System Call Logs with Natural Random Faults: Experimental Design and Application

Apurva Narayan, Sean Kauffman, Jack Morgan, Guy Martin Tchamgoue, Yogi Joshi, Sebastian Fischmeister, Chris Hobbs

***Abstract*—In this paper we present a large dataset for use in embedded systems research which has been gathered from a realistic development environment operating in the path of an accelerated neutron beam. The dataset contains traces with events from a real-time operating system as well as user events from a safety critical application. All data is carefully timestamped and in human understandable form. We present two use cases for this dataset: mining timed regular expressions to extract system specifications from clean and possibly anomalous data generated during the operation of the neutron beam, and runtime monitoring to extract information from traces with incomplete information. The dataset is available for research at: http://doi.org/10.5281/zenodo.248008**

I. Introduction

Errors in the computer memory system that change an instruction or a data value in a program lead to software related issues that can be resolved by cold booting the system. These kind of errors are also referred to as *soft* errors. A soft error will not damage the system hardware; the damage only occurs to the data being processed. However, such errors can be catastrophic if they occur in safety-critical systems.

The design and implementation of embedded systems are complex tasks that involve developing heterogeneous subcomponents. Development of strategies to monitor and analyze these errors in embedded systems is a challenge [1]. Analysis of event traces provides an approach for studying the conformity of embedded systems behavior to specified requirements [2], [3].

One challenge of the trace analysis approach is the requirement for *good* datasets to design and test algorithms for tasks such as *runtime monitoring*, *specification mining*, and *anomaly detection*. Here, a good dataset can be classified as the one in which there are data without any errors or system faults and fragments of data where the system under consideration is exposed to random errors. It is common practice to create artificial datasets with manually induced errors during system execution for the design of novel algorithms.

Modern Systems-on-Chips (SoCs) must undergo experiments to evaluate their reliability. SoCs are found in almost every electronic device, from consumer electronics like refrigerators and televisions to devices in safety critical applications such as biomedical implants, airplanes, satellites, and autonomous vehicles. Most of these devices are exposed to radiation to induce failures during testing. Failure rates under these conditions are not negligible even in consumer electronics [4], [5] due to the shrinking size of transistors, and they can have a catastrophic impact when occurring in safety critical applications such as airplanes [6].

A. Narayan, S. Kauffman, J. Morgan, G. M. Tchamgoue, and S. Fischmeister are with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, N2L 3G1 Canada e-mail: (a22naray, skauffman, jamorgan, gmtchamg, y2joshi, sfischme)@uwaterloo.ca.

C. Hobbs is at QNX Software Systems, 1001 Farrar Road, Ottawa, ON K2K 0B3 Canada email: chobbs@qnx.com

Our idea is to use the same radiation testing that commercial SoCs undergo to generate good, genuine datasets for the design and evaluation of algorithms. By using a radiation experiment to generate soft errors, we can avoid the bias inherent in the injection of manual errors into a dataset. Because the error rates are known to be significant, we can expect errors to appear frequently enough to be useful.

The rest of the paper is organizes as follows: in Section II, we provide a general overview of the approach adopted for the conduct of such an experiment. Section III delves deeper into the experimental setup and explains the parameters, applications, and metrics used for analysis and control. In Section IV, we present two use cases of our datasets in the context of system analysis and monitoring. The first use case is Timed Regular Expression (TRE) mining, a technique for mining system specifications with timing constraints, and the second is an evaluation of a runtime monitoring technique that is fault tolerant and designed specifically for lossy traces. In Section V we present a discussion on lessons learned from the experiment and the steps required to clean the data to make it available to others for evaluation of their algorithms. We provide concluding remarks in Section VI.

II. Approach

In this work, we present a dataset acquired by running a safety critical application in a high-radiation environment typically used for testing resilience to soft errors. Four development boards were exposed to an accelerated neutron beam in a test facility at the Los Alamos Neutron Science Center (LANSCE), New Mexico, USA. The boards ran the application-under-test continuously for a period of five days while exposed to the neutron beam. The dataset is comprised of the system trace from each development board with various configurations of the application.

This dataset provides high quality system traces with random effects from the ionization beam on the application. The dataset is valuable for any kind of analysis in the domain of anomaly detection or prediction.

Figure 1 provides a high level overview of our approach to collect data from a real time operating system running on a embedded development board with a safety critical application exposed to a ionization beam.

