QRET : QRE\langle QTrace, \tau\rangle &= split(split_1, R_2, splitOp) \\ &= split(\cdots, M, R_2, splitOp) \quad (5) \end{aligned}$$

Using property 5, large and complex QRET can be formed. For further understanding of QREs one may refer to the work [19], [20].

V. Discussion

In this section, we would like to show that our algorithm is correct, sound, and complete. Since QMine mines all possible instances of a QRET from the system trace. We argue that the technique we proposed and described in this article is sound and complete. By sound, we mean that a mined specification reported by our algorithms satisfies the given thresholds for confidence and support in the provided input traces. This is the case because our algorithm continuously keeps track of the success and reset rates for each QRE instance in the result *mine-map*. The rates are direct results of the execution of the acceptor automaton for the QRE on a trace and are used for calculation of the confidence value for the QRE instance. Given that the reported rates and the derived confidence value are valid, it follows that the derived support value is valid as

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on January 14,2022 at 21:05:52 UTC from IEEE Xplore. Restrictions apply.

well. Therefore, any mined specification that our algorithms report satisfies the confidence and support constraints.

By characterizing our technique as complete, we mean that our algorithms report all the mined specifications that comply with the given thresholds for confidence and support in the provided input traces. Upon reading an event sequence from the trace, our technique examines every QRE instance that can be affected. All the evaluation results are contained within the *mine-map*, the results for every QRE instance will be considered when evaluating against the confidence and support thresholds. Therefore, the algorithm will report all the QRE instances, the mined specifications, that comply with the confidence and support constraints.

A. Memory Requirements

The proposed quantitative regular expression mining technique requires memory space for the *mine-map*, the finite state acceptor, and the trace contents. The storage requirements for the *mine-map* are directly influenced by τ and Σ. The *mine-map* scales up to hold the evaluation results of every QRE instance generated by binding the events from Σ to the τ event variables in the QRET. There are Σ^τ possible TRE instances, thus we need $O(\Sigma^\tau)$ space to hold the acceptor results.

The space requirements of the *mine-map* grows proportionally to τ, the number of symbols in the desired property. Complex properties that encode a relationship between a large number of events will result in higher values of τ. Increasing values of τ in turn will result in exponential space growth. However, the majority of properties of interest represent relationships among a few events only [27], [28]. Therefore, in general, the dimension τ of the matrix will remain small.

The number of unique events Σ in the trace will depend on the complexity of the program or system. However, Σ has a much smaller effect on the growth of the *mine-map* since it only affects the base number in the exponential space complexity.

Next, we consider the storage requirements for the finite state acceptor used to evaluate the QRE. The storage required for the FSM is proportional to the number of its states [29]. If τ is the length of the QRE, then an equivalent NFA has $O(\tau)$ states and an equivalent DFA has at most $O(\Sigma^\tau)$ states. Similar to the *mine-map*, complex properties that encode a lot of relationships between events result in exponential space growth for the FSM. However, a majority of interesting properties will remain simple [28].

Lastly, the storage requirements for the trace are equivalent to the length of the trace, $O(L)$, and for storing our *mine-map* where the worst-case space requirements are $O(\sigma^\tau)$. Our mining approach examines one event from the trace at a time, without needing to store any past or future events for context. It also does not perform any changes in the contents of the trace. Thus the trace is stored only once and never replicated.

The proposed technique is thus scalable with the growing trace size L and with the growing event alphabet Σ. This is important since we have no control over the events and traces generated by real systems. The technique is sensitive to growing values of τ. However, as we already mentioned, in practice this will not be a concern since properties remain relatively simple as shown in the evaluation section with industry grade case studies.

Complexity Analysis	
Time	O($\Sigma^\tau + L$)
Space	O($\Sigma^\tau + L$)
Execution	O($\Sigma^\tau + L/(\texttt{th-count}) + th_{oh}$)

TABLE I
QMINE COMPLEXITY

Table I shows the space and time complexity for QMine. Execution complexity refers to the time taken for parallel execution whereas time complexity states the theoretical time bound for the same. Expression th_{oh} represents the multi-threading overhead in the system. It is important to highlight at this point that the time and space complexity of QMine is linear with respect to the length of the trace which makes is a promising approach when mining over large system traces.

