Mining Timed Regular Specifications from System Traces

APURVA NARAYAN, GRETA CUTULENCO, YOGI JOSHI,
and SEBASTIAN FISCHMEISTER, University of Waterloo

Temporal properties define the order of occurrence and timing constraints on event occurrence. Such specifications are important for safety-critical real-time systems. We propose a framework for automatically mining temporal 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. We analytically derive speedup for the fragment and confirm the speedup using empirical validation with synthetic traces. The framework is evaluated on industrial-strength safety-critical real-time applications using traces with more than 1 million entries.

CCS Concepts: • **Software and its engineering** → *Software maintenance tools*; *Dynamic analysis*; *Formal software verification*;

Additional Key Words and Phrases: Timed regular expressions, timed automata, specification mining, real-time systems

ACM Reference format:
Apurva Narayan, Greta Cutulenco, Yogi Joshi, and Sebastian Fischmeister. 2018. Mining Timed Regular Specifications from System Traces. *ACM Trans. Embed. Comput. Syst.* 17, 2, Article 46 (January 2018), 21 pages.
https://doi.org/10.1145/3147660

1 INTRODUCTION

Temporal behavior of programs is an extensively studied topic (Lamport 1978; Dwyer et al. 1999). Recently, the idea of mining likely temporal properties of programs from system traces has become popular (Lemieux et al. 2015). 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 (Dallmeier et al. 2010), automated program verification (Kincaid and Podelski 2015), anomaly detection (Christodorescu et al. 2008), debugging (Gabel and Su 2010), and so forth. Further, mined specifications can assist automated verification techniques because they provide an easy and user-friendly way to describe a program's specifications. As argued by Ammons et al. (2002), 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.

Authors' addresses: A. Narayan, G. Cutulenco, Y. Joshi, and S. Fischmeister, Department of Electrical and Computer Engineering, University of Waterloo, 200 University Ave W, Waterloo, ON N2L 3G1 Canada; emails: {a22naray, gcutulen, y2joshi, sfischme}@uwaterloo.ca.

https://doi.org/10.1145/3147660

Different techniques have been developed for mining specifications from templates expressed using regular expressions, LTL (Bonato et al. 2012), and other custom formats.

There have been both static and dynamic approaches proposed for property mining. Static property mining (mining properties by exploring design models or implementations) is found to be effective and accurate but faces challenges in scaling with the program size (Alur et al. 2005; Ernst 2003). On the other hand, dynamic property mining (infer properties from design execution traces) approaches guarantee scalability but lack in the quality of mined properties that depend on various factors such as the observed executions and the test suite used for stimulating the design (Dallmeier et al. 2012).

There are two main subsets of dynamic property mining approaches: invariant miners (Ernst et al. 2001, 2007; Boshernitsan et al. 2006; Hangal and Lam 2002; Hangal et al. 2005) and temporal property miners (Ammons et al. 2002; Lorenzoli et al. 2008; Engler et al. 2001; Yang et al. 2006; Yang and Evans 2004a).

Of all the available tools, Daikon (Ernst et al. 2007) has proven to be most successful in inferring the most probable invariants. It has been used widely for debugging, testing, documentation, and maintainability (Ernst et al. 2007). It allows one to specify variables and program points to be monitored. In this case, program points are generally all the functions contained in the program. The variables under monitoring will be checked at every entry and exit of a program point. The trace files generated by the programs are processed by an inference system that is monitoring continuously for invariants under consideration (Perkins and Ernst 2004).

There are numerous other tools available for mining invariants. Their usage is restricted to specific applications. There have been extensions to Daikon (Ernst et al. 2007) for developing specific packages such as Agitator (Boshernitsan et al. 2006). The other common packages are DIDUCE (Hangal and Lam 2002) and IODINE (Hangal et al. 2005). DIDUCE is used for identifying root causes in Java programs, whereas IODINE is used for inferring invariants from hardware design descriptions. Various other tools can be found (Hastings and Joyce 1992; Savage et al. 1997) that use dynamic assertion mining and are limited in bug detection.

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 Ammons et al. (2002), they learn a state machine that 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 Lorenzoli et al. (2008), 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 that are useful for analysis and system verification. They generate the Extended Finite State Machines by combining classical algorithms for generating finite state machines and Daikon. These approaches suffer from one main drawback. Mining of a single state machine from system traces is an NP-hard problem (Gold 1978).

In another work (Dwyer et al. 1999), the authors defined a set of temporal property patterns based on case studies of hundreds of real property specifications. 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 (Yang and Evans 2004a, 2004b; Yang et al. 2006). The main motive behind their approach was to achieve scalability to large system traces, in mining binary specifications, in the form of automata, and postprocessing them using inference rules to form complex state machines.

Another interesting attempt (Li et al. 2010) 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 the same as Yang and Evans (2004b), but it tries to mine all the satisfied patterns from the trace file, while in Yang and Evans (2004b), they mine only the strictest pattern satisfied by the state machine.

Most of the existing research in the context of mining temporal specifications focuses on the qualitative notion of time; that is, the specifications describe an *ordering* of events. For example, an LTL specification $\Box(\text{request} \rightarrow \Diamond\text{response})$ 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 the *quantitative* notion of time; that is, 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 (Asarin et al. 2002) 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. (2002) 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. This article is a full version of our previous work (Cutulenco et al. 2016) and includes detailed theoretical proofs and real-world examples to illustrate the technique.