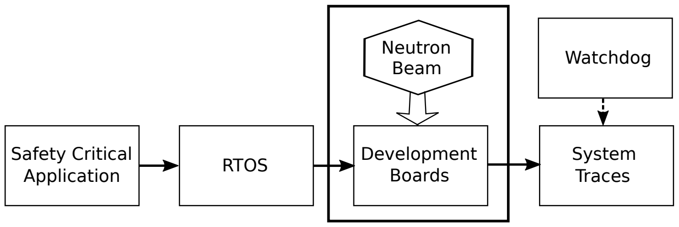

Fig. 1. An overview of the experimental setup

In Figure 1, the Real-Time Operating System (RTOS) receives a safety critical application for execution. In this case the application is an automotive *cruise-control* simulation and the RTOS under consideration is QNX 6.6.0 [7]. The RTOS with application is loaded onto the on-chip Static Random Access Memory (SRAM) memory of the development boards to avoid damage that is possible when writing to flash memory during neutron bombardment. The application is designed to generate *user events* in the system trace to identify the state of the application. The system traces are collected on a hard disk connected to the an external control computer away from the ionization beam's influence. A custom designed *watchdog* program continuously monitors the trace logs generated by the boards under test. In the event that logs have not been written to for a given time, the script reprograms the board and the cycle continues.

III. Experimental Setup

We placed the SoCs of four Xilinx development boards in the path of an accelerated neutron beam at LANSCE in New Mexico, USA. Two of the development boards were the Xilinx ZC702 featuring a XC7Z020 SoC, while the other two boards were the Xilinx ZC706 featuring a XC7Z045 SoC [8]. The boards were programmed via their onboard Digilent Joint Test Action Group (JTAG) debuggers connected via Universal Serial Bus (USB) to control computers and programmed using a Xilinx Microprocessor Debugger (XMD) console interface. Power to the boards was supplied using an APC switched rack Power Distribution Unit (PDU) controlled via Ethernet, allowing the boards to be power-cycled remotely. The control computers were a Lenovo T420 and a Lenovo T430 laptop running Ubuntu Linux 16.04. Each of the boards under test were placed a different physical distance from the origin of the beam, so they received different numbers of accelerated neutrons.

Each SoC ran identical software apart from differences to account for hardware variations between the ZC702 and ZC706 models. The software was an automotive cruise-control simulation developed by QNX Software Systems and running on version 6.6.0 of the QNX real-time operating system [7] out of SRAM. After booting, QNX was configured to write system logs using the tracelogger utility to a mounted Network File System (NFS) share served by the control computers over Ethernet. Tracelogger was configured with the flags `-c -s 60 -f <log filename>`. Watchdog software running on the control machines monitored the logs and reprogrammed an SoC if its logs stopped growing - an indication that the SoC was no longer functioning properly. If a board failed to boot after reprogramming, it was power-cycled by the watchdog using the APC PDU.

The cruise-control software attempts to keep the driving speed of a simulated car at 100 kilometers per hour. An artificial, randomized speed sensor periodically sends signals to multiple, identical controller programs which calculate the necessary brake or accelerator pressure to correct the value. If the controller programs are not unanimous in their calculations, an error is logged. If they are unable to reach a quorum, an error is logged and the simulation is terminated.

Figure 2 shows a high level overview of the cruise-control software. The simulated speed sensor on the left represents the randomized value sent to the brake and accelerator controllers, represented by the boxes in the center of the diagram. The number of controller programs is parameterized from 1 to N, with each controller receiving an identical speed value from the simulated sensor. Once their calculations are complete their results are compared to see if a unanimous answer, or at least a quorum is reached.

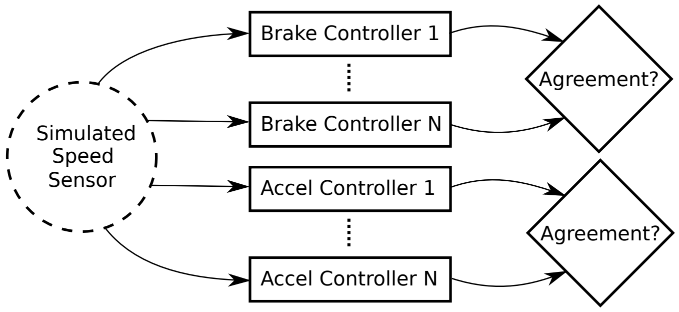

Fig. 2. High level overview of the software-under-test

The simulation had two parameters: the number of controller programs to run, and the probability of injecting a synthetic software error. In the initial configuration, the program was run with three controllers and a 1% chance of injecting a software error. After the experiment had stabilized, the probability of injecting software errors was set to 0% and four different quantities of controller programs were tested: two, three, five, and seven.

