Mining Timed Regular Expressions from System Traces

Greta Cutulenco
gcutulen@uwaterloo.ca

Yogi Joshi
y2joshi@uwaterloo.ca

Apurva Narayan
a22naray@uwaterloo.ca

Sebastian Fischmeister
sfischme@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, ON N2L 3G1 Canada

ABSTRACT

Dynamic behavior of a program can be assessed through examination of events emitted by the program during execution. Temporal properties define the order of occurrence and timing constraints on event occurrence. Such specifications are important for safety-critical real-time systems for which a delayed response to an emitted event may lead to a fault in the system. Since temporal properties are rarely specified for programs and due to the complexity of the formalisms, it is desirable to suggest properties by extracting them from traces of program execution for testing, verification, anomaly detection, and debugging purposes.

We propose a framework for automatically mining properties that are in the form of timed regular expressions (TREs) from system traces. Using an abstract structure of the property, the framework constructs a finite state machine to serve as an acceptor. As part of the framework, we propose two novel algorithms optimized for mining general TREs and a fragment without negation. The framework is evaluated on industrial strength safety-critical real-time applications (a deployed autonomous hexacopter system and a commercial vehicle in operation) using traces with more than 1 Million entries. Our framework is open source and available online: https://bitbucket.org/sfischme/tre-mining

CCS Concepts

•**Software and its engineering** → *Software maintenance tools;*

Keywords

Timed Regular Expressions; Timed Automata; Specification Mining; Real Time Systems

1. INTRODUCTION

Temporal behavior of programs is an extensively studied topic [20, 10]. Recently, the idea of mining likely temporal properties of programs from system traces has become popular [22]. 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 [9], automated program verification [19], anomaly detection [7], etc. Further, mined specifications can assist automated verification techniques because they provide an easy and user-friendly way to describe a programs' specifications. As argued by Ammons et. al. [4], automated verification techniques are unlikely to be widely adopted unless cheaper and easier ways of formulating specifications are developed. Consequently, specification mining has gained significant attention in recent times. Different techniques have been developed for mining specifications from templates expressed using regular expressions, LTL [5], and other custom formats.

There have been both static and dynamic approaches proposed for property mining. Static property mining is found to be effective and accurate but faces challenge in scaling with the program size [2, 12]. On the other hand, dynamic property mining approaches guarantee scalability but lack in the quality of mined properties which depend on various factors such as the observed executions and the test suite used for stimulating the design [8].

There are two main subsets of dynamic property mining approaches: invariant miners [13, 14, 6, 17] and temporal property miners [4, 24, 11, 27, 25]. Of all the available tools, Daikon [14] has proven to be most successful in inferring the most probable invariants. It has been used widely for debugging, testing, documentation and maintainability [14].

A vast majority of tools for mining of temporal properties infer properties in the form of state machines. These tools learn a single complex state machine instantly and extract simpler properties from it. In [4] they learn a state machine which captures both temporal properties and data dependencies.These find utility in identifying errors and refining specifications. These also find use in automatic verification tools to find bugs in the program execution. In [24] a model is inferred representing among various components of the software. The tool generates Extended Finite State Machines from the traces of component interactions. They solve the dual purpose of modeling both data and control aspects which are useful for analysis and system verification. They generate the Extended Finite State Machines by combining classical algorithms for generating finite state machines and

SoftwareMining'16, September 3, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-4511-8/16/09...$15.00
http://dx.doi.org/10.1145/2975961.2975962

Daikon. These approaches suffer from two main drawbacks. First, mining of a single state machine from system traces is a NP-hard problem [16]. Second, extraction of formula-based properties from complex state machines still exists.

In another work [10], the authors defined a set of temporal property patterns based on case study of hundreds of real property specification. The main idea behind the exercise was to help designers unfamiliar with formal specifications and static verification approaches.

Another work based on the intuition that frequently occurring behavior that matches temporal patterns are likely to be true is the foundation of Peracotta [26, 27, 25]. The main motive behind their approach was to achieve scalability to large system traces, in mining binary specifications, in the form of automata, and post-processing them using inference rules to form complex state machines.

Another interesting attempt [23] has been to make temporal pattern mining suitable to digital circuits by changing the used timing reference to adjust to the hardware environment. The basic algorithm is same as [26] but it tries to mine all the satisfied pattern from the trace file, while in [26] they mine only the strictest pattern satisfied by the state machine.

