TREM: A Tool for Mining Timed Regular Specifications from System Traces

Lukas Schmidt
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, ON N2L 3G1
Email: lfschmid@uwaterloo.ca

Apurva Narayan
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, ON N2L 3G1
Email: apurva.narayan@uwaterloo.ca

Sebastian Fischmeister
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, ON N2L 3G1
Email: sfischme@uwaterloo.ca

***Abstract*—Software specifications are useful for software validation, model checking, runtime verification, debugging, monitoring, etc. In context of safety-critical real-time systems, temporal properties play an important role. However, temporal properties are rarely present due to the complexity and evolutionary nature of software systems.**

We propose Timed Regular Expression Mining (TREM) a hosted tool for specification mining using timed regular expressions (TREs). It is designed for easy and robust mining of dominant temporal properties. TREM uses an abstract structure of the property; the framework constructs a finite state machine to serve as an acceptor. TREM is scalable, easy to access/use, and platform independent specification mining framework. The tool is tested on industrial strength software system traces such as the QNX real-time operating system using traces with more than 1.5 Million entries. The tool demonstration video can be accessed here: youtu.be/cSd_aj3_LH8

***Index Terms*—Specification Mining, Timed Regular Expressions, Real-time systems**

I. Introduction

Temporal behavior of programs is an extensively studied topic [1], [2]. Recently, the idea of mining likely temporal properties of programs from system traces has become popular [1]. 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) [3]. These activities include software testing [4], automated program verification [5], anomaly detection [6], debugging [7], etc.

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. Further, mined specifications can assist automated verification techniques because they provide an easy and user-friendly way to describe a programs' specifications.

II. Related Work

Specification mining has gained significant attention in recent times. Different techniques have been developed for mining specifications from templates expressed using regular expressions, LTL [8], STL [9], and other custom formats. There are numerous temporal property mining tools [10], [11], [12], [13], [14], [15], [16]. As argued by Ammons et. al. [10], automated verification techniques are unlikely to be widely adopted unless cheaper and easier ways of formulating specifications are developed.

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 [10] they learn a state machine which captures both temporal properties and data dependencies. These find utility in identifying errors and refining specifications. These also find use in automatic verification tools to find bugs in the program execution.

These approaches suffer from one main drawbacks. Mining of a single state machine from system traces is a NP-hard problem [17].

Another work based on the intuition that frequently occurring behavior that matches temporal patterns are likely to be true is the foundation of Peracotta [18], [12], [13].

Specifications refer to properties of systems, for example, a specification of a typical smart phone might be 'within 5 seconds of pushing the power button the screen should turn on'. Such specifications find applications in various domains, for example, anomaly detection in networks [19], specification-based testing in software development, and formal verification in hardware [20]. The work on timed regular expression (TRE) mining by Cutulenco et. al. [21] is the backbone of this tool. The work is similar to other temporal mining frameworks such as MONTRE [22].

This paper presents TREM a hosted specification mining platform designed for easy and robust mining of temporal properties. The automatic generation of properties is useful for finding missing program specifications, debugging during the software development life-cycle, exploration/understanding of legacy or undocumented software, and anomaly detection.

Section III discusses the design and work-flow of TREM. Section IV provides an overview of the implied methodology of TREM through an industrial strength case study. Finally, Section V concludes the paper by evaluating TREM.

III. Workflow and Design

A. Overview of the Specification Mining Process Used

TREM mines specifications of systems by analyzing the sequence of events emitted by the system. These sequences of events are called Traces. Traces must take the form of an event series with time stamps. The alphabet of a trace is the set of all events that the trace contains.

Definition 1 (Trace, Event, and Alphabet): An event is a string. A trace is a sequence of events with a time stamp for each event. The alphabet of a trace is the set of all events in the trace.

For example, a trace of a smart phone may look like the trace displayed in Table I. In this case, the alphabet is `Power_Button, Graphics_Loaded, Screen_On`. The elements of the alphabet are referred to as events.

TABLE I
Sample Trace

Time (seconds)	Event
0	Power_Button
2	Graphics_Loaded
3	Screen_On

Specifications are mined using behavioural pattern templates. TREM uses Timed Regular Expressions (TREs) [23] to encode specifications and Timed Regular Expression Template (TRET) to encode behavioural patterns. Both TREs and TRETs are defined below.