IV. Use cases

As an example, this section presents two use cases of the dataset: mining of TRE [9] to extract system specifications and runtime verification of Linear Temporal Logic (LTL) [10] properties.

A. TRE Mining

Regular expressions offer a declarative way to express the patterns for any system property or specification. Every language defined by a regular expression can be recognized by a finite automaton [11]. It is possible to convert any regular expression into a non-deterministic automaton, and further to convert from a non-deterministic to a deterministic automaton. We can thus generate a classical Deterministic

Finite Automaton (DFA) for any property expressed as a regular expression.

Classical automata theory handles only the *qualitative* notion of time, i.e. a sequence of events specifies the ordering of events but not the time between the occurrence of those events. Such a qualitative abstraction is useful for the analysis of certain systems, but many other real-time, safety-critical application domains require more detailed models which include accurate timing information. For example, we might want to modify a formal specification *"a is followed by b"* to a more precise specification with timing constraints *"a is followed by b within x seconds"*. Since our focus lies on real-time safety-critical systems, we use the technique for mining specifications that include the relevant timing information using the formalism of TREs [3].

The following four TRE templates are used for the evaluation of traces. The templates are parametrized with a time interval of 0 to 1,500.

T-1(response): $(\hat{}P)^*.((\langle P.(\hat{}S)^*.S\rangle[0,1500]).(\hat{}P)^*$
T-2(alternating): $(\hat{}P,S)^*.((\langle P.(\hat{}P,S)^*.S\rangle[0,1500]).(\hat{}P,S)^*$
T-3(multi-effect): $(\hat{}P,S)^*.((\langle P.(\hat{}P,S)^*.S\rangle[0,1500]).(\hat{}P)^*$
T-4(multi-cause): $(\hat{}P,S)^*.((\langle P.(\hat{}S)^*.S\rangle[0,1500]).(\hat{}P,S)^*$

The trace used for the evaluation contains a minimum of three million events, with 139 distinct events. We used the four TRE templates, T-1 to T-4, for evaluation. The interval used in the templates is sufficient for most interesting interactions to complete. The tables below report the most dominant specifications mined by our algorithm in traces with 2 controllers. We mined the dominance and presence of properties in traces collected during the experiment with injection of the ionization beam and clean traces collected in the lab with no interference. The results are presented in Table I and Table II for both cases. The results indicate that the template T-1 is the most dominant with the five most dominant properties presented in the table. The properties ranked 3 and 5 might look apparently similar but they are associated with class of `interrupt enter` and `interrupt handler enter` but for different interrupts `(0x00000036), (0x0000001d)` respectively.

TABLE I
DOMINANT PROPERTIES OBTAINED FOR TRE - T-1 WITH FAULT INJECTION

Rank	Instance Counts	Actual Events
1	45765	`P:CONTROL TIME, S:COMMSND_PULSE_EXE`
2	12613	`P:INT_EXIT(0x00000036), S:THREAD READY`
3	10976	`P:INT_ENTR(0x00000036), S:INT_HANDLER_ENTR(0x00000036)`
4	9598	`P:INT_HANDLER_EXIT(0x00000036), S:THREAD RUNNING`
5	9381	`P:INT_ENTR(0x0000001d), S:INT_HANDLER_ENTR(0x0000001d)`

We can similarly mine more TREs to determine the properties of the complex embedded software where event sequence and timing is important. In the above results we see the dominance of the property at rank 3 is different in the two cases which might be an effect of fault injection. We may be able to compare the situations by crafting an interesting TRE that could reveal more in-depth workings of the application and faults occurring due to impact of the ionization beam.

TABLE II
DOMINANT PROPERTIES OBTAINED FOR TRE - T-1 WITHOUT FAULT INJECTION

Rank	Instance Counts	Actual Events
1	44644	`P:CONTROL TIME, S:COMMSND_PULSE_EXE`
2	6269	`P:INT_EXIT(0x00000036), S:THREAD READY`
3	5721	`P:CONTROL TIME, S:THREAD RUNNING`
4	4965	`P:INT_HANDLER_EXIT(0x00000036), S:THREAD RUNNING`
5	4347	`P:INT_HANDLER_EXIT(0x000001d), S:INT_EXIT(0x000001d)`

B. Runtime Monitoring

Runtime verification (RV) is the problem of, given a program P and an execution trace σ of P along with a specification φ, deciding whether σ satisfies φ. A monitor $\mathcal{M}^\varphi$ is synthesized for φ. Thus, RV aims to find whether P exhibits the behavior described by φ, but generally requires the existence of a complete execution trace [12], [13].