Most of the existing research in context of mining temporal specifications focuses on the qualitative notion of time, i.e. the specifications describe an *ordering* of events. For example, a LTL specification $\Box$(request $\rightarrow \Diamond$response) specifies that a *request* event should always be eventually followed by a *response* event. Most state-of-the-art techniques do not take into account the *quantitative* notion of time, i.e. the actual duration of time between events is not considered. For safety-critical real-time systems, it is important to develop techniques that allow mining of specifications that account for the quantitative notion of time. For example, specifications for interrupt handlers, which for real-time systems must complete their executions within a set of predefined deadlines. The time constraints are of importance in such a case since a delayed response to an emitted event may lead to a fault in the system.

With the motivation to address the problem of mining specifications with an explicit notion of time, we propose a technique to mine instances of timed regular expression (TRE) templates satisfied by a given system's traces. TREs [15] extend regular expressions by providing additional operators to specify timing constraints between events. By using a set of common patterns of specifications for LTL and regular expressions, we develop the corresponding patterns for TREs. Further, we use the method proposed by Asarin et al. [15] to synthesize a timed automaton for a given TRE. The timed automaton is then used as a checker to verify whether traces satisfy the corresponding TRE.

We provide two algorithms for mining instances of TREs from their templates. Both algorithms require a TRE template and system traces as input. The algorithms then use the distinct events from the traces to replace the event variables in the TRE templates with actual events. The resultant *permutations* of the template are TRE instances. Intuitively, traces are processed against the TRE instances with the help of the timed automaton. Further, we define *confidence* and *support* as metrics to evaluate the degree to which a TRE instance is satisfied by the traces. The algorithms report only the instances which satisfy the given threshold values of confidence and support.

The first algorithm is designed for TRE templates without a negation operator. The second algorithm processes TRE templates with a negation operator. The execution times of both the algorithms' are exponential in terms of the number of variables in individual TRE templates, the first algorithm generally runs faster than the second by a certain factor, which depends upon the number of distinct events in a trace and the number of variables in a given TRE template. Our technique is sound, i.e., a mined specification reported by our algorithms actually satisfies the given thresholds of support and confidence on the provided input traces. Also, our technique is complete, i.e., our algorithms report all TRE instances which comply on given traces w.r.t. the thresholds of support and confidence.

We evaluate our technique on real-world datasets that consist of traces produced by the QNX Neutrino Real-time Operating System [1] during various runs of application software on different hardware platforms. We report the performance of our algorithms on the real-life traces in terms of the execution time. We also demonstrate the scalability and efficiency of our approach by running the implementations on synthesized traces of different sizes with different values of parameters such as the number of distinct events, the total number of events in the traces, and the complexity of the TRE templates.
The key contributions of this paper include:

- An efficient technique for extracting temporal properties with time constraints in the form of Timed Regular Expressions (TREs),
- Two novel algorithms to mine instances of TREs from given system traces. To our knowledge, this is the first technique for mining specifications with an explicit notion of timing constraints.
- Computational approaches that optimize the extraction of the specified properties for fragments of TREs.
- An analytic bound on the speedup of the optimization with an empirical validation corroborating the correctness.
- A feasibility and viability study using traces collected from operating safety-critical real-time systems showing the applicability and scalability of the approach.

2. BACKGROUND

2.1 Timed Regular Expressions

Regular expressions offer a declarative way to express the patterns for any system property or specification. Every language defined by a regular expression is also defined by a finite automaton [18]. There is a way to convert any regular expression into a non-deterministic automaton, and further to convert from a non-deterministic to a deterministic automaton. We can thus generate a classical Deterministic Finite Automaton (DFA) for any property expressed as a regular expression.

Classical automata theory handles only the *qualitative* notion of time, i.e. a sequence of events specifies the ordering of events but not the time between occurrence of these events in terms of "real time". An abstraction of this sort has been found useful for analysis of certain systems, whereas many other real time safety critical application domains require

more detailed models which include accurate timing information. For example, we might want to modify a formal specification *"a is followed by b"* to a more precise specification with timing constraints *"a is followed by b within x seconds"*. Since our focus lies on real-time safety-critical systems, we need to develop techniques for mining specifications that include the relevant timing information. We use the formalism of Timed Regular Expressions (TREs) for our purpose as it allows for defining explicit timing constraints in the model. We will formally describe our nomenclature.

DEFINITION 1 (TRACE AND EVENT). *The alphabet of events is a finite alphabet of strings. A timed sequence of events is the trace.*
The sequences of events in the trace are ordered by time stamps. The alphabet of events is defined by the system generating the traces. The events have associated meaning pertaining to the functionality of the system.