We provide two algorithms for mining instances of TREs from their templates and also detailed worst-case complexity analysis of the two algorithms. 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 use *confidence* and *support* as metrics (Lemieux et al. 2015) to evaluate the degree to which a TRE instance is satisfied by the traces. The algorithms report only the instances that 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. We later prove that although both algorithms' execution times 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 on 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; that is, 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; that is, our algorithms report all TRE instances that 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 (Krten 1999) during various runs of application software on different hardware platforms. We also evaluate the algorithms on the controller area network (CAN) (Davis et al. 2007) traces collected from an automobile during different driving phases. 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 article include:

- An efficient technique for extracting temporal properties with time constraints in the form of TREs from system traces.
- 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.
- Detailed proofs for the space and time complexity of the algorithms.
- 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 (Hopcroft et al. 2006). There is a way to convert any regular expression into a nondeterministic automaton, and further to convert from a nondeterministic to a deterministic automaton. We can thus generate a classical Deterministic Finite Automaton (DFA) for any property expressed as a regular expression.

Generally, the *quantitative* value of time, i.e. the “real time”, interval between two events in a sequence of events, is not considered in classical automata theory. Classical automata theory is able to describe the *qualitative* time, i.e. the ordering of events in a sequence of events. The qualitative notion of time has been found useful for analysis of numerous systems. But, for many other real-time and safety critical systems, more detailed models are required that include accurate timing information. For instance, a formal specification of the form “*event a is followed by event b*” is modified to a more quantified and precise specification with real-values of time associated with it as “*event a is followed by event b within t time units*”. Our work focuses on real-time safety-critical systems, therefore, we need to develop techniques for mining specification that include the relevant timing information. We use the formalism of TREs for our purpose as it allows for defining explicit timing constraints in the model. Formal nomenclature is stated below.

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 timestamps. 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 (Asarin et al. 2002)). Timed regular expressions over an alphabet Σ (also referred to as Σ expressions) are defined using the following families of rules:

- 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 $\langle\varphi\rangle_I$, $\varphi_1.\varphi_2$, $\varphi_1|\varphi_2$, φ^*, and $\hat{}\varphi$ are Σ-expressions.

The novel features are very well explained in (Asarin et al. 2002) as described next. Here the difference 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 $\langle\phi\rangle_I$ operator, which restricts the metric length of the time-event sequences in $[\phi]$ to be in the interval I. Here represents the concatenation operator, ˆ is the negation operator, | is the disjunction or OR operator, and * denotes the Kleene-*. It is important to note that we use TREs, as defined above (Asarin et al. 2002), to provide specification templates.

Definition 3 (Event Variables). An event variable is an atomic proposition in a TRE that can take any event value from the trace alphabet Σ.

Definition 4 (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 $< 0.1 > [x, y]$ 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. Any expression within < and > has to be followed by a time interval that is specified within [and]. The values x and y are separated by ,(comma) the definitions below have been taken and/or modified from (Lemieux et al. 2015).

Definition 5 (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 6 (Binding (Lemieux et al. 2015)). Let Σ be an alphabet of events and let V be a finite set of event variables. Then, a binding is a function $b : V \rightarrow \Sigma$.

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. We mine all occurrences of TRE instances generated by the binding function on the alphabet of the input trace.

2.2 Timed Automata

The incentive to use timed automata is their ability to model time dependent behavior and monitor their reachability (Larsen et al. 1997). Timed automata are furnished with clocks, which makes them optimal for modeling and verification of real-time systems' behavior (Alur and Dill 1994). Other models such as finite automata, Petri-nets, and others are in appropriate given their inability to express explicit timing constraints. Another attractive feature of timed automaton is their reachability property where checking reachability in a timed automaton reduces to checking the reachability of a finite automaton.

Definition 7 (Timed Automata (Asarin et al. 2002)). 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, $s \in Q$ is an initial state, and $F \subset Q$ is 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, and $\rho \subseteq C$ and ϕ (the transition guard) are 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 $v : C \rightarrow \mathbb{R}^+$, 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.

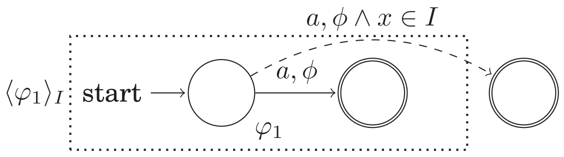

Fig. 1. Timed automata from a timed regular expression (Asarin et al. 2002).

Every subset $\rho \subseteq C$ induces a reset function $Reset_\rho : \mathcal{H} \rightarrow \mathcal{H}$ defined for every clock valuation v and every clock variable $x \in C$ as

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

$Reset_\rho$ 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 (Asarin et al. 2002). 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. Taken from (Asarin et al. 2002). 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. To avoid confusion between continuous time and discrete time TRE semantics, it is stated that the language of a TRE is a set of a sequence of timestamped events.

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 few times in the trace. We thus use a ranking component to reduce the mined set to contain only the dominant instances, or specifications. We consider properties that are both frequently occurring and interesting as dominant properties.

First, we only consider properties that are *valid* (Lemieux et al. 2015). We express that a binding and its corresponding TRE instance are interesting if the TRE instance holds on each trace, and is thus 100% valid. We also use the concepts of support and confidence from Lemieux et al. (2015).

Definition 8 (Support Potential). The support of a TRE instance π on a trace t is the number of time points of trace t that falsify π.

Definition 9 (Support). The support of a TRE instance π on a trace t is the number of time points of trace t that *do not* falsify π.

Definition 10 (Confidence). The confidence of a TRE instance π on a trace t is the ratio of trace support to trace support potential.

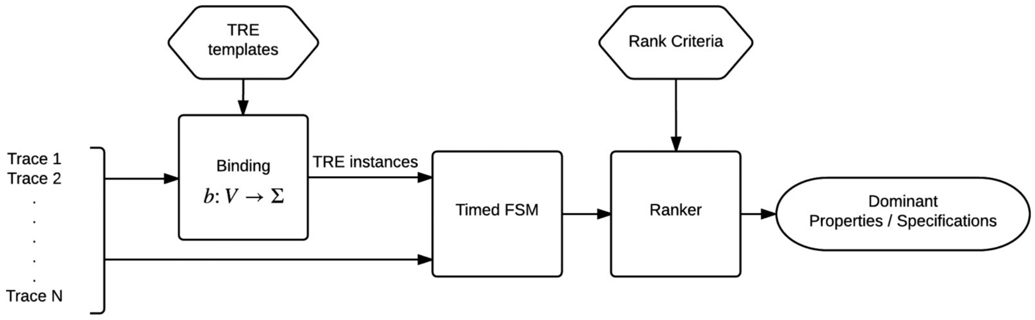

Fig. 2. Property mining workflow (Cutulenco et al. 2016).

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 (Cutulenco et al. 2016) provides a high-level overview of the technique we propose for mining temporal properties from system traces.

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, and so forth. 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 in the 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.

Let us demonstrate the above workflow through an example. Consider the TRE template (1.<0>[2,5])+, which is a simplified 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 property contains two event variables 0 and 1, meaning that the value of p is 2. The binding function will bind the events in Σ to the template and generate an adjacency matrix for the TRE instances.

The timed FSM iterates over the time-event pairs in a trace and at each new event evaluates the relevant TRE instances. Let's assume the matrix contains an entry where 0 is bound to an event "send" and 1 is bound to an event "receive." The FSM reads in the event "send" in the trace at time 0. According to the property, if the next event in the trace is *not* "receive," the FSM will enter an error state and will increase the failure count for this TRE instance to 1 in the matrix. Similarly, if the next event *is* "receive" but the timestamp is 6, the failure count would increase. If the next event is "receive" and the timestamp is 3, which is within the 2-to-5 interval, then the automaton will enter a final state and will increase the success count for the TRE instance.

Once the entire trace has been processed, the ranker will iterate over the matrix to calculate the confidence and support values for each TRE instance. The ranker will report only the properties that meet the defined thresholds for these metrics.

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 is 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.

Below, in Algorithm 1, is a detailed description of the above steps in the form of pseudo code.

ALGORITHM 1: Timed Regular Expression Mining without Negation

```
Require: Given a trace with $\Sigma$ 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 $\Sigma^p$
 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
```

Table 1. TRE Property Patterns

Name	TRE
Response	$(\hat{}(P)^*.(\langle P.\hat{}(S)^*.S\rangle[x,y]).\hat{}(P)^*)+$
Alternating	$(\hat{}(P \mid S)^*.(\langle P.\hat{}(P \mid S)^*.S.\hat{}(P \mid S)^*\rangle[x,y]))+$
MultiEffect	$(\hat{}(P \mid S)^*.(\langle P.\hat{}(P \mid S)^*.S.\hat{}(P)^*\rangle[x,y]))+$
MultiCause	$(\hat{}(P \mid S)^*.(\langle P.\hat{}(S)^*.S.\hat{}(P \mid S)^*\rangle[x,y]))+$
EffectFirst	$(\hat{}(P)^*.(\langle P.\hat{}(P \mid S)^*.S.\hat{}(P \mid S)^*\rangle[x,y]))+$
CauseFirst	$(\hat{}(P \mid S)^*.(\langle P.\hat{}(S)^*.S\rangle[x,y].\hat{}(P)^*))+$
OneCause	$(\hat{}(P)^*.(\langle P.\hat{}(P \mid S)^*.S\rangle[x,y].\hat{}(P)^*))+$
OneEffect	$(\hat{}(P)^*.(\langle P.\hat{}(S)^*.S.\hat{}(P \mid S)^*\rangle[x,y]))+$

ALGORITHM 2: Timed Regular Expression Mining with Negation

```
Require: Given a trace with $\Sigma$ 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 $\Sigma^p$
 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
```

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 Discussion for an explanation of the differences in runtimes of the loops).

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 Yang and Evans (2004b). 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. 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.

4 DISCUSSION

We will examine the following important characteristics of the proposed temporal property mining technique: soundness, completeness, optimality (in terms of space and runtime), and scalability.

4.1 Sound and Complete

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 actually 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 TRE instance in the results matrix. The rates are direct results of execution of the acceptor automaton for the TRE on a trace and are used for calculation of the confidence value for the TRE instance. Given that the reported rates and the derived confidence value are valid, it follows that the derived support value is valid as 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 a time-event pair from the trace, our technique examines every TRE instance that can be affected. If the property contains negation, all the TRE instances are examined at every line of the trace. If the property does not contain negation, all TRE instances that can be affected by the trace event are examined. Thus, at every line of the trace, all the relevant TRE instances are evaluated by the timed automaton. Since all the evaluation results are contained within the matrix, the results for every TRE will be considered when evaluating against the confidence and support thresholds. Therefore, the algorithm will report *all* the TRE instances, the mined specifications, that comply with the confidence and support constraints.

4.2 Memory Requirements

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 want to 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, and 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 higher values of p. Increasing values of p in turn will result in exponential space growth. However, the majority of properties of interest represent relationships among a few events only (Yang and Evans 2004b; Dwyer et al. 1999). Therefore, in general, the dimension p of the matrix will remain small. According to the properties listed in Dwyer et al. (1999), the typical property will be constrained to six 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 timed automaton used to evaluate the TRE. The storage required for the FSM is proportional to the number of its states (Hopcroft et al. 2006). 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, the majority of interesting properties will remain simple (Dwyer et al. 1999).

Lastly, the storage requirements for the trace are equivalent to the length of the trace, $O(L)$. Our mining approach examines one time-event pair from the trace at a time, without needing to store

Table 2. Execution Time with Negation

p	TRE Template	Permutations
1	$\langle \hat{}0\rangle$	$(\Sigma - 1)$
2	$\langle \hat{}0.\hat{}1\rangle$	$(\Sigma - 1)(\Sigma - 1) = (\Sigma - 1)^2$
3	$\langle \hat{}0.\hat{}1.\hat{}2\rangle$	$(\Sigma - 1)(\Sigma - 1)(\Sigma - 1) = (\Sigma - 1)^3$

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.

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 p and the length of the expressions n. However, as we already mentioned, in practice this will not be a concern since properties remain relatively simple.

4.2.1 Runtime Requirements. We assess the performance of our temporal property mining technique in terms of the time required to mine a TRE template on a collected system trace t of length L. The execution time of the proposed technique differs depending on whether negation is included within the expression. We will consider the execution time of the main loop in Algorithm 1 (without negation) and Algorithm 2 (with negation).

TRE Templates *with* Negation. Let's first consider the runtime of the loop in Algorithm 2 for TRE templates with negation. When an event variable in the TRE template is negated, the automaton must evaluate nearly all the TRE instances in the matrix. This is because for any negated event variable, we must consider the entire event alphabet minus the event bound to the variable. The satisfiability of any of these TRE instances may be affected. For the analysis, we look at the total number of distinct event sequences to be evaluated.

Consider TRE template *not 0.* 0 can be bound to "b," forming TRE instance *not "b."* An event "a" in the trace will affect the satisfiability of this TRE instance, and thus its automaton is updated. Similarly, satisfiability will be affected for TRE instances that result from bindings with all other event symbols other than "a." On the other hand, if 0 is bound to "a," forming TRE instance *not "a,"* and we see an event "a" in the trace, we do not need to update the automaton for the instance. Thus, for $p = 1$, we need to evaluate the automaton for $(\Sigma - 1)$ permutations.

For the worst-case analysis, we consider TRE templates where all event variables are negated. At any negated event variable in the template, we will similarly consider $(\Sigma - 1)$ possibilities. Table 2 contains the analysis for the first three values of p. The negation in a TRE template is denoted by ˆ, whereas in the derivation alongside we use $\neg$ to denote negation to avoid confusion.

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 $O((\Sigma - 1)^p \cdot L)$.

TRE Templates *without* Negation. Next we examine the runtime of the main loop in Algorithm 1 for TRE templates without negation. This case differs in that only the TRE instances where the bounded event corresponds to that read from the trace needs to be evaluated. We can thus reduce the previous runtime by evaluating only a subset of all TRE instances that contain the trace event. Providing a bound on the worst-case execution time relies on an analysis of the subset of TRE instances that contain the observed event.

Table 3. Execution Time with Negation

p	**TRE Template**	**Permutations**
1	$\langle\, 0 \,\rangle$	$\underline{\text{a}} = 1$
2	$\langle\, 0.1 \,\rangle$	$\underline{\text{a}}\ \underline{\neg\ \text{a}} + \underline{\neg\ \text{a}}\ \underline{\text{a}}$ $= (1)(\Sigma - 1) + (\Sigma - 1)(1) = 2(\Sigma - 1)$
3	$\langle\, 0.1.2 \,\rangle$	$\underline{\text{a}}\ \underline{\neg\ \text{a}}\ \underline{\neg\ \text{a}} + \underline{\neg\ \text{a}}\ \underline{\text{a}}\ \underline{\neg\ \text{a}} + \underline{\neg\ \text{a}}\ \underline{\neg\ \text{a}}\ \underline{\text{a}}$ $= (1)(\Sigma - 1)(\Sigma - 2)+$ $(\Sigma - 1)(1)(\Sigma - 2) + (\Sigma - 1)(\Sigma - 2)(1)$ $= 3(\Sigma - 1)(\Sigma - 2)$

For the worst-case analysis, we consider permutations of p events without repetition and with ordering. One of the event variables will always be bound to the observed trace event. In Table 3, we assume that event "a" is observed in the trace and $\underline{\text{a}}$ means that the corresponding event variable is bound to event "a." By $\underline{\neg\ \text{a}}$, we mean that an event symbol other than "a" is bound to the event variable. If $\underline{\neg\ \text{a}}$ is used more than once in an event sequence, the chosen events have to be distinct since the permutations are formulated without replacement. Thus, with each such chosen event, the total number of available events decreases by 1. Table 3 contains the analysis for the first three values of p.

In general, for an arbitrary value of p, the runtime is $p \cdot (\Sigma - 1)(\Sigma - 2) \ldots (\Sigma - p + 1)$. We can rewrite this as $p \cdot \frac{(\Sigma-1)(\Sigma-2)\ldots(\Sigma-p+1)(\Sigma-p)\ldots(1)}{(\Sigma-p)(\Sigma-p-1)\ldots(1)}$. This result can be summarized using the expression $p \cdot \frac{(\Sigma-1)!}{(\Sigma-p)!}$ and the permutation notation $p \cdot {}^{(\Sigma-1)}P_{(p-1)}$.

The automaton will thus evaluate $O(p \cdot {}^{(\Sigma-1)}P_{(p-1)})$ 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.

Generally, the scalability of property mining techniques is limited. The techniques tend to scale poorly with the growing size of the input trace L. Our technique is optimal in that respect as for both algorithms the runtime grows linearly with the increasing size of L. Given that $L \gg \Sigma$ and $L \gg p$, reading each line of the trace at most once is optimal. Finally, as mentioned earlier, since the majority of interesting properties are not complex, Σ^p will remain small in practice.

5 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 an 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 (Yang and Evans 2004b). By using these TRE variants as TRE templates, we mined

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

```
259018352791,13,INT_ENTR::0x00000044
259018354208,14,INT_HANDLER_ENTR::0x00000044
259018357541,15,INT_HANDLER_EXIT::0x00000044
259018368666,18,COMMSND_PULSE_EXE
259018370333,18,COMMSND_PULSE_EXE
259018371583,18,COMMSND_PULSE_EXE
...
259567335041,22,COMMREC_MESSAGE
259567336541,23,KER_EXITMSG_RECEIVEV/14
259567355458,13,INT_ENTR::0x00000044
259567366416,14,INT_HANDLER_ENTR::0x00000044
259567381750,15,INT_HANDLER_EXIT::0x00000044
259567384916,16,INT_EXIT::0x00000044
259567403625,25,COMMREPLY_MESSAGE
```

Fig. 3. Trace snippet from QNX *tracelogger*.

TRE instances from real-world system traces: QNX tracelogger and CAN communication traces. 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 number of distinct events in traces. For the experiments, we used a single eight-core machine equipped with the Intel i7-3820 CPU at 3.60GHz and 31.4Gb 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 parameterized 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): $(\hat{}(P|S)^*.(\langle P.\hat{}(P|S)^*.S.\hat{}(P|S)^*\rangle[0,3000]))+$
T-3 (multieffect): $(\hat{}(P|S)^*.(\langle P.\hat{}(P|S)^*.S.\hat{}(P)^*\rangle[0,3000]))+$
T-4 (multicause): $(\hat{}(P|S)^*.(\langle P.\hat{}(S)^*.S.\hat{}(P|S)^*\rangle[0,3000]))+$

5.2 Performance 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, and so forth. 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 they 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. The trace snippet in Figure 3 shows the sample trace used in our experiments.

Table 4 presents the results of the evaluation. The hexacopter trace used for the evaluation contained 1 million events, with 205 distinct events. We used the four TRE templates, T-1 to T-4,

Table 4. Mining TREs on QNX Traces

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

(a) Results of Mining TRE T-1

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

(b) Results of Mining TRE T-2

Support \ Confidence	**1**	**0.9**	**0.8**	**0.7**	**0.6**
1–1,000	0	38	52	67	72
1,000–5,000	0	35	47	60	64
5,000–10,000	0	30	40	50	53
10,000–50,000	0	39	39	40	40
50,000–80,000	0	14	14	16	16
80,000–300,000	0	3	3	3	3

(c) Results of Mining TRE T-3

Support \ Confidence	**1**	**0.9**	**0.8**	**0.7**	**0.6**
1–1,000	0	30	38	45	49
1,000–5,000	0	27	34	39	41
5,000–10,000	0	22	25	29	31
10,000–50,000	0	19	20	22	22
50,000–80,000	0	10	10	11	11
80,000–300,000	0	2	2	2	2

(d) Results of Mining TRE T-4

Table 5. Mining TREs on CAN Traces

Support \ Confidence	**1**	**0.9**	**0.8**	**0.7**	**0.6**
1–1,000	12	398	485	506	510
1,000–5,000	12	331	340	358	362
5,000–10,000	12	130	137	154	156
10,000–50,000	12	39	44	44	44
50,000–80,000	12	15	15	15	15
80,000–150,000	1	1	1	1	1

(a) Results of Mining TRE T-1

Support \ Confidence	**1**	**0.9**	**0.8**	**0.7**	**0.6**
1–1,000	6	55	102	118	131
1,000–5,000	5	12	15	20	27
5,000–10,000	5	12	15	18	23
10,000–50,000	5	11	14	15	16
50,000–80,000	5	11	11	11	11
80,000–150,000	1	1	1	1	1

(b) Results of Mining TRE T-2

Support \ Confidence	**1**	**0.9**	**0.8**	**0.7**	**0.6**
1–1,000	5	20	63	74	93
1,000–5,000	5	12	15	18	23
5,000–10,000	5	12	15	18	23
10,000–50,000	5	11	14	15	16
50,000–80,000	5	11	11	11	11
80,000–150,000	1	1	1	1	1

(c) Results of Mining TRE T-3

Support \ Confidence	**1**	**0.9**	**0.8**	**0.7**	**0.6**
1–1,000	8	85	194	255	288
1,000–5,000	7	41	55	87	96
5,000–10,000	7	34	47	78	87
10,000–50,000	7	34	41	59	63
50,000–80,000	7	30	30	30	30
80,000–150,000	1	1	1	1	1

(d) Results of Mining TRE T-4

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 5a that TRE T-1, the response property, occurs frequently in the trace for a large number of events. Due to an abundance of interprocess communication and system calls within the kernel, this is expected. The other three TREs are more complex and thus appear to be generally satisfied for a smaller number of event combinations.

To evaluate the interesting properties obtained for TRE T-1, we chose a confidence of 0.9 and support in the range of 10,000 to 50,000. It is found that that property is related to the interrupt handler in the QNX operating system. One of the most dominant properties has event `P:INT_ENTR` and `S:INT_EXIT`. Such a property is also intuitive to the designer as an interrupt enter is followed by an interrupt exit during normal execution of the system.

The results show that higher-threshold values for support and confidence result in fewer 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 4, the support threshold should be kept between 10,000 and 50,000 and the confidence threshold should be kept at 0.8 or higher.

5.3 Performance Evaluation Using Real CAN Traces

CAN bus is the primary communication medium over which the electrical control units (ECUs) within a vehicle exchange commands and information. Each message on the network can be distinguished by an associated ID, which is unique per message type. So, for instance, messages that contain the vehicle speed will have a unique and unchanging message ID. Messages transmitted on the network also have an associated timestamp. The actual contents of each message are encoded in a manufacturer-specific way. Our interest lies in the time and ID of transmitted messages, and not in the exact content of the message. We treat each message ID as an event occurring in the system.

For the evaluation, we collected a set of traces of CAN network activity produced from an idling vehicle with the engine on. 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 system designers.

Table 5 presents the results of the evaluation. The trace used for the evaluation contained 140,000 events, with 43 distinct events. We used the four TRE templates, T-1 to T-4, for the evaluation. The interval used in the templates is equivalent to that used previously in the evaluation of QNX traces. 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 5a that TRE T-1, the response property, occurs infrequently in the trace for a large number of events. This result resembles what we got using the QNX traces. The other three TREs are more complex and thus appear to be generally satisfied for a smaller number of event combinations.

The results show that higher-threshold values for support and confidence result in fewer specifications being reported. Let's again assume that an engineer would be able to assess around 30 specifications per property. Based on the results in Table 5, the support threshold should be kept between 10,000 and 50,000 and the confidence threshold should be kept at 0.9 or higher.

The thresholds required for both QNX and CAN traces are very similar. The number of mined specifications appears to be very sensitive to the size of the time interval in the TREs. This is evident from Table 6, where the interval in TRE T-1 was set to [0, 6,500] and the property was mined on the CAN trace. Comparing Table 6 to Table 5a makes it clear that many more properties are mined when the time interval is increased. Therefore, the interval size is the primary reason for the similarity between the derived threshold values.

Based on the shown results, we should also note that the TREs are satisfied infrequently for a lot more event combinations in CAN traces than in QNX traces. This is likely a consequence of a smaller event alphabet. CAN traces have a less versatile mix of events than QNX traces, and thus a larger number of CAN 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.

Table 6. Mining TRE T-1 with Interval [0, 6,500] on CAN Traces

Support \ Confidence	**1**	**0.9**	**0.8**	**0.7**	**0.6**
1–1,000	27	596	694	696	696
1,000–5,000	27	518	548	548	548
5,000–10,000	27	479	508	508	508
10,000–50,000	27	343	357	357	357
50,000–80,000	27	100	100	100	100
80,000–150,000	1	1	1	1	1

5.4 Performance 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. For instance, in real systems, we have little control over the number of distinct events used in the traces and thus cannot control the alphabet size. We also have little control of the exact length of the traces in real systems.

We synthesized a set of traces by varying the following three parameters: the distinct number of events (the size of the event 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 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 tree 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 four, and the number of variables in TRE templates was set to two. The total number of events in the traces was varied exponentially.

The results are shown in Figure 4. 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. 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 4, 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 two. As shown in Figure 5, 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

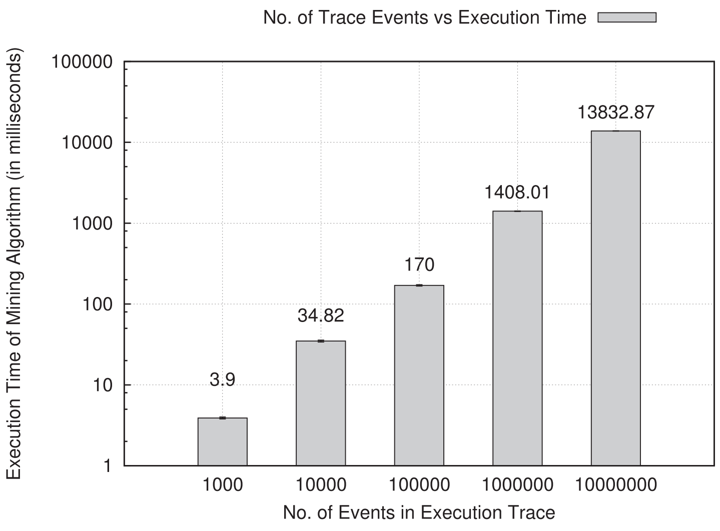

Fig. 4. Varying total number of events.

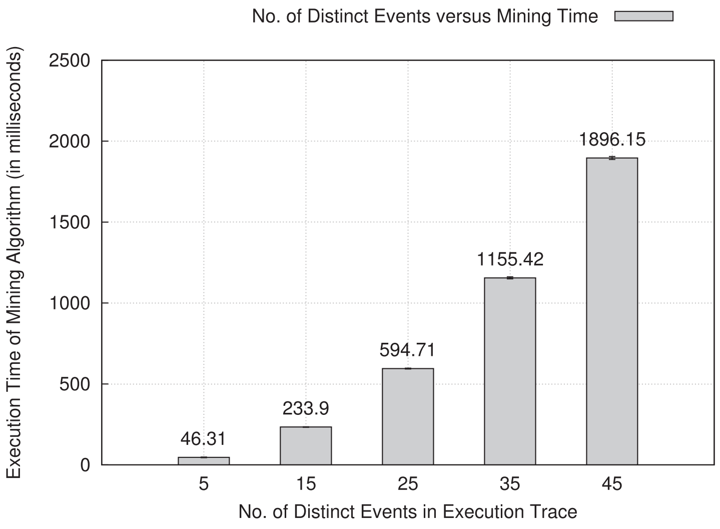

Fig. 5. Varying number of distinct events.

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 6 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 four variables is 4,156 milliseconds.

5.5 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

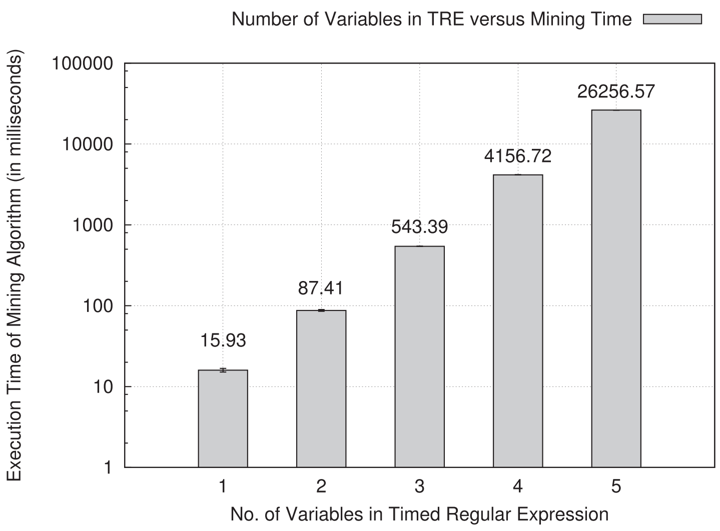

Fig. 6. Varying Number of Variables in TRE Templates.

Table 7. Comparing Algorithm 1 and Algorithm 2

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

Table 8. Comparing Execution Time of Mining Different Classes of TRE Templates

	Time without Negation (in Seconds)	**Time with Negation (in Seconds)**
Synthetic Traces	9.50	111.82

Table 9. Results for Real System Traces: QNX and CAN

Real System	**Time without Negation (in Seconds)**	**Time with Negation (in Seconds)**
CAN	1.03	21.83
QNX	34.38	3,498.37

templates listed in Table 7 were provided as input to the two algorithms when using synthetically generated traces.

For evaluation of real system traces, we used the TRE patterns T-1, T-2, T-3, and T-4.

Table 9 shows the results of the algorithm comparison on real traces generated from a QNX system. The trace used for the evaluation contained 1 million events, with 205 distinct events. 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 $\frac{205}{2}$.

Table 9 also shows the results of the algorithm comparison on real traces generated from a CAN network. The trace used for the evaluation contained 142,866 events, with 43 distinct events. Again, Table 9 clearly demonstrates that mining properties with negation takes longer than mining properties without negation, in this case 21.5 times longer. This corresponds to the expected speedup of $\frac{43}{2}$.

Lastly, Table 8 shows the results of the comparison on a synthesized trace. The size of the trace used for this experiment was 50,000 events, with 30 distinct events. It is evident from Table 8 that mining properties with negation takes approximately 12 times longer than mining properties without negation. This result also closely approaches the expected speedup of $\frac{\Sigma}{p}$ (see the Discussion section), which in this case is $\frac{30}{2}$.

6 CONCLUSION

This article 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 synthetically generated traces to analyze the scalability of the algorithms with an increase of variables in the TRE template, varying the total number of unique events and varying the total number of events in the system trace. Second, 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 the future, we propose to improve the framework to perform better on TREs with negation and gain considerable speedup as compared to TREs without negation.

REFERENCES

Rajeev Alur, Pavol Černý, Parthasarathy Madhusudan, and Wonhong Nam. 2005. Synthesis of interface specifications for Java classes. *ACM SIGPLAN Notices* 40, 1 (2005), 98–109.

Rajeev Alur and David L. Dill. 1994. A theory of timed automata. *Theoretical Computer Science* 126, 2 (1994), 183–235.

Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. *ACM Sigplan Notices* 37, 1 (2002), 4–16.

Eugene Asarin, Paul Caspi, and Oded Maler. 2002. Timed regular expressions. *Journal of the ACM* 49, 2 (2002), 172–206.

Marco Bonato, Giuseppe Di Guglielmo, Masahiro Fujita, Franco Fummi, and Graziano Pravadelli. 2012. Dynamic property mining for embedded software. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM, New York, 187–196. 100104.

Marat Boshernitsan, Roongko Doong, and Alberto Savoia. 2006. 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*. ACM, New York, 169–180. 594061.

Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. 2008. Mining specifications of malicious behavior. In *Proceedings of the 1st India Software Engineering Conference*. ACM, New York, 5–14.

Greta Cutulenco, Yogi Joshi, Apurva Narayan, and Sebastian Fischmeister. 2016. Mining timed regular expressions from system traces. In *Proceedings of the 5th International Workshop on Software Mining*. 3–10. DOI: http://dx.doi.org/10.1145/2975961.2975962

Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Gordon Fraser, Sebastian Hack, and Andreas Zeller. 2012. Automatically generating test cases for specification mining. *IEEE Transactions on Software Engineering* 38, 2 (2012), 243–257.

Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. 2010. Generating test cases for specification mining. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ACM, New York, 85–96. 594101.

Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. 2007. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems* 35, 3 (2007), 239–272. DOI: http://dx.doi.org/10.1007/s11241-007-9012-7

Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering*. IEEE, 411–420.

Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. *SIGOPS Operating Systems Review* 35, Article 5 (Oct. 2001), 16 pages. DOI: http://dx.doi.org/10.1145/502059.502041

Michael D. Ernst. 2003. Static and dynamic analysis: Synergy and duality. In *ICSE Workshop on Dynamic Analysis (WODA'03)*. IEEE Computer Society, 24–27.

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123.

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1 (2007), 35–45.

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

E. Mark Gold. 1978. Complexity of automaton identification from given data. *Information and Control* 37, 3 (1978), 302–320.

Sudheendra Hangal, Naveen Chandra, Sridhar Narayanan, and Sandeep Chakravorty. 2005. IODINE: A tool to automatically infer dynamic invariants for hardware designs. In *Proceedings of the 42nd Annual Design Automation Conference.* ACM, New York, 775–778. 477050.

Sudheendra Hangal and Monica S. Lam. 2002. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering.* ACM, New York, 291–301. 592020.

Reed Hastings and Bob Joyce. 1992. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 Usenix Conference.* USENIX, San Francisco, CA, 125–136.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Zachary Kincaid and Andreas Podelski. 2015. Automated program verification. In *Proceedings of Language and Automata Theory and Applications: 9th International Conference, (LATA'15).* Vol. 8977. Springer, Nice, France, 25.

Rob Krten. 1999. *Getting Started with QNX Neutrino 2: A Guide for Realtime Programmers.* PARSE Software Devices, Ottawa, Canada.

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

Kim G. Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a Nutshell. In *International Journal on Software Tools for Technology Transfer (STTT'97)* 1, 1 (1997), 134–152.

Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General LTL specification mining. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15).* ACM, New York, 81–92.

Wenchao Li, Alessandro Forin, and Sanjit A. Seshia. 2010. Scalable specification mining for verification and diagnosis. In *Proceedings of the 47th Design Automation Conference.* ACM, New York, 755–760. ACM Order No.: 4770101.

Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic generation of software behavioral models. In *Proceedings of the 30th International Conference on Software Engineering.* ACM, New York, 501–510. 529080.

Jeff H. Perkins and Michael D. Ernst. 2004. Efficient incremental algorithms for dynamic detection of likely invariants. In *ACM SIGSOFT Software Engineering Notes.* ACM, New York, 23–32.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.

Jinlin Yang and David Evans. 2004a. Automatically inferring temporal properties for program evolution. In *15th International Symposium on Software Reliability Engineering, 2004 (ISSRE'04).* IEEE, 340–351.

Jinlin Yang and David Evans. 2004b. Dynamically inferring temporal properties. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04).* ACM, New York, 23–28. DOI: http://dx.doi.org/10.1145/996821.996832

Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: Mining temporal API rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering.* ACM, New York, 282–291. 592060.

Received July 2017; revised September 2017; accepted September 2017