Real-world applications, however, often produce lossy traces due to reasons which include lossy network protocols, logging failures, sampling-based profiling, and partial instrumentation. A computing system subjected to radiation, as is the case in this paper, may produce errors and may even fail. The portion of the trace obtained during the period of exposure to radiation should not be trusted but rather be considered as lossy.

A sound monitor for complete traces may deliver an incorrect verdict on lossy traces [14] or on execution traces obtained under recording uncertainty [15]. A loss-tolerant monitor $\mathcal{M}^\varphi$ verifies a specification φ on a lossy trace σ. A lossy interval in a trace σ represents an interval of time in which events may be produced by a *program*, but none of such events are observed by the corresponding *monitor*. Thus, whenever the logger fails to observe events or simply cannot record them precisely due either to soft or firm errors, we mark the corresponding intervals as lossy intervals in the final trace.

In this paper, we use the loss-tolerant monitoring technique and tool developed by Joshi *et al.* [14] to evaluate the following LTL properties on the dataset. This monitor produces a sound verdict on lossy traces without any other requirements on the lost events. However, the lossy input trace should not end with a lossy interval. The loss-tolerant monitor accepts an input trace σ and the LTL formula φ to be evaluated on σ. The output of the monitor is in the truth-domain $\mathbb{B}_5 = \{\top, \top_P, ?, \bot, \bot_P\}$, which is the truth-domain of an RV-LTL monitor [16] augmented with a '?' for unknown (i.e., lost) events. In this definition, the monitor outputs $\top$ for true (respectively $\bot$ for false) when the input trace satisfies (respectively violates) the property. Similarly, a verdict of $\top_P$ *presumably* true (respectively $\bot_P$ for *presumably* false) is generated when the trace presumably satisfies (respectively violates) the monitored property.

We evaluated the following LTL properties on the dataset:

P1. $\Box(thread\_create \rightarrow \Box(thread\_ready \rightarrow \Diamond thread\_run))$. This states that whenever a thread is created, it is *always* the case that when it becomes ready, it *eventually* runs.

P2. $\Box(kernel\_enter \rightarrow \Diamond kernel\_exit)$. This property states that it is *always* the case that when a thread enters into a kernel call, it *eventually* exits.

P3. $\Box(int\_enter \rightarrow \Diamond int\_exit) \rightarrow \Box(thread\_ready \rightarrow \Diamond thread\_run)$. This property states that if the processing of an interrupt starts and eventually completes, then the corresponding thread gets ready and eventually runs.

P4. $\Box(msg\_send \rightarrow \Diamond(msg\_receive \wedge msg\_reply))$. This property verifies that whenever a message is sent, it is eventually received and acknowledged by the receiving thread.

P5. $\Box(thread\_block \rightarrow \Diamond(thread\_ready \wedge thread\_run))$. This property verifies that whenever a thread is blocked, it eventually becomes ready and runs.

P6. $\Box(int\_enter \rightarrow \Box((handler\_enter \rightarrow \Diamond handler\_exit) \rightarrow \Diamond int\_exit))$. This property checks that whenever the system enters the interrupt state, an interrupt handler is invoked to service the interrupt. The handler eventually exits and the system also eventually exits from the interrupt state.

We monitored the properties using a computer that runs Ubuntu 16.04 on an Intel Core i3 processor at 2.10GHz with 8GB of memory. We focused the experiments only on the *cruise-control* application running on the boards.

Table III presents the verdict of each property on the input execution trace. The verdict for P1, P2 and P5 is presumably false ($\perp_P$), meaning that the properties were presumably not satisfied by the execution. We believe one of the reasons for these violations is the radiation that caused errors in the entire system. The final verdict for properties P4 and P6 is presumably true ($\top_P$), showing that the properties were presumably satisfied by the application. Finally, property P3 is evaluated to an *unknown* verdict. In this case, the verdict is unknown due to the lossy trace not satisfying the requirements [14] for a conclusive verdict for the property.