DEFINITION 2 (TIMED REGULAR EXPRESSION). *Timed regular expressions (TREs) over an alphabet Σ (also referred to as Σ expressions) are defined using the following families of rules:*

- *$\underline{a}$ for every letter $a \in \Sigma$ and the special symbol ϵ are expressions,*
- *If φ, φ_1 and φ_2 are Σ-expressions and I is an integer bound interval then $\langle\varphi\rangle_I$, $\varphi_1 \dot{\varphi}_2$, $\varphi_1 \vee \varphi_2$ and φ^* are Σ-expressions.*

The novel features here with respect to untimed regular expressions are the meaning of the atom $\underline{\alpha}$ which represents an arbitrary passage of time followed by an event α and the $<\phi>_I$ operator which restricts the metric length of the time-event sequences in $[\phi]$ to be in the interval I. It is important to note that we use TREs, as defined above, to provide specification templates.

DEFINITION 3 (TRE TEMPLATES). *A TRE template is a TRE in which all of the atomic propositions are either event variables, events, or time intervals.*

A TRE template is a template for the specifications that we want to mine. We use the term *event variable* to denote a place holder for an event. For example, the TRE template $\langle 0.1\rangle[x, y]$ represents *"0 is always followed by 1 within the time interval [x,y]"*, where 0 and 1 are *event variables* and where $x \leq y$ are fixed doubles used in the *time interval*. We use p to denote the number of event variables present in a TRE template. In the given example p is 2.

DEFINITION 4 (TRE INSTANCE). *Let Π be a TRE template. Then, π is a TRE instance of Π if π has a TRE similar to Π in structure where all the atomic propositions are events.*

DEFINITION 5 (BINDING). *Let Σ be an alphabet of events and let V be a finite set of event variables. Then, a binding is a function $b\colon V \to \Sigma$*

A TRE instance corresponds to a TRE template and has an identical TRE structure. Applying a binding to the event variables in a TRE template creates a TRE instance corresponding to that binding. The *binding* is thus a map used to replace event variables with events from the given alphabet.

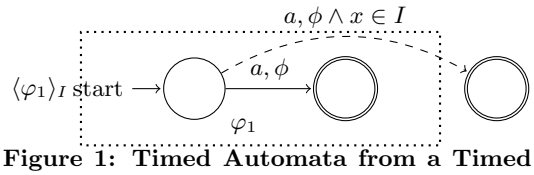

Figure 1: Timed Automata from a Timed Regular Expression

2.2 Timed Automata

Timed automata [3] have been investigated quite rigorously in the recent past. The main motivation for using timed automata is their suitability for modeling time dependent behavior, and the ability to monitor their reachability [21].

DEFINITION 6 (TIMED AUTOMATA [15]). *A timed automaton $\mathcal{A}$ is a tuple $\langle Q, C, \Delta, \Sigma, s, F\rangle$ where Q is a finite set of states, C is a finite set of clocks, Σ is an input (or event) alphabet, Δ is a transition relation, $s \in Q$ an initial state and $F \subset Q$ a set of accepting states. The transition relation consists of tuples of the form $\langle q, \phi, \rho, a, q'\rangle$ where q and q' are states, $a \in \Sigma \cup \{\epsilon\}$ is a letter, $\rho \subseteq C$ and ϕ (the transition guard) is a boolean combination of formulae of the form $(x \in I)$ for some clock x and some integer-bounded interval I.*

A clock valuation is a function $v\colon C \to \mathbb{R}^+$, or equivalently a $|C|$-dimensional vector over $\mathbb{R}^+$. We denote the set of all clock valuations by $\mathcal{H}$. A configuration of the automaton is hence a pair $(q, v) \in Q \times \mathcal{H}$ consisting of a discrete state (sometimes called "location") and a clock valuation. Every subset $\rho \subseteq C$ induces a reset function $Reset_\rho : \mathcal{H} \to \mathcal{H}$ defined for every clock valuation v and every clock variable $x \in C$ as

$$Reset_\rho v(x) = \begin{cases} 0, & \text{if } x \in \rho \\ v(x) & \text{if } x \notin \rho \end{cases} \quad (1)$$

$Reset_\rho$ resets all the clocks in ρ to zero and leaves the other clocks unchanged.

It has been proven that every timed regular language defined by a generalized regular expression can be recognized by a timed automaton [15]. We present a few simple examples of representing TREs using timed automata.

Consider the TRE $\langle\varphi_1\rangle_I$ and its equivalent timed automaton shown in Figure 1. Within the rectangle, we have the acceptor for φ_1 with no timing constraints. For the timed operator, a new clock x is introduced and a test $x \in I$ has been added to guard every transition leading to the final state f.

Let us further examine the basic idea behind construction of a timed automaton for φ^*. If the time interval applies to each φ separately, then the clocks need to be reset at each new iteration of φ. On the other hand, if the time interval applies to the whole φ^* expression, we need the values of all clocks at each new iteration of φ to represent the total time elapsed in the previous iterations. This is achieved by adding a new clock x which is never reset to zero and transitions to the initial state in which all the clocks get the value of x.