VI. EVALUATION

In this section, we demonstrate the performance and scalability of our approach using a set of real system traces and a set of synthetic traces. The implementation is done using C++. Ragel[1] is a framework we used to synthesize Finite State Automata for QREs. For the experiments, we used a single eight-core Intel i7-2630QM CPU with 8 GB of RAM running on Ubuntu 18.04 64 bit with GCC version 7.5.0. With the application of *QMine* software on the real-world Arrhythmia dataset, our experiments demonstrate the performance and scalability by using different combinations of QRET and trace lengths.

A. Evaluation on Real World Traces

1) Arrhythmia Dataset: Arrhythmia generally occurs with an irregular heartbeat pattern [30], [31]. This can be due to improper flow of electrical signals by nerves to the heart. Although the majority of arrhythmia cases are harmless, the occurrence of the same may be a precursor to a major health problem. The existence of arrhythmia leads to higher health risks. Therefore it is important to identify such irregularities before time.

For our analysis, we used health records from the UCI Machine Learning Repository [32]. These are 452 patient records each having 279 attributes. Artery disease is divided into 15 classes with normal class denoted by 1 while class 2 refers to coronary artery diseases and class 3-15 exhibits different groups of arrhythmia. For demonstration purposes, we have only taken 7 attributes namely Age, Sex, Height, Weight, Heart Rate, and Class. For much detailed analysis, one could perform the same procedures on a larger dataset. The point of interest is the distribution of height, weight, heart rate, and age for different classes of patients. While such analysis does give insight into the general trend, it also provides a quick

[1] http://www.colm.net/open-source/ragel/

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on January 14,2022 at 21:05:52 UTC from IEEE Xplore. Restrictions apply.

high-level understanding of the distribution of readings across the dataset for rapid medical identification in the future. We provide our results by finding the mean and variance.

Age	Sex	Count	Height (cm)		Weight (kg)		Heart Rate (bpm)		Class
	0 -Male 1-Female		Mean	Variance	Mean	Variance	Mean	Variance	Class
0-39	0	36	174.194	24.8789	77.8333	124.806	92.0833	69.6319	1
	1	118	159.356	14.3987	60.8305	131.175	81.4661	52.5031	
	0	5	167.2	5.36	58	49.2	98.8	21.36	2
	1	12	161.333	65.2222	62.1667	270.472	90.5	63.5833	
	0	13	168.154	6.74556	69.3846	282.544	95.8462	19.2071	Not 1
	1	28	162.5	42.25	61.6429	244.944	83.2857	108.347	
40-99	0	123	171.407	42.5665	76.0569	187.289	89.9024	60.771	1
	1	183	160.541	25.9532	68.0383	188.933	80.9071	55.6253	
	0	29	169.103	40.9893	80.8276	746.212	94.8621	96.3948	2
	1	40	158.7	20.61	73.7	172.81	84.45	66.4475	
	0	115	170.13	43.7656	76.1043	249.641	96.8174	374.81	Not 1
	1	77	157.987	21.2596	71.2208	144.432	90.6883	573.955	

TABLE II
SUMMARY OF QUANTITATIVE VALUES ON THE ARRYTHMIA DATASET

Table II, we can see detailed breakdown for each age-group and attributes. The above analysis was done with *QRET* = 0M1M2M3M4M5M6, i.e with $\tau = 7$ and $\Sigma = 7$ due to only 7 possible events. After our mine map, ξ was created, we were interested in various combinations of the pattern. For example, a query of $a[4-9][0-9]b1c[0-9]+d[0-9]+e[0-9]+f2g$ would denote female patients of Coronary Artery for all age group of range $40-99$ and females

For females of age group $0-39$, we see that there is variability in heart rate. More specifically usual heart rate is quite low in the normal case while those with coronary artery diseases exhibit a bit higher heart rate. While our variance is not so significant but the insufficient sample size for females in such division could be improved with a larger data set. Nevertheless, the above quick insight does display a deviation in the mean heart rate for such patients. Moreover, for males of age group $40-99$, we see a clear distinction in heart rate between a normal person and arrhythmia patients. A normal person would have a heart rate of 89.9 bpm while that of a patient would be somewhere around 96.8 bpm. Our results corroborate with the details outlined in the dataset description.

B. Evaluation using Synthetic Traces

System traces related to the real-world dataset are good for validating the correctness and for quick analysis of an algorithm. It provides a better understanding and helps in strengthening the purpose of an algorithm.

We perform validation on scalability and performance using synthetic traces by adjusting the following parameters: τ, alphabet size Σ, and length of QTrace. We record the overall execution time and memory requirements.