Definition 2 (Timed Regular Expression): Timed regular expressions (TREs) over an alphabet Σ (also referred to as Σ expressions) are defined using the following families of rules:

- $\underline{a}$ for every event $a \in \Sigma$ 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.

Here . represents the concatenation operator, $\hat{}$ is the negation operator, | is the disjunction or OR operator, and * denotes the Kleene-*. These operators only allow for specifications on the order of occurrence of events without considering the time at which these events occur. The $\langle\phi\rangle_I$ operator allows for the creation of temporal properties; this operator restricts the metric length of the time-event sequences in $[\phi]$ to be in the interval I. It is important to note that we use TREs, as defined above, to provide specifications.

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

Definition 4 (TRE Template (TRET)): A TRE template is a TRE in which all of the atomic propositions are either event variables or events.

We use the term *event variable* to denote a placeholder 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).

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

A TRE instance corresponds to a TRE template with an identical 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.

The mining framework TREM uses timed automaton for mining properties. Timed automata [24] have been investigated quite rigorously in the recent past. The main motivation for using timed automata is their suitability for modeling time dependent behavior, and the ability to monitor their reachability [25]. Timed automata are equipped with clocks, making them perfect for modelling and verification of real-time systems' behavior [24]. Classical models like finite automata, Petri-nets, etc., are not suitable since they cannot express such explicit timing constraints naturally present in real-life systems. Another important property of the timed automata is that the reachability properties are decidable [24], even though the timed automata have an infinite number of configurations. The main idea behind this result is the construction of a region-automaton, which finitely abstracts the behavior of timed automata in a way that checking reachability in a timed automaton reduces to checking reachability in a finite automaton. More details on the implementation can be found in [21].

The TRE instances generated by the binding contain every permutation of events in the alphabet within the TRET. 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. We use the concepts of support and confidence from [1][26] for finding dominant properties.

Definition 6 (Support): The support of a TRE instance π on a trace t is the number of time points of trace t which *do not* falsify π.

Definition 7 (Confidence): The confidence of TRE instance π on a trace t is the ratio of trace support to trace number of total instances.

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.

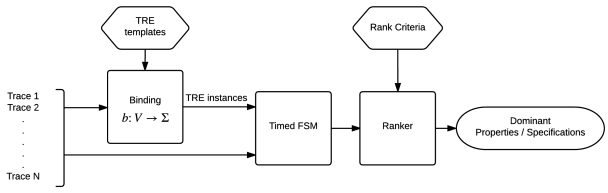

Fig. 1. TRE Mining workflow [21]

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

The timed FSM evaluates the TRE instances on the same set of N traces. The Σ^p TRE instances are encoded 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 a 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 work-flow 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 finite state machine (FSM) iterates over the time event pairs in a trace and at each new event evaluates the relevant TRE instances. Lets 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 time stamp is 6, the failure count would increase. If the next event is "receive" and the time stamp 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.

B. Interface Design Overview

TREM is a web application. The front end is responsible for providing an interface to work with TREs, set up experiments, view documentation, and display results.

The back end is responsible for specification mining and trace storage. This allows the front end to be light and delegate the computationally intensive processes to the server (Figure 2).

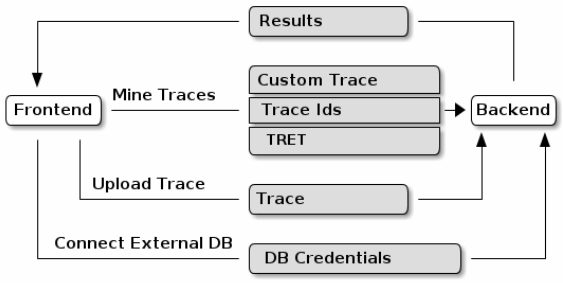

Fig. 2. Overview of interface

C. Frontend Design

The front end of TREM provides multiple ways to create TRETs, select or upload traces, and display results (Figure 3).