2.3 Dominant Properties

The TRE instances generated by the binding contain every permutation of events in the alphabet within the TRE

template. There is thus a total of Σ^p possible TRE instances. Generally, we are interested in mining all of the valid TRE instances. However, these instances contain both interesting and frequently occurring patterns, as well as those that might have been present just a handful of times in the trace. We thus use a ranking component to reduce the mined set to contain only the dominant instances, or specifications.

First, we will only consider properties that are *valid* [22]. We express that a binding and its corresponding TRE instance are interesting if the TRE instance holds on each trace, thus 100% valid. We also use the concepts of support and confidence from [22].

Definition 7 (Support Potential). *The support of a TRE instance π on a trace t is the number of time points of trace t which* could *falsify π.*

Definition 8 (Support). *The support of a TRE instance π on a trace t is the number of time points of trace t which could falsify π, but* do not *falsify π.*

Definition 9 (Confidence). *The confidence of TRE 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 TRE 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 will examine different threshold values for both support and confidence and will evaluate best thresholds for reducing all feasible properties to just the dominant and interesting properties.

3. APPROACH

3.1 Workflow

Figure 2 provides a high level overview of the technique we propose for mining temporal properties from system traces.

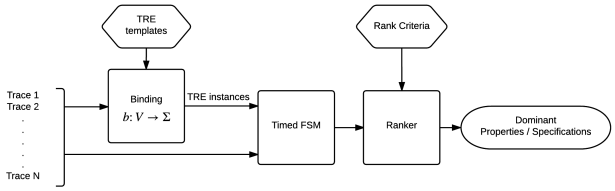

Figure 2: Property Mining Workflow

The binding function accepts a set of N traces and a TRE template. We use execution traces collected during system runtime. The time-event traces are generated using instrumentation already present in the system and may include network traffic logs, operating system logs, program instrumentation logs, etc. The binding function accepts a set of N logs, where $N \geq 1$. From these logs, the function extracts an alphabet Σ of unique events. The TRE template is an abstraction of the desired temporal property, a temporal relationship of interest for the system. The TRE template uses a set V of event variables, where the variables range from 0 to p. The binding function binds the set V to the alphabet of events Σ to generate a set of TRE instances.

The timed FSM evaluates the TRE instances on the same set of N traces. The Σ^p TRE instances are encoded into a p dimensional incidence matrix that is used by the timed FSM to keep track of state, clocks, and evaluation results. As the timed automaton evaluates each TRE instance on the trace, it updates the success and failure values in the matrix. When the automaton is finished evaluating the TREs on all the traces, it passes the results matrix to the ranker.

The ranker uses the results matrix generated by the timed FSM to calculate the confidence and support values for each TRE instance. The rank criteria are the threshold values for confidence and support that are used to select only the dominant TRE instances. The ranker uses these criteria to filter out any of the instances with confidence and support values below the specified thresholds. Thus, we are left with only the dominant instances, which are the dominant properties or specifications for the system being analyzed.

3.2 TRE Mining Algorithm

An intuitive way to evaluate a TRE on a system execution trace is to recursively evaluate the TRE according to its semantics at every line of the trace. A high level description of the steps taken by the proposed algorithm are as follows:

- **Representing a TRE** Parse the input TRE template and transform it into a timed automaton.
- **Representing a trace** Parse the input trace into a linear array representation where each unique event has its corresponding time and event id, *[time, eventid]*.
- **Checking TRE instances over traces** Iterate over the trace and process each time event pair by matching them to the relevant TRE instances. Update the *success* and *reset* for the relevant TRE instances at each trace event.

Algorithm 1 Timed Regular Expression Mining without Negation

Require: Given a trace with Σ unique events and a TRE pattern with p event variables
Ensure: Parse TRE and formulate a finite Timed Automaton $\rightarrow$ A
1: Initialize an incidence matrix of size Σ^p
2: Initialize the *success* and *reset* counters for all permutations in the matrix
3: **for** (each unique event i from the trace) **do**
4: **for** (each permutation with event i) **do**
5: Update the automaton state;
6: Update the success if A $\rightarrow$ FS;
7: Update the reset if A $\rightarrow$ ES;
8: **end for**
9: **end for**
10: Evaluate Support - S
11: Evaluate Confidence - C

Algorithm 1, is a detailed description of the above steps in form of pseudo code. In Algorithm 1, Σ is the events alphabet and p is the number of event variables in the TRE template without negation. A denotes the timed automaton and FS and ES denote the final and error states of the timed automaton respectively. S and C denote the *support* and *confidence* metrics of the TRE instances on the given trace.

In contrast, Algorithm 2 evaluates the TREs with negation and has a loop to iterate over all possible permutations (see the Discussion for an explanation of the differences in run times of the loops).

