Mining timed regular expressions from system traces

Greta Cutulenco, Yogi Joshi, Apurva Narayan, and Sebastian Fischmeister
University of Waterloo, ON, Canada
{gcutulen, y2joshi, apurva.narayan, sfischme}@uwaterloo.ca

Abstract

Dynamic behavior of a program can be assessed through an examination of events emitted by the program during execution. For the behavior to be correct it must satisfy a set of specified temporal properties. 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, temporal properties are automatically extracted from traces of program execution. 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 (FSM) to serve as an acceptor. We evaluate our technique on real-world datasets. We formally demonstrate the scalability and performance of our approach by running our implementations on synthesized traces of different sizes with different parameter values.

1. Introduction

Temporal behavior of programs has been extensively studied [6, 11]. Recently, the idea of mining system traces to derive likely temporal specifications of programs has become popular [12]. Mining techniques generally identify a set of specifications which are satisfied by traces w.r.t. certain criteria. Commonly used patterns of specifications are provided for mining a systems' specifications. Many programs lack formal temporal specifications, and mined specifications are therefore valuable and could be used for a wide variety of activities in software development life cycle (SDLC). These activities include software testing [4], automated program verification [10], anomaly detection [3], debugging [8], etc. Further, mined specifications can assist automated verification techniques because they provide an easy and user-friendly way to describe programs' specifications. As argued by Ammons et al. [2], 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 using templates expressed using regular expressions, LTL and other custom formats.

Most of the existing research in the context of mining temporal specifications focuses on qualitative notion of time, i.e., the specifications describe an *ordering* of events. For example, an LTL specification $\square(\text{request} \rightarrow \Diamond\text{response})$ 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 a *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, which allow mining of specifications that account for the quantitative notion of time. For example, mining specifications of interrupt handlers, which complete their executions within a set of predefined deadlines for a real-time system, involves consideration about the quantitative notion of time. Such specifications are of important for safety-critical real-time systems for which a delayed response to an emitted event may lead to a fault in the system. Certain specifications of real-time systems can be modeled using timed regular expressions, which can be in turn translated into the corresponding timed automata.

With a motivation to address the problem of mining specifications with explicit notion of time, we propose a technique to mine instances of timed regular expression (TRE) [7] templates satisfied by a given system's traces. TREs extend regular expressions by providing additional operators to specify timing constraints between events. By using a set of commonly used 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. [7] to synthesize the timed automaton for a given TRE. Such timed automaton is then used as a checker to verify whether traces satisfy the corresponding TRE. TRE templates replace actual events in a TRE instance by a variable, which can be assigned a value of an actual event in traces. We provide two algorithms for mining instances of TREs from their templates. Both algorithms require a TRE template and a system's traces as input. The algorithms then use the distinct events in the traces to generate different possible *permutations* of the events that can be formed by substituting the actual events into the variables in the TRE template. Intuitively, the given trace is processed against every permutation to check whether the trace satisfies TRE instances denoted by the corresponding permutations. Further, we define *support* and *confidence* as metrics to evaluate the degree to which a TRE instance is satisfied by a trace. The algorithms then report the permutations which satisfy the given threshold values of support and confidence.

The first algorithm is designed for TRE templates containing negation operator. The second algorithm processes TRE templates without a negation operator. We later prove that although both algorithms' execution times are exponential in terms of the number of variables in individual TRE templates, the second algorithm generally runs faster than the first 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. We also prove that 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 which consist of logs produced by QNX realtime RTOS [1] during various runs of application software on different hardware platforms and controller area network (CAN) [**?**]traces of an automobile. We report the performance of our algorithms in terms of the execution time. We demonstrate the scalability and performance of our approach by running our 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 traces and the complexity of TRE templates.

In this paper we propose a framework to automatically mine properties that are in the form of timed regular expressions from system traces. Using an abstract structure of the property the framework constructs a finite state machine (FSM) to serve as an acceptor. We encode the concrete properties in a set of matrices. The FSM is used to evaluate the satisfaction of every potential concrete property in the trace. Using a ranking system we infer the strongest concrete properties that describe the system and thus deduce a set of interesting system specifications.

The key contributions of this paper involve developing an efficient framework for extracting complex temporal properties that take on the form of timed regular expressions and evaluating the framework experimentally on industrial real-time systems:

- We develop two novel algorithms to mine instances of timed regular expressions satisfied by a given system's traces. To our knowledge, this is the first technique for mining specifications with explicit notion of timing constraints (Section 3),
- Identifying computational approaches that optimize the extraction of the specified properties (Section 3),
- Demonstrating the frameworks' applicability to large traces collected from real industrial systems (Section 4).