To create a TRET we provide 3 options:

- Visual TRET creation using Blockly: We use Google's library Blockly to create a visual interface to enter TRETs. There is a Blockly block equivalent for every rule in the syntax required to write TRET's. Therefore, the Blockly interface has the full expressive power over

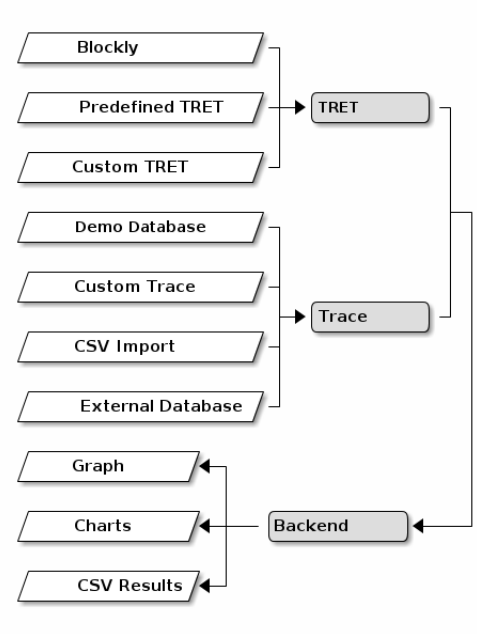

Fig. 3. Overview of front end interface

TRETs. Figure 4 shows a sample Blockly expression. It is read from top to bottom, for example in Figure 4, the equivalent TRET is `<0.1>[0,3].`

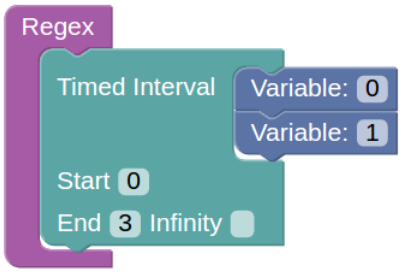

Fig. 4. Simple Blockly expression for a TRE

- Pre-defined TRETs: There is also a list of commonly used pre-defined TRETs based on [2] with accompanying documentation. Figure 5 shows a sample pre-defined TRET with its explanation.

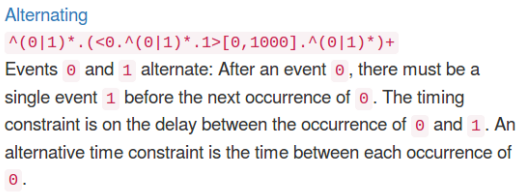

Fig. 5. Pre-defined TRET with its explanation

- Direct TRET Creation: Finally, there is also a field which allows the user to directly enter their own TRETs.

Blockly TRET models may be converted to and from text TRETs. This is encouraged to make the user comfortable with the syntax of TRETs over time.

There are three ways to provide the traces for evaluation to the tool as outlined below:

- Database Connection: The tool provides option to connect any Postgres database. A list of traces are shown for each database connected to TREM, any number of these may be selected. Each comes with the option to limit the length of the traces.
- Custom Trace: A field is provided into which a short custom trace may be written. A pattern is mined on all the selected traces and the custom trace if present.
- CSV File: Traces can also be uploaded in a csv file format.

The results are displayed in three different ways for ease of visualization and understanding of the end user (Figure 6).

- Histogram: For immediate visual feedback, a histogram is generated that displays the event combinations resulting in the largest success values. The histogram displays the event combination, success and reset values.
- Tables: There are 2 tables displayed per trace, the first table is the equivalent of the histogram. It displays the event combination, success and reset values in the decreasing order of successes. The second table displays all event combinations that have sufficient support and confidence values when compared to the respective threshold values.
- CSV file: Finally, a CSV file is generated with the entire set of results. The results take the form of matrices. The first one contains the success values, the second one contains the reset values, and a list of the event combinations that passed the support and confidence threshold.

D. Interface Backend Description

The back end is made up of a demo_database, connections to external databases, and the interface with the TRE mining framework.

During execution, the relevant traces are loaded, the TRET is parsed, and the traces are then mined by the TRE mining framework [21]. The results are processed and returned to the graphical user interface.

IV. Industrial Case Study

A. 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, etc. The QNX RTOS has a very advanced logging facility, *tracelogger*. Tracelogger facilitates detailed tracing of the kernel and user process activity on any system. More specifically, it can log interrupt activity, various states of processes and threads, communication within the system, kernel calls, custom user events, and much more. The logged events give a detailed view into the behavior of the system, but due to the large quantity of the produced information are often difficult to make use of by developers and system designers. These traces are thus a perfect resource for dynamic mining of system properties and specifications.