Algorithm 2 Timed Regular Expression Mining with Negation

Require: Given a trace with Σ unique events and a TRE pattern with p event variables
Ensure: Parse TRE and formulate a finite Timed Automaton $\rightarrow$ A
1: Initialize an incidence matrix of size Σ^p
2: Initialize the *success* and *reset* counters for all permutations in the matrix
3: **for** (each unique event i from the trace) **do**
4: **for** (each possible permutation) **do**
5: Update the automaton state;
6: Update the success if A $\rightarrow$ FS;
7: Update the reset if A $\rightarrow$ ES;
8: **end for**
9: **end for**
10: Evaluate Support - S
11: Evaluate Confidence - C

3.3 Temporal Property Patterns

The proposed technique relies on mining patterns that include timing constraints. Of interest are the most commonly occurring temporal property patterns, as described in [26]. In order to make use of these patterns, we extend them by transforming them into TRE Property Patterns.Let us assume we have several causing events P to share one effect event S. Examples of the TRE property patterns are presented in Table 1. The properties impose timing constraints on events by adding a time interval $[x, y]$, where $x \leq y$ are fixed doubles.

Table 1: TRE Property Patterns

Name	TRE
Response	$[-P]^*; ((P; [-S]^*; S\langle x, y\rangle); [-P]^*)^*$
Alternating	$[-P, S]^*; (P; [-P, S]^*; S; [-P, S]^* \langle x, y\rangle)^*$

4. DISCUSSION

We examine the following important characteristics of the proposed temporal property mining technique: soundness, completeness, optimality (in terms of space and runtime), and scalability. We argue that the technique we proposed and described in this paper is sound and complete.

By sound we mean that a mined specification reported by our algorithms actually satisfies the given thresholds for confidence and support in the provided input traces. 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. The proposed temporal property mining technique requires memory space for the multidimensional results matrix, the timed FSM, and the trace contents. The storage requirements for the matrix are directly influenced by p and Σ. We encode the matrix to hold the evaluation results of every TRE instance generated by binding the events from Σ to the p event variables in the TRE template. There are Σ^p possible TRE instances, thus we need $O(\Sigma^p)$ space to hold the acceptor results.

Table 2: Characterization of the Algorithms

Characteristic	Details
Soundness	Our algorithm continuously keeps track of the success and reset rates for each TRE instance in the results matrix.
Complete	Our technique examines every TRE instance that can be affected in both cases with and without negation.
Run Time Requirement	With Negation - $O((\Sigma-1)^p \cdot L)$, Without Negation - $O(p \cdot {}^{(\Sigma-1)}P_{(p-1)} \cdot L)$
Memory Requirement	$O(\Sigma^p)$
Scalability	$O(L)$ as $L \gg \Sigma$, $L \gg p$ and Σ^p is small for interesting properties

The storage required for the FSM is proportional to the number of its states [18]. If n is the length of the TRE, then an equivalent NFA has $O(n)$ states and an equivalent DFA has at most $O(\Sigma^n)$ states. Similar to the matrix, complex properties that encode a lot of relationships between events result in exponential space growth for the FSM. However, majority of interesting properties will remain simple [10]. Lastly, the storage requirements for the trace are equivalent to the length of the trace, $O(L)$.

The value of Σ, on the other hand, affects the size of each dimension but has a much smaller effect on the growth of the matrix since it only affects the base number in the exponential space complexity. In general, for any TRE template, we are sampling p events from an alphabet of size $\Sigma - 1$ with replacement and ordering. This results in $(\Sigma - 1)^p$ possible combinations to be considered in the loop of Algorithm 2. The automaton will thus evaluate $O((\Sigma-1)^p)$ TRE instances for every time-event pair in the trace. Given that the length of the trace t is L, the worst case execution time per trace will be $O((\Sigma - 1)^p \cdot L)$.

In Algorithm 1 for TRE templates without negation. This case differs in that only the TRE instances where the bound ed event corresponds to that read from the trace needs to be evaluated. The automaton evaluates $O(p \cdot {}^{(\Sigma-1)}P_{(p-1)})$ TRE instances for every time-event pair in the trace. Given that the length of a trace t is L, the worst case execution time per trace will be $O(p \cdot {}^{(\Sigma-1)}P_{(p-1)} \cdot L)$.

Notice that the approximate relative speedup of Algorithm 1 (without negation) as compared to Algorithm 2 (with negation) is $\frac{\Sigma}{p}$. This observation will be assessed through the experimental evaluation in the next section. Finally, as mentioned earlier, since the majority of interesting properties are not complex, Σ^p will remain small in practice.

We do not include the proofs for various characteristics in the paper due to space constraints.

