Title

Reasoning About Emergent Behaviors at Runtime

Author

Batista de Melo, Caio

Publication Date

2023

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA,
IRVINE

Reasoning About Emergent Behaviors at Runtime

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Caio Batista de Melo

Dissertation Committee:
Professor Nikil Dutt, Chair
Professor Fadi Kurdahi
Professor Tony Givargis

2023

DEDICATION

To my wife Xiaohui, my family, and my friends.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

My successful completion of this program was by no means an easy feat. I had a tough and long time making it through some steps, and I would not be able to get here without a lot of help from many amazing people. I want to thank some of them below:

Thanks to Nik for all his support, mentoring, and advice. You helped me so much to grow as a researcher, teacher, and person. You allowed me to focus on my interests, even if they didn't exactly align with a traditional program plan, and you helped open so many doors that I didn't even know about. I was extremely fortunate to have you as my advisor.

Thanks to Shannon for teaching me so much about teaching, for our random chats in the afternoon, and for helpful advice when something new came up in the classroom. I've learned a lot working with you in 31 and even more later.

Thanks to Jennifer for her guidance and for supporting my teaching endeavors.

Thanks to Fadi for all our fruitful discussions and his insightful feedback on my ideas.

Thanks to Genaína for her continuous mentoring even after I was out of UnB.

Thanks to Xiaohui for all her support and love, for listening to me complain about how my research isn't going anywhere at various times, for watching tons of movies, tons of TV shows, and two Billy Joel concerts *(so far)*, for comforting me at every defeat, for celebrating with me at every success, and for so much more ♡

Thanks to my family for everything! You've given me so much, allowed me to take so many chances, and supported me every step of the way – even in the ones you didn't want me to take. I won't mention any names, so there's no arguing about the order :)

Thanks to all my friends for their company and for bringing joy everywhere we met. Things always seemed a little better when we got together for a few laughs.

Thanks to the Donald Bren School of Information and Computer Sciences for all the support through fellowships and teaching appointments throughout the program. In particular, the teaching appointments which also guided the next steps of my career.

Thanks to the NSF for supporting the IPF project through the IPF grant CCF-1704859 and the SARE EAGER grant ECCS-2028782. My research would not have been possible without all the collaborations and discussions that arose from IPF.

Thanks to all my co-authors for their contributions, feedback, and ideas. The projects in this dissertation would not have been completed without your help.

Lastly, thanks to Victor, Everson, Marcos Rocha, Réver, Léo Silva, Arana, Allan, Zaracho, Dátolo, Guilherme, Bernard, Ronaldinho, Tardelli, Keno, and Hulk. Obri**GALO**!

VITA

Caio Batista de Melo

EDUCATION

Doctor of Philosophy in Computer Science **2023**
University of California, Irvine *Irvine, CA*

Masters of Science in Computer Science **2018**
Universidade de Brasília *Brasília, Brazil*

Bachelors of Science in Computer Science **2017**
Universidade de Brasília *Brasília, Brazil*

RESEARCH EXPERIENCE

Graduate Research Assistant **2018–2023**
University of California, Irvine *Irvine, California*

Graduate Research Assistant **2017–2018**
Universidade de Brasília *Brasília, Brazil*

Undergraduate Research Assistant **2016–2017**
Universidade de Brasília *Brasília, Brazil*

Undergraduate Research Assistant **2016**
Fordham University *Bronx, NY*

TEACHING EXPERIENCE

Associate Instructor **2022**
University of California, Irvine *Irvine, California*

Teaching Assistant **2018–2023**
University of California, Irvine *Irvine, California*

Teaching Assistant **2018**
Universidade de Brasília *Brasília, Brazil*

Undergraduate Tutor **2012–2016**
Universidade de Brasília *Brasília, Brazil*

REFEREED JOURNAL PUBLICATIONS

The Self-Aware Information Processing Factory Paradigm for Mixed-Critical Multiprocessing **Jul 2020**
IEEE Transactions on Emerging Topics in Computing