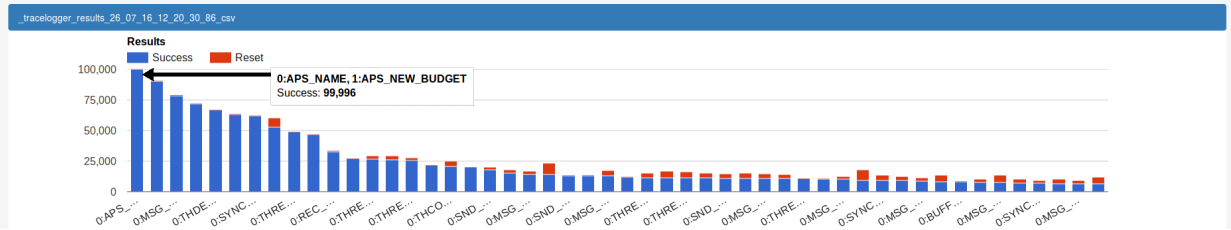

Fig. 6. TRE mining results of QNX trace from hexacopter using the alternating TRET

For the evaluation, we used a set of traces collected from an operational hexacopter loaded with the QNX Neutrino 6.4 and a user control process. The vehicle is a commercially available gyro-stabilized Mikrokopter hexacopter equipped with 6 electric motors and a 6200 mAh lithium polymer battery. The hexacopter can be seen in Figure 7. The trace length for our expriments have 1.6 million event entries.

Fig. 7. The hexacopter in flight

The trace is uploaded into a database and then connected to TREM via the 'Connect Database' option. For evaluation we use the pre-defined TRET, the "alternating" pattern [2] as it is quite common in real-time embedded systems. The intuition for using an alternating pattern arises from the fact that a vast majority of tasks in real-time systems are periodic.

One can either select from our pop-up screen or customize it as per the system under consideration either in text or Blockly interface. For example, the time constraint for each system would be unique and depend on the time units of the system trace. In our case, we modify the time constraint from [0, 1000] to [0, 4000] since that is a desirable value for inter-event time lapse for QNX traces from hexacopter. The histogram view of the results is presented in Figure 6. The results presented here are for support = 1 and confidence = 0.9. These parameters can be changed to refine the results for better understanding of the system under evaluation.

B. Result Analysis

The properties mined by TREM under specified thresholds for support and confidence are analyzed for understanding the system. The CSV file contains all the mined specification that take the form of the alternating pattern. Our goal is to identify dominant and interesting properties of the system. Therefore, we sort the TRE instances that obey the thresholds of support and confidence based on their frequency of occurrence in the system trace.

In Figure 6, the mined properties from the hexacopter trace are evaluated, each bar represents a unique instance of a TRE. In this case, the most dominant property is where, the placeholder `0` is associated with the event `APS_NAME` and the placeholder `1` is associated with `APS_NEW_BUDGET`. These events are associated with QNX and ensure that there is sufficient memory available all the time for the application. The event `APS_NAME` creates a new partition to ensure sufficient memory is available. The event `APS_NEW_BUDGET` is emitted automatically when the adaptive partitioning scheduler clears a critical budget as part of handling a bankruptcy situation. The timing constraint ensure the timely response to a memory request, which is extremely critical in real-time systems. This is just one of the dominant properties and is very important as it ensures that in a safety critical real-time application such as a hexacopter, there is always sufficient memory available.

The significance of such properties is better understood in case of system failure. If the system prohibits memory provisioning for the application, it may lead to catastrophic consequences. For example, in case of hexacopter, the autopilot module tries to obtain direction information from gyroscope to send a control signal to the elevation controller. The inertial measurement unit (IMU) requires memory to store and process this information in real-time. For a faulty system (the fault can be in IMU or memory management system), it is predicted that TREM results would reveal that either the property described above is completely missing or has more resets. This indicates that the system is trying to clear memory but unable to do so. Not only does this allow us to monitor anomalous behavior of the system, but also helps in post-mortem analysis.

V. Discussions and Conclusions

A. Evaluation and Scalability

TREM's server side is written in python using the micro-web framework Flask [27]. TREM delegates all of the memory and processor intensive tasks such as trace storage and mining to the server.