2. Background

2.1 Timed Regular Expressions

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 the 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 application domains require more detailed models which include accurate timing information. For example, we might want to modify a formal specification from "a is followed by b" into a more precise specification with timing information as " a is followed by b within x seconds". We need to develop algorithms for mining specification with timing information. Timed automata [**?**] (automata equipped with clocks) have been investigated quite rigourously in the recent past. The main motivation of using timed automata is due to their suitability for modeling time-dependent behavior, and their ability to monitor their reachability [**?**].

We will formally describe our nomenclature.

DEFINITION 1 (Trace and event). *The alphabets of events is a finite alphabet of strings. An ordered sequence of events is a trace.*

We use the term *event* to denote a place holder for an event. We use Time Regular Expressions (TRE) [7] to provide specification templates.

DEFINITION 2 (Timed Regular Expression (TRE)[7]). *Timed regular expressions 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 $\langle\phi\rangle_I$, $\phi_1\dot{\phi}_2$, $\phi_1 \vee \phi_2$ and ϕ^* are Σ expressions.*

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.*

For example, the TRE template $< 0.1 > [x, y]$ represents the "0 is always followed by 1 within the time interval [x,y]", where $x \leq y$ and are fixed doubles.

A TRE instance corresponds to a TRE template and has an identical TRE structure. With each event variable in the TRE template when replaced by an event, we refer to the map as $binding$.

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 and 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 $b\colon V \to \Sigma$*

Applying a binding to the variables in a TRE template creates a TRE instance corresponding to that binding.

As we propose a framework that mines properties in the form of TREs from system traces using an abstract structure of the property using a finite state machine (FSM) to serve as an acceptor. We use the formalism of Timed automata for our purpose as it allows for enforcing explicit timing constraints in the model.

2.2 Timed Automata

Timed automata is a theory for modeling and verification of real time systems. Explicit timing constraint are naturally present in real-life systems. Classical models (finite automata, Petri-nets etc.) cannot express such real time constraints. Time automata [**? ?**] has been used extensively for modeling real-time systems. One of the most important properties of the timed automata is probably that the reachability properties are decidable [**?**] even though the timed automata have infinite number of configurations. The main idea behind this result is the construction of region-automaton, which finitely abstracts the behavior of timed automata in a way that checking reachability in a timed automata reduces to checking reachability in a finite automaton.

DEFINITION 6 (Timed Automata [7]). *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 $(q, v) \in Q \times H$ consisting of a discrete state (sometimes called location) and a clock valuation. Every subset $\rho \subseteq C$ induces a reset function $Reset_\rho : H \to H$ defined for every clock valuation v and every clock variable $x \in C$ as

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

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

It has been proved [7] that every timed regular language can be recognized by a timed automaton. We present a simple example of representing the TRE using timed automata and vice-versa. The basic idea behind construction of a timed automata φ^* is that we need the values of all the 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

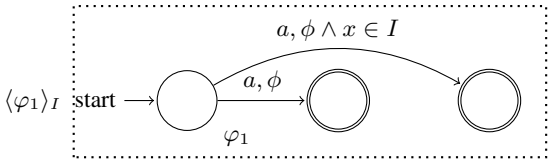

Figure 1: Timed Automata from a Timed Regular Expression

transitions to the initial state in which all the clocks get the value of x. In Figure 1 we see that a new clock x and a test $x \in I$ has been added to guard every transition leading to final state.

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 [[Φ]] to be in the interval I.

The '.' is the concatenation operator and the '+' is the operator for one or more instances of the expression.

2.3 Dominant Properties

We express that a binding and its corresponding TRE instance are valid on a trace if the TRE instance holds on each trace in the log. Generally, we are interested in mining all of the valid TRE instances.

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

DEFINITION 8 (Trace Confidence (To be Verified)). *Trace Confidence of a TRE π on a trace t is the number of time points of t which could falsify π but do not falsify π.*

The concrete properties examined by the analyzer contain every permutation of trace events within the template. These contain both interesting and frequently occurring patterns, as well as those that might have been found just a handful of times in the trace. Since the number of concrete properties could be very large, S^L, a ranking component is used to reduce the set to interesting properties only.

The effectiveness of selecting a meaningful subset of properties depends on picking a good set of criteria and associated satisfaction thresholds. The ranking criterion we use is a combination of support and probability. Support is the percentage of all traces that contain the concrete property. Probability is the ration of successful evaluations of the property over total possible occurrences. We specify a threshold for these measures to extract properties that are consistently dominant in the collected traces. In section X our experiments explore the appropriate values for the thresholds.