Setup 1 (Memory and Compilation Time): In this case, we fix a QRET, $0 \cdot M \cdot 1$, where 0 and 1 are event place holders and M corresponds to a numerical quantity. We also fix the trace length but vary the alphabet length Σ. Since we compute and compile for all possible patterns using the placeholders in our QRET and alphabets in Σ, understanding the compilation time and space complexity for our program is also crucial.

QRET	Alphabet Size	Total QRE Instances	Compile Time Taken (ms)
0M1	10	90	2,840
	50	2450	2,955
	500	249500	61,274
	1000	999000	247,728
0M1M2	10	720	2,912
	20	6840	3,258
	50	117600	23,114
	100	970200	638,822

TABLE III
ANALYSIS FOR QRET SIZE, ALPHABET AND COMPILATION TIME

Table III demonstrates the time taken by *QMine* to compile different pattern combinations. For illustration purposes we have shown QRET with only $\tau = 2$ and $\tau = 3$ while varying the number of unique events Σ.

The results in Table III show an exponential growth in the time taken which is due to the generation of such a large number of patterns for the given alphabet Σ. However, this is only a one time task and most practical systems have limited alphabet size and pattern requirements.

Setup 2 (Mine Time): We evaluate the scalability of our framework *QMine* with respect to the trace length and the alphabet size.

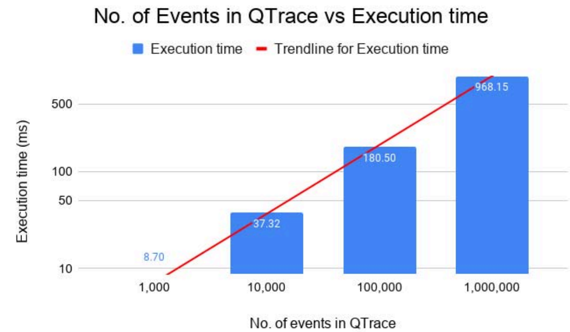

Fig. 4. Varying number of events in QTrace

Figure 4 presents comparison for time taken to mine based on total event count. For this experiment, we restricted our τ for QRET to 2 and alphabet $\Sigma = 26$ and vary the number of events in the system trace. We are interested in learning the impact of trace size in terms of the total number of events in the trace versus the mining time. We replicated our experiment 10 times to ensure reproducibility and reliability in our results. Figure 4 shows that our QMine scales linearly with increase in trace length.

In Figure 5 we present the scalability analysis of QMine by varying the alphabet length Σ and keeping other parameters constant such as the dimensionality of QRET, $\tau = 2$ and length of the trace $L = 10,000$.

The results of Figure 5 and Figure 4 provide empirical evidence for the correctness of our space-time complexity analysis. Here we see that the time complexity of our algorithm is linear with respect to the length of the trace (L) and exponential to the number of place holders (τ) in the QRET.

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on January 14,2022 at 21:05:52 UTC from IEEE Xplore. Restrictions apply.

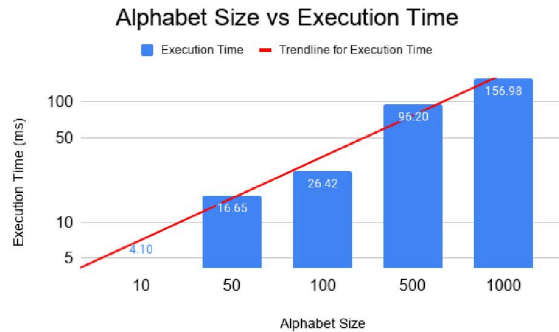

Fig. 5. Varying number of distinct event

VII. Conclusion

This paper presents a novel algorithm for mining trace log and extract meaningful summaries using the quantitative regular expression as the formalism. We present QMine, a generic framework for mining and extracting interesting patterns in a given trace log. We presented a detailed complexity analysis of the algorithm. We presented two sets of experiments to demonstrate that the algorithm is scalable, robust, sound, and complete. First, we presented experimental results on synthetically generated traces to analyze the scalability of the algorithms with an increase of variables in the QRE template, varying the number of unique and total events in the system trace. Secondly, we validate our framework on safety-critical real-world application traces.

The experimental results confirm the asymptotic analysis of our algorithm's complexity. We believe that our framework is generally applicable and is especially useful for constructing more advanced analysis tools that require QRE specification mining. In the future, we propose to improve the framework to provide a querying language that can be used by developers and researchers in analyzing and mining system specifications.

References

[1] B. Marr, "How much data do we create every day? the mind-blowing stats everyone should read," 2018.