5. PERFORMANCE 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 a combination of R and C++ and is made available as a R package [1]. We use the R package 'Rcpp'[2] for integration of R and C++ and make use of C++ internally for better performance. Ragel[3] is a framework we used to synthesize Timed Automata for TREs.

We developed TRE property pattern variants, such as shown in Table 1, of commonly used patterns of QREs [26]. By using these TRE variants as TRE templates, we mined TRE instances from a real-world system traces: QNX tracelogger. We used different thresholds for *support* and *confidence* to uncover interesting TRE instances. Further, to demonstrate the performance and scalability of our approach, we synthesized traces for different configurations w.r.t. the length of traces, number of variables in a TRE template, and the number of distinct events in traces. For the experiments, we used a single eight core machine equipped with the Intel i7-3820 CPU at 3.60 GHz and 31.4 Gb of RAM. The machine runs Ubuntu 14.04 LTS 64 bit.

5.1 TRE Templates

The following four TRE templates are used for the evaluation of both real and synthetic traces. These are the first four TRE property patterns listed in Table 1 and parametrized with a time interval of 0 to 3,000.

T-1(response): $(\hat{}P)^*.((\langle P.(\hat{}S)^*.S\rangle[0,3000]).(\hat{}P)^*$
T-2(alternating): $(\hat{}P,S)^*.((\langle P.(\hat{}P,S)^*.S\rangle[0,3000]).(\hat{}P,S)^*$

5.2 Evaluation using Real QNX Traces

The QNX real time operating system (RTOS) is used in many safety critical systems, such as medical devices, nuclear monitoring systems, vehicles, etc. The QNX RTOS has a very advanced logging facility, *tracelogger*. Tracelogger facilitates detailed tracing of the kernel and user process activity on any system. More specifically, it can log interrupt activity, various states of processes and threads, communication within the system, kernel calls, custom user events, and much more. The logged events give a detailed view into the behavior of the system, but due to the large quantity of the produced information are often difficult to make use of by developers and system designers. These traces are thus a perfect resource for dynamic mining of system properties and specifications.

For the evaluation we used a set of traces collected from an operational hexacopter loaded with the QNX RTOS and a user control process. Of interest in this portion of the evaluation are the optimal thresholds for support and confidence that would produce a minimal dominant set of specifications that can be assessed by developers or system designers.

Table 3 presents the results of the evaluation. The hexacopter trace used for the evaluation contained 1 million events, with 205 distinct events. We used two TRE templates, T-1 and T-2, for the evaluation. The interval used in the templates is sufficient for most interesting interactions to complete. The tables report the number of specifications mined by our algorithm given varying values of support and confidence.

It is evident from the results in Table 3a that TRE T-1, the response property, occurs infrequently in the trace for a large number of events. Due to an abundance of inter-process communication and system calls within the kernel, this is expected.

The results show that higher threshold values for support and confidence result in less specifications being reported. Let's assume that a developer or engineer would be able to assess around 30 specifications per property, more than that would be abundant. Based on the results in Table 3, the support threshold should be kept between 10,000 to 50,000 and the confidence threshold should be kept at 0.8 or higher.

Table 3: Mining TREs on QNX Traces

(a) Results of Mining TRE T-1

Support \ Confidence	**1**	**0.9**	**0.8**	**0.7**	**0.6**
1 - 1,000	0	194	275	324	363
1,000 - 5,000	0	155	192	211	219
5,000 - 10,000	0	91	95	98	100
10,000 - 50,000	0	39	39	40	40
50,000 - 80,000	0	15	15	16	16
80,000 - 300,000	0	3	3	3	3

(b) Results of Mining TRE T-2

Support \ Confidence	**1**	**0.9**	**0.8**	**0.7**	**0.6**
1 - 1,000	0	30	36	41	45
1,000 - 5,000	0	27	32	36	39
5,000 - 10,000	0	22	25	28	31
10,000 - 50,000	0	19	19	21	22
50,000 - 80,000	0	10	10	11	11
80,000 - 300,000	0	2	2	2	2

Based on the shown results, we should also note that the TREs are satisfied frequently in QNX traces. This is likely a consequence of a large event alphabet. QNX traces have a large versatile mix of events, thus a smaller number of QNX events may appear to satisfy any particular property. However, the support threshold can be used to remove the effects of alphabet size variation, since only the interesting properties will be satisfied frequently in the traces.

5.3 Evaluation using Synthesized Traces

Real system traces are suitable for testing the feasibility and usefulness of our proposed technique. However, they cannot be easily used to demonstrate some important performance and scalability properties of the approach.