Another motivation for using the ranker is the presence of imperfect traces. Traces can be imperfect as a result of dropped events or execution of faulty programs. In such cases, properties may not be perfectly satisfied in the collected traces. By using the probability and support ranking criteria we focus on finding the dominant properties in the trace.

3. Approach

We use traces of execution collected during the runtime of a system. The event traces are generated using instrumentation already present in the system and include network traffic logs, operating system logs, or program instrumentation logs. The analyzer accepts a set of N logs, where N $\geq$ 1.

The templates in 2 refer to templates of temporal properties - timed regular expressions [7] that encode constraints on the relationships between subsets of trace events. Each template is an

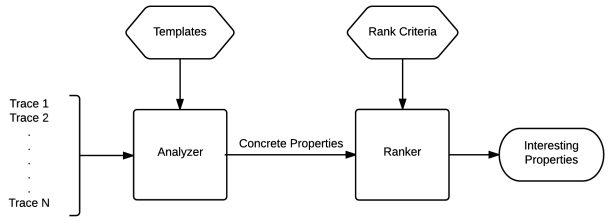

Figure 2: Property Mining Workflow

abstraction of the desired temporal property, like request, alternating, etc. [5]. A template uses an abstract set of symbols ranging from 0 to L, where L is the length of the abstract alphabet. It also uses operators and time intervals as defined in [7]. For instance, (1.<0>[2,5])+ is a template for the response pattern. The '.' is the concatenation operator and the '+' is the operator for one or more instances of the expression. The template specifies that some log event 1 is followed by another log event 0 within 2 to 5 time units, and this pattern occurs at least once in the execution trace.

The abstract alphabet is used as a placeholder. The analyzer extracts a set of S unique event symbols from the collected traces and substitutes these in place of the abstract symbols to form concrete temporal properties. In the case of (1.<0>[2,5])+ the analyzer will evaluate the property for all combinations of two events in S. The key insight for achieving O(nL) time complexity is that an event X from the trace can be treated as either a P event or an S event.

Using an incidence matrix of dimension L we will evaluate which event combinations satisfy the given property. The acceptor FSM will iterate over the events in the system log and at each new event evaluate if any of the combinations in the matrix are satisfied. The matrix keeps track of the number of successes and failures to evaluate each concrete temporal property.

An intuitive way to evaluate a time regular expression on a system execution trace is to over the trace and recursively evaluate the TRE according to its semantics. A high level description of our algorithm is presented in

- **Representing a TRE** We parse the input timed regular expression and transform it into a timed automata.
- **Representing a trace** We parse the input trace into a linear array representation where each unique event has its corresponding time and event id [time, eventid].
- **Checking property instances over traces** We iterate over the set of each unique event and process the entire trace with respect to each unique event. Updating the *success* and *reset* for the entire trace. Processing the trace refers to matching the required TRE in the process trace.

In Algorithm 1 Σ is the set of alphabets and p is the number of place holders in the TRE. A denotes the timed automaton and FS and ES denote the final and error states of the timed automaton. S and C denote the *support* and *confidence* metrics of the TRE on the given trace.

We used the most commonly occurring temporal property patterns [**?**] by transforming them into Time Regular Temporal Property patterns. Let us assume we have several causing events P to share one effect event S. The TRE property patterns are presented in Table 1.

Algorithm 1 Timed Regular Expression Mining

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

Name	TRE
Response	$[-P]^*;((P;[-S]^*;S\langle x,y\rangle);[-P]^*)^*$
Alternating	$[-P,S]^*;(P;[-P,S]^*;S;[-P,S]^*\langle x,y\rangle)^*$
MultiEffect	$[-P,S]^*;(P;[-P,S]^*;S;[-P]^*\langle x,y\rangle)^*$
MutiCause	$[-P,S]^*;(P;[-S]^*;S;[-P,S]^*\langle x,y\rangle)^*$
EffectFirst	$[-P]^*;(P;[-P,S]^*;S;[-P,S]^*\langle x,y\rangle)^*$
CauseFirst	$[-P,S]^*;(P;[-S]^*;S;\langle x,y\rangle[-P]^*)^*$
OneCause	$[-P]^*;(P;[-P,S]^*;S;\langle x,y\rangle[-P]^*)^*$
OneEffect	$[-P]^*;(P;[-S]^*;S;[-P,S]^*\langle x,y\rangle)^*$

Table 1: TRE Property Patterns

4. Discussion

Our work mines temporal properties that can be expressed using timed regular expressions. The following discussion explores how the chosen approach accomplishes this and why it does so optimally.