[2] C. Lemieux, D. Park, and I. Beschastnikh, "General ltl specification mining (t)." IEEE, 2015, pp. 81–92.

[3] M. Tekieh and B. Raahemi, "Importance of data mining in healthcare: A survey." ACM, 2015, pp. 1057–1062.

[4] Y. . s. Yang, O. for Higher Education, and E. A. A. Books, *Temporal data mining via unsupervised ensemble learning*, 1st ed. Amsterdam: Elsevier, 2016;2017;.

[5] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, 1999, pp. 411–420.

[6] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," pp. 558–565, 1978.

[7] J. v. Leeuwen, *Handbook of theoretical computer science*. New York;Amsterdam;Cambridge, Mass;: Elsevier, 1990.

[8] Wikipedia, "Temporal logic," 2019.

[9] T.-D. B. Le and D. Lo, "Beyond support and confidence: Exploring interestingness measures for rule-based specification mining," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 331–340.

[10] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, "Generating test cases for specification mining," in *Proceedings of the 19th international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2010, pp. 85–96, 594101.

[11] Z. Kincaid and A. Podelski, "Automated Program Verification," in *Language and Automata Theory and Applications: 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings*, vol. 8977. Nice, France: Springer, 2015, p. 25.

[12] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the 1st India software engineering conference*. New York, NY, USA: ACM, 2008, pp. 5–14.

[13] M. Gabel and Z. Su, "Online inference and enforcement of temporal properties," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. New York, NY, USA: ACM, 2010, pp. 15–24.

[14] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," *ACM Sigplan Notices*, vol. 37, no. 1, pp. 4–16, 2002.

[15] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 501–510, 529080.

[16] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," *SIGOPS Oper. Syst. Rev.*, vol. 35, pp. 57–72, Oct. 2001. [Online]. Available: http://doi.acm.org/10.1145/502059.502041

[17] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal API rules from imperfect traces," in *Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 282–291, 592060.

[18] J. Yang and D. Evans, "Automatically inferring temporal properties for program evolution," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*. Saint-Malo, Bretagne, France: IEEE, 2004, pp. 340–351.

[19] R. Alur, D. Fisman, and M. Raghothaman, "Regular programming for quantitative properties of data streams," in *European Symposium on Programming*. Springer, 2016, pp. 15–40.

[20] R. Alur, K. Mamouras, and C. Stanford, "Modular quantitative monitoring," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–31, 2019.

[21] K. Mamouras, M. Raghothaman, R. Alur, Z. G. Ives, and S. Khanna, "Streamqre: modular specification and efficient evaluation of quantitative queries over streaming data." ACM, 2017, pp. 693–708.

[22] D. Lo, S.-C. Khoo, J. Han, and C. Liu, *Mining software specifications: methodologies and applications*. CRC Press, 2011.

[23] C. Lemieux, D. Park, and I. Beschastnikh, "General LTL Specification Mining," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. New York, NY, USA: ACM, 2015, pp. 81–92.

[24] A. Narayan, G. Cutulenco, Y. Joshi, and S. Fischmeister, "Mining timed regular specifications from system traces," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 2, pp. 1–21, 2018.

[25] A. Narayan, N. Benann, and S. Fischmeister, "Mining specifications using nested words," in *2017 6th International Workshop on Software Mining (SoftwareMining)*, 2017, pp. 9–16.

[26] A. Thurston, "Ragel state machine compiler," 2015.

[27] J. Yang and D. Evans, "Dynamically Inferring Temporal Properties," in *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '04. New York, NY, USA: ACM, 2004, pp. 23–28. [Online]. Available: http://doi.acm.org/10.1145/996821.996832

[28] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in Property Specifications for Finite-State Verification," in *Software Engineering, 1999. Proceedings of the 1999 International Conference on*. New York, NY, USA: IEEE, 1999, pp. 411–420.

[29] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation," 2007.

[30] C. Nordqvist, "What to know about arrhythmia," 2017.

[31] T. J. Bunch, J. P. Weiss, B. G. Crandall, May, and more, "Atrial fibrillation is independently associated with senile, vascular, and alzheimer's dementia," *Heart Rhythm*, vol. 7, no. 4, pp. 433–437, 2010.

[32] C. Blake and C. Merz, "UCI machine learning repository," 1998. [Online]. Available: http://www.ics.uci.edu/mlearn/MLRepository.html

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on January 14,2022 at 21:05:52 UTC from IEEE Xplore. Restrictions apply.