We synthesized a set of traces by varying the following three parameters: the distinct number of events (the size of the events alphabet Σ), the total number of events (length L of the traces), and the number of variables in TRE templates (p event variables). We used the TRE property patterns listed in Table 1. Below we will describe each setup for evaluating the performance of our mining algorithm, as well as will present the associated results.

Setup and Results: We performed the experiment under three different setups. In each of the setups, the value of one of the three parameters was varied. The values of others were kept constant. We executed each experiment 10 times.

Setup 1: In this first setup, we varied the total number of events in the traces. The number of distinct events was set to 4, and the number of variables in TRE templates was set to 2. The total number of events in the traces were varied exponentially.

The results are shown in Figure 3. The x axis represents the total number of events in the traces, and the y axis represents the average execution time of our mining algorithm.

[1] https://bitbucket.org/sfischme/tre-mining
[2] http://www.rcpp.org/
[3] http://www.colm.net/open-source/ragel/

Note that the measurement of the execution time includes only the processing of trace events (within the main loop of Algorithm 1). The measurement does not include the initialization of the algorithm, the overhead of which is constant.

As shown in Figure 3, the execution of our mining algorithms grows linearly w.r.t. the total number of events in the trace. For example, the average execution time is 170 milliseconds for processing 0.1 million events.

Setup 2: In this setup, we varied the number of distinct events. There were 10,000 events in total in each trace, and the number of variables in TRE templates was set to 2. As shown in Figure 4, the x axis represents the number of distinct events in the traces, and the y axis represents the average execution time of our mining algorithm. The figure shows that the execution time increases exponentially with the growing number of distinct events. The average execution time for processing a trace of 10,000 events with 25 distinct events is 594 milliseconds.

Setup 3: Lastly, in this setup, we varied the number of event variables in the TRE templates. The total number of events in the traces was set to 10,000, and the number of distinct events was set to 10. For this setup, the four TRE templates used for setup 1 and 2 were varied to include different numbers of event variables. For the sake of brevity, we do not list these TRE templates.

Figure 5 shows the variation in the average execution time w.r.t. the number of variables in the TRE templates. It is evident that the average execution time grows exponentially. The average time for processing a trace of 10,000 events with 10 distinct events and a TRE template containing 4 variables is 4156 milliseconds.

5.4 Comparing Algorithm Performance

Next, we compared the performance of Algorithm 1 and Algorithm 2, where Algorithm 2 considers TRE templates with negation and Algorithm 1 considers TRE templates *without* negation. The TRE templates listed in Table 4 were provided as input to the two algorithms when using synthetically generated traces.

Table 4: Comparing Algorithm 1 and Algorithm 2

TRE Templates for Algorithm 1	TRE Templates for Algorithm 2
$(\langle P.Q.R\rangle[1,5000])$	$(\langle P.\hat{}Q^*.Q.\hat{}R^*.R\rangle[1,5000])$