4.1 Using Timed Automata as Acceptors

First we can look at regular expressions and their acceptors without the timing constraints. Regular expressions offer a declarative way to express the patterns that we want to accept. Every language defined by a regular expression is also defined by a finite automaton, as defined by Theorem 3.7 in [9]. 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 Deterministic Finite Automaton (DFA) for any regular expression.

Similarly, timed automata are recognizers of timed languages. They have states, as any FSM, as well as clocks. Every timed language defined by a (generalized extended) regular expression is accepted by a timed automaton, as defined by Theorem 2 in [7]. Thus, timed automata and generalized timed regular expressions have the same expressive power.

The language of a timed automaton consists of all the strings that it accepts [7]. The strings in our case are sequences of time-event pairs. The timed automata we generate are acceptors for strings that satisfy the desired property. The more strings in the trace that can be accepted, the more dominant is the property.

4.2 Memory Requirements

Our property mining technique requires space for storing the multidimensional results matrix, the FSM, and the trace itself.

The storage requirements for the matrix are directly influenced by p and Σ. We want to encode the matrix to hold the evaluation results of every combination of unique events that fit in the TRE. Thus we need $O(\Sigma^p)$ space to hold the acceptor results.

The dimensionality of the matrix grows proportionally to p, the number of symbols in the desired property. Complex properties that encode a relationship between a large number of events will result in higehr values of p. Increasing values of p result in exponential space growth. However, the majority of properties of interest represent relationships among a few events only [5, 6]. Thus, in general, the dimension p of the matrix will remain small. According to the properties listed in [6**?**], the typical property will be constrained to 6 events.

The value of Σ, on the other hand, affects the size of each dimension. 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 matrix since it only affects the base number in the exponential space complexity.

Next, we consider the storage requirements for the FSM used to evaluate the TRE. The storage required for the FSM is proportional to the number of its states [9]. 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 [6].

Lastly, the storage requirements for the trace are equivalent to the length of the trace, $O(t)$. Our mining approach examines one time event pair from the trace at a time, without needing to store any past or future events for context. It also does not perform any changes on the contents of the trace. Thus the trace is stored only once and never replicated.

4.2.1 CPU Requirements

TBD

Length of a single trace T - the number of lines/entries of time-event combinations. The FSM will iterate over the trace once. At each trace line it will check the satisfiability for each entry in the matrix (All or just for the event it just read from the trace?). The overall runtime will this be $T * S^L$. This is a huge benefit as the length of traces can be huge, much larger than the complexity of the property. The processing complexity for each character in the input is O(1) in a DFA. The key insight for achieving $O(ST)$ time complexity is that an event X from the trace can be treated as either a 0 event or a 1 event. (that was true in the peracotta paper for a property with 2 symbols only)

4.2.2 Optimality

TBD

Why finite state machines are fastest for checking satisfiability of property in a trace?

Since the properties are contained in a regular language, we can efficiently analyze them. The efficient analysis applies to any property that falls into the regular language. Each one can be represented with a matrix.

Why is using the matrix ensure fastest runtime?

4.2.3 Scaling the Approach

TBD

How does the algorithm scale with respect to the property that we mine?

The scalability of dynamic property mining techniques is generally limited. The techniques scale poorly with the size of the input trace.

The number of symbols present in the property directly influences the matrix size. Thus the more complex the property, the more memory space will be required to store the results.

The automaton for the property is generated once at the start. The automaton is then reused for every cell in the matrix.

5. Conclusion

TBD

References

[1] QNX Neutrino RTOS. http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html.

[2] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining Specifications. In John Launchbury and John 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. ISBN 1-58113-450-9. doi: 10.1145/503272.503275. URL http://doi.acm.org/10.1145/503272.503275.

[3] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 1st India software engineering conference*, pages 5–14. ACM, 2008.

[4] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 85–96. ACM, 2010.

[5] Jinlin Yang David Evans. Dynamically Inferring Temporal Properties.

[6] Matthew B Dwyer, George S Avrunin, and James 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.

[7] Oded Maler Eugene Asarin, Paul Caspi. Timed Regular Expressions.

[8] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 15–24. ACM, 2010.

[9] Rajeev Motwani John E. Hopcroft and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation.

[10] Zachary Kincaid and Andreas 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.

[11] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[12] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General ltl specification mining (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 81–92. IEEE, 2015.