TREM requires minimal resources on the client device. This allows TREM to scale to handle large system traces. We have tested the framework on industrial strength system

traces from QNX, Controller Area Network (CAN) bus, and Robotics Operating System (ROS) messages of length more than 2 million event entries.

The space and time complexity of our framework depends on the complexity of the TRET. It is exponential w.r.t the number of variables or placeholders in the TRET whereas linear w.r.t to the length of the trace. Generally, most properties in systems are simple with only 2 or 3 variables. The detailed derivation of the space and time requirements of our framework has been presented in [21].

B. Conclusion

Since TREM is a web app, the client is platform independent and lightweight. This allows it to be run on most modern terminal devices with a web browser. TREM is containerized through Docker, and can be deployed on most existing servers.

TREM allows for fast iteration of specification mining patterns or TRETs due to the following:

- Processor intensive tasks are done on the server
- Traces may be shortened quickly and easily to allow for testing
- Short custom traces may be entered to quickly test different variants of your TRETs

TREM provisions for mining of TRETs combined with the visual interface and a visual summary of results. This enables learning and rapid development of specifications for complex software, embedded software, legacy systems, and evolution of software systems.

REFERENCES

[1] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General LTL Specification Mining. In *Proceedings of the 2015 30th IEEE/ACM*

[2] 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, New York, NY, USA, 1999. IEEE.

[3] Manuvir Das. *Formal Specifications on Industrial-Strength Code—From Myth to Reality*, pages 1–1. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[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, New York, NY, USA, 2010. ACM. 594101.

[5] 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, Nice, France, 2015. Springer.

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

[7] 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, New York, NY, USA, 2010. ACM.

[8] Marco Bonato, Giuseppe Di Guglielmo, Masahiro Fujita, Franco Fummi, and Graziano Pravadelli. Dynamic property mining for embedded software. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 187–196, New York, NY, USA, 2012. ACM. 100104.

International Conference on Automated Software Engineering (ASE), ASE '15, pages 81–92, Washington, DC, USA, 2015. IEEE Computer Society.

[9] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V Deshmukh, and Sanjit A Seshia. Mining requirements from closed-loop control models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1704–1717, 2015.

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

[11] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35:57–72, October 2001.

[12] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*, pages 282–291, New York, NY, USA, 2006. ACM. 592060.

[13] Jinlin Yang and David Evans. Automatically inferring temporal properties for program evolution. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 340–351, Saint-Malo, Bretagne, France, 2004. IEEE.

[14] Fabrizio Pastore, Daniela Micucci, and Leonardo Mariani. Timed k-tail: Automatic inference of timed automata. *CoRR*, abs/1705.08399, 2017.

[15] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. Behavioral resource-aware model inference. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 19–30, New York, NY, USA, 2014. ACM.

[16] Marc Brünink and David S. Rosenblum. Mining performance specifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 39–49, New York, NY, USA, 2016. ACM.

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

[18] Jinlin Yang and David Evans. Dynamically Inferring Temporal Properties. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04, pages 23–28, New York, NY, USA, 2004. ACM.

[19] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Specification-based anomaly detection: A new approach for detecting network intrusions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 265–274, New York, NY, USA, 2002. ACM.

[20] Zerksis D. Umrigar and Vijay Pitchumani. Formal verification of a real-time hardware design. In *Proceedings of the 20th Design Automation Conference*, DAC '83, pages 221–227, Piscataway, NJ, USA, 1983. IEEE Press.

[21] Greta Cutulenco, Yogi Joshi, Apurva Narayan, and Sebastian Fischmeister. Mining timed regular expressions from system traces. In *Proceedings of the 5th International Workshop on Software Mining*, pages 3 – 10, Singapore, 2016.

[22] Dogan Ulus. Montre: A tool for monitoring timed regular expressions. *arXiv preprint arXiv:1605.05963*, 2016.

[23] Paul Caspi Eugene Asarin and Oded Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, March 2002.

[24] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[25] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell, 1997.

[26] Tuan-Anh Doan, David Lo, Shahar Maoz, and Siau-Cheng Khoo. Lm: A miner for scenario-based specifications. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 319–320, New York, NY, USA, 2010. ACM.

[27] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, Inc., 1st edition, 2014.