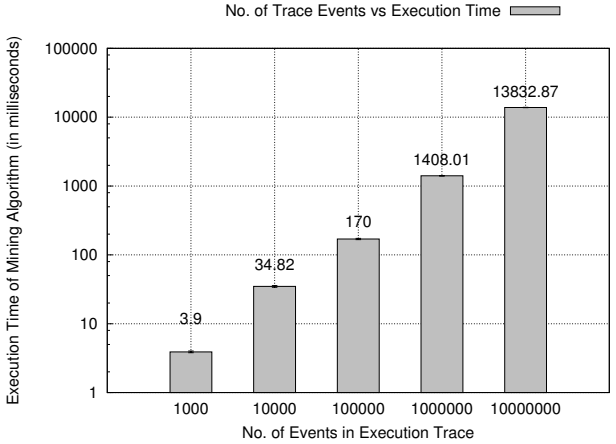

Figure 3: Varying Total Number of Events

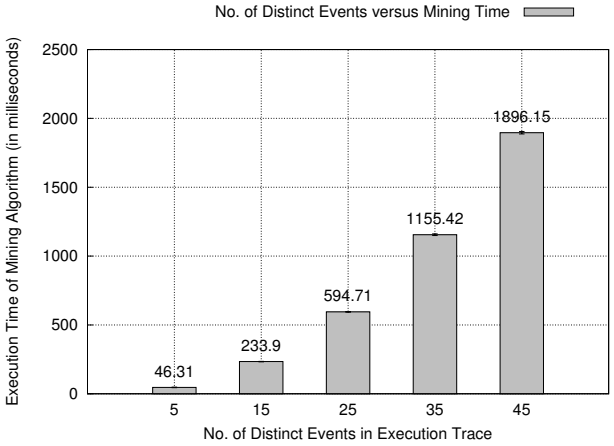

Figure 4: Varying Number of Distinct Events

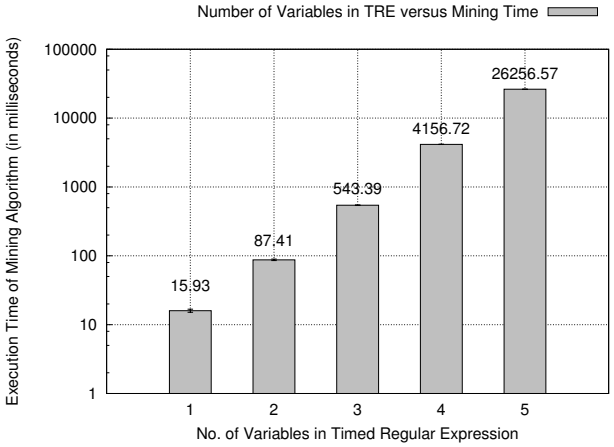

Figure 5: Varying Number of Variables in TRE Templates

For evaluation of real system traces we used the TRE patterns T-1 and T-2.

The trace from QNX used for the evaluation contained 1 million events, with 205 distinct events. It took 3498.37sec and 34.38sec for TREs with and without negation respectively. It is evident from the figure that mining properties with negation takes approximately 100 times longer than mining properties without negation. This corresponds to the expected speedup of $\frac{\Sigma}{p}$ (see the Discussion section), which in this case is $\frac{205}{2}$.

Lastly, the synthesized traces took 111.82sec and 9.50sec for TREs with and without negation respectively. The size of the trace used for this experiment was 50,000 events, with 30 distinct events. It is evident from the figure that mining properties with negation takes approximately 12 times longer than mining properties without negation. This result also closely approaches the expected speedup of $\frac{\Sigma}{p}$, which in this case is $\frac{30}{2}$.

6. CONCLUSION

This paper presented two novel algorithms for mining of TREs with and without negation in embedded system traces. We presented a detailed complexity analysis of both the algorithms. We presented two sets of experiments to demonstrate that the algorithm is scalable, robust, sound and complete. First, we presented experimental results on syntheti-

cally generated traces to analyze the scalability of the algorithms with increase of variables in the TRE template, varying total number of unique events and varying total number of events in the system trace. Secondly, we validate our framework on industrial strength safety-critical real-time applications 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 TRE specification mining. In future, we propose to improve the framework to perform better on TREs with negation and gain considerable speedup as compared to TREs without negation.

7. REFERENCES

[1] *QNX Operating System: system architecture*. QNX Software Systems Ltd., 1997.

[2] R. Alur, P. Černỳ, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. *ACM SIGPLAN Notices*, 40(1):98–109, 2005.

[3] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2):183–235, Apr. 1994.

[4] G. Ammons, R. Bodík, and J. R. Larus. Mining Specifications. In J. Launchbury and J. C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 4–16. ACM, 2002.

[5] M. Bonato, G. Di Guglielmo, M. Fujita, F. Fummi, and G. Pravadelli. Dynamic Property Mining for Embedded Software. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 187–196. ACM, 2012.

[6] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 169–180. ACM, 2006.

[7] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 1st India software engineering conference*, pages 5–14. ACM, 2008.

[8] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller. Automatically generating test cases for specification mining. *Software Engineering, IEEE Transactions on*, 38(2):243–257, 2012.

[9] 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*, pages 85–96. ACM, 2010.

[10] 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*, pages 411–420. IEEE, 1999.

[11] 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*, volume 5-35. ACM, 2001.

[12] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27. Citeseer, 2003.

[13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27(2):99–123, 2001.

[14] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[15] O. M. Eugene Asarin, Paul Caspi. Timed Regular Expressions. *J. ACM*, 49(2):172–206, Mar. 2002.

[16] E. M. Gold. Complexity of automaton identification from given data. *Information and control*, 37(3):302–320, 1978.

[17] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty. IODINE: a tool to automatically infer dynamic invariants for hardware designs. In *Proceedings of the 42nd annual Design Automation Conference*, pages 775–778. ACM, 2005.

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

[19] 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*, volume 8977, page 25. Springer, 2015.

[20] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[21] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell, 1997.

[22] C. Lemieux, D. Park, and I. Beschastnikh. General LTL Specification Mining. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 81–92. IEEE, 2015.

[23] W. Li, A. Forin, and S. A. Seshia. Scalable specification mining for verification and diagnosis. In *Proceedings of the 47th design automation conference*, pages 755–760. ACM, 2010.

[24] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, pages 501–510. ACM, 2008.

[25] J. Yang and D. Evans. Automatically inferring temporal properties for program evolution. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 340–351. IEEE, 2004.

[26] 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*, PASTE '04, pages 23–28, New York, NY, USA, 2004. ACM.

[27] 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*, pages 282–291. ACM, 2006.