TABLE III
MONITORING VERDICTS

P1	$\perp_P$	P2	$\perp_P$
P3	?	P4	$\top_P$
P5	$\perp_P$	P6	$\top_P$

Figure 3 shows the monitoring overhead for ten runs of each property. We observe that the monitoring overhead is linear with respect to the trace size. These results confirm that instead of monitoring full execution traces, one can soundly monitor LTL properties on lossy traces with reduced overhead.

V. Discussion and Lessons Learned

A. Cleanup of Dataset

The data collected from the four Xilinx development boards consisted of the *tracelogger* logs. The boards under test would

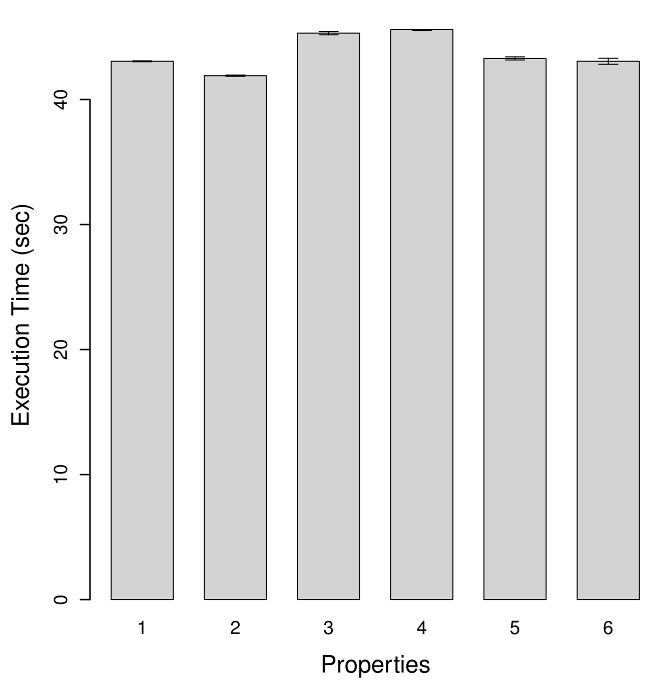

Fig. 3. Monitoring Overhead

crash randomly in the presence of the neutron beam. Hence, for every board we marked the time when the board crashed and required a reboot. The logs are split up by their crash times so that logs until a crash occurred are in a single data file. The total number of logs (total number of crashes) for each board are listed in Table IV. The number of logs varies for each board as it depends on the time during which the board was powered down during the experiment or was crashing more frequently resulting in more time spent booting.

TABLE IV
TOTAL NUMBER OF LOGS FROM THE SOCS

Board	Number of Logs
Xilinx ZC702 (1)	989
Xilinx ZC702 (2)	631
Xilinx ZC706 (1)	1324
Xilinx ZC706 (2)	485

The standard trace generated by the *tracelogger* utility in QNX has the following fields as shown in Table V.

TABLE V
VARIOUS FIELDS IN A QNX TRACE

Fields		
time	inkernal	msb
cpu	area	lsb
class	sigevent	d0
event	intnum	d1
pid	callnum	retval
tid	rcvid	priority
ppid	scoid	policy
name	sequence	strid
ip	numevents	str

The *tracelogger* utility produces files in a specific format with files having **.kev* extension. We transformed the files to an easily readable **.csv* format.

Since each of the trace event is associated with various experimental parameters such as the beam intensity, the number of controllers, neutron counts etc. The time on the board resets after every crash and hence the *time* logged by the operating system is not the actual time and it is difficult to correlate the trace events with the neutron flux and other experimental parameters. Therefore, we added another column to the **.csv* file with the actual time that enabled us to get accurate experimental parameters associated with each event in the trace. The application also has the option to induce random software faults but we think that in this experiment it is more important to evaluate the effect of the neutron beam on the software operation. We added five fields to the **.csv* file, as shown in Table VI, for completeness and to enable easier utilization of the dataset.

TABLE VI
AUGMENTED FIELDS AND THEIR DESCRIPTION

Augmented Fields	Description
clock_time	Actual time
controllers	No. of controllers in the application
software_faults	Randomly induced simulated faults (TRUE/FALSE)
flux	neutron flux data (in $neutron/cm^2/s$)
last_neutron_count	total number of neutrons emitted cumulatively

Every final version of a trace has a total of thirty two fields and there are a total of 107 traces which have simulated random software faults turned ON and the remaining 3,322 log files with it turned OFF.

Trace file names have been carefully kept where each file name is in the following format, *crashed.<TIMESTAMP>*. The *timestamp* descriptor in the file name is the time at which the crash on the board occurred or when the log file stopped growing in size.