Characterization of Implied Scenarios as Families of Common Behavior **Dec 2019**
Journal of Systems and Software

REFEREED CONFERENCE PUBLICATIONS

Information Processing Factory 2.0-Self-awareness for Autonomous Collaborative Systems **Apr 2023**
Design, Automation and Test in Europe Conference

The Information Processing Factory: A Paradigm for Life Cycle Management of Dependable System **Oct 2019**
International Conference on Hardware/Software Codesign and System Synthesis

WATSHERE: A Watson Cognitive System to Navigate Social/Health Resources for Public Schools **Jul 2017**
Global Business and Technology Association Conference

BOOK CHAPTERS

Reflecting on Self-aware Systems-on-Chip **Jul 2020**
A Journey of Embedded and Cyber-Physical Systems

ABSTRACT OF THE DISSERTATION

Reasoning About Emergent Behaviors at Runtime

By

Caio Batista de Melo

Doctor of Philosophy in Computer Science

University of California, Irvine, 2023

Professor Nikil Dutt, Chair

As Autonomous Vehicles (AVs) become more common in the market, the development of new technologies and solutions for them advances quickly. However, there still are safety concerns regarding the increasing presence of AVs on public roads. For example, the California Department of Motor Vehicles reported an approximate average of 3 disengagement events for each AV under test in 2021 and 1 event for every 1,500 driven miles. Considering that a vehicle drives, on average, 15,000 miles per year, an AV would have ten such incidents annually. With this in mind, work needs to be done to enable AVs to handle unexpected situations (i.e., emergent behaviors) that can arise while driving, increasing the safety for both humans and AVs on the road. This dissertation discusses possible approaches to improve the detection and reasoning of emergent behaviors at runtime.

First, it describes SAFER, a framework to monitor individual systems using runtime verification alongside machine learning anomaly detection to detect emergent behaviors at runtime. SAFER achieves an average F1-score of 83% over four different example systems, showcasing its efficacy to be on par with related work while being applicable to different classes of applications. Next, it discusses DRIVE, a framework for monitoring collective systems that can work together to find global anomalies that affect collective safety by identifying violations in local properties. Throughout a state-of-the-art inspired truck platooning case

study, DRIVE can detect all safety property violations, with most queries resolved under 1 ms and a maximum latency of 11 ms per query, demonstrating its promise. Lastly, it presents LOCoCAT, a low-overhead framework for emergent behavior reasoning to classify anomaly types based on vehicular data. LOCoCAT achieves an F1-score of up to 99.16% within the first 50 ms of the anomaly, allowing the system to react quickly. These three techniques can be combined to allow future AVs to operate more safely on the road.

Chapter 1

Introduction

Emergent (mis)behavior often means complex systems operate in unexpected ways that are not easily predictable from the behavior of components [59]. This happens not only in components at different levels of abstraction, such as hardware, software, operating systems, or computer networking, but also in fully-integrated computer systems, as well as unforeseen interactions between them. For example, protocols/controllers such as SCSI provide various techniques optimized for stability in the operation of a large number of physical drives in datacenters. However, the vibrations generated by adjacent drives adversely affect the stability of the entire system [81]. In another example, well-known issues such as priority inversion in operating systems can result in the blocking of higher-priority tasks because they require resources held by low-priority tasks, which can also be considered an emergent behavior [33].

These problems are exacerbated when computer systems are integrated into Cyber-Physical Systems (CPS) that sense environmental signals, perform computations, and control for actuation in the physical world. Furthermore, CPS are often systems-of-systems, which result in a complex interplay of individual system behaviors that open the door for many

yet-unseen emergent behaviors. As such, emergent behavior is almost impossible to detect by checking only the fragmented side of the system, and this requires a holistic system-level analysis.

A CPS is a non-terminating system that continuously reacts to external inputs and usually operates within stringent safety and security constraints. With increasing