We gathered from the experiment and data analysis that the cruise-control application never failed completely under the neutron beam. The failing of such an application means that there is no quorum between the instances of the compute engine. This reflects the reliability of the such an application under extreme test conditions. On the other hand, there may be other issues that might have gone unidentified.

VI. CONCLUSION

In this work, we created a dataset for analysis and post mortem analysis on the performance of real-time embedded software when the hardware is exposed to ionizing radiation. The experiment for generation of the dataset was conducted at LANSCE with a safety-critical, real-time application running on SoCs.

We also presented two use cases of the dataset. The first use case was of specification mining from these large scale system traces, and the second was of runtime monitoring where the traces can have missing events.

We believe that availability of such vast datasets in the domain of real-time embedded systems is useful for analyzing the robustness and reliability of these system in harsh environments. These datasets also enable researchers to develop better algorithms for anomaly detection, as finding benchmark datasets with real random anomalies is challenging. This dataset is a great example of data with random faults in the system.

ACKNOWLEDGMENT

The authors would like to thank Dr. Paolo Rech, Universidade Federal do Rio Grande do Sul for his kind support and enabling us to run our experiments at Los Alamos National Laboratory, NM, USA.

REFERENCES

[1] A. D. Pimentel, L. O. Hertzbetger, P. Lieverse, P. van der Wolf, and E. E. Deprettere, "Exploring embedded-systems architectures with artemis," *Computer*, vol. 34, no. 11, pp. 57–63, Nov 2001.

[2] M. Salem, M. Crowley, and S. Fischmeister, "Anomaly detection using inter-arrival curves for real-time systems," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016, pp. 97–106.

[3] G. Cutulenco, Y. Joshi, A. Narayan, and S. Fischmeister, "Mining timed regular expressions from system traces," in *Proceedings of the 5th International Workshop on Software Mining*, ser. SoftwareMining 2016. New York, NY, USA: ACM, 2016, pp. 3–10. [Online]. Available: http://doi.acm.org/10.1145/2975961.2975962

[4] T. Santini, P. Rech, L. Carro, and F. R. Wagner, "Exploiting cache conflicts to reduce radiation sensitivity of operating systems on embedded systems," in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Oct 2015, pp. 49–58.

[5] Y. Chen, "Cosmic ray effects on personal entertainment applications for smartphones," in *2013 IEEE Radiation Effects Data Workshop (REDW)*, July 2013, pp. 1–4.

[6] C. Dyer and P. Truscott, "Cosmic radiation effects on avionics," *Microprocessors and Microsystems*, vol. 22, no. 8, pp. 477 – 483, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933198001069

[7] *QNX Operating System: system architecture*. QNX Software Systems Ltd., 1997. [Online]. Available: https://books.google.ca/books?id=13oZAQAAIAAJ

[8] "Zynq-7000," https://www.xilinx.com/products/boards-and-kits/device-family/nav-zynq-7000.html, (Accessed on 12/30/2016).

[9] "Timed regular expressions," *J. ACM*, vol. 49, no. 2, pp. 172–206, Mar. 2002. [Online]. Available: http://doi.acm.org/10.1145/506147.506151

[10] A. Pnueli, "The Temporal Logic of Programs," in *Proceedings of Foundations of Computer Science*. IEEE, 1977.

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

[12] A. Bauer, M. Leucker, and C. Schallhart, "Runtime Verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 14:1–14:64, Sep. 2011.

[13] K. Havelund and G. Rosu, "Efficient monitoring of safety properties," *Int. J. Softw. Tools Technol. Transf.*, vol. 6, no. 2, pp. 158–173, 2004.

[14] Y. Joshi, G. M. Tchamgoue, and S. Fischmeister, "Runtime Verification of LTL on Lossy Traces," in *Proceedings of the 32nd Annual ACM Symposium on Applied Computing*, ser. SAC '17. ACM, 2017.

[15] S. Wang, A. Ayoub, O. Sokolsky, and I. Lee, "Runtime verification of traces under recording uncertainty," in *Proceedings of the Second International Conference on Runtime Verification*, ser. RV'11. Springer-Verlag, 2012, pp. 442–456.

[16] A. Bauer, M. Leucker, and C. Schallhart, "Comparing LTL Semantics for Runtime Verification," *J. Log. Comput.*, vol. 20, no. 3, pp. 651–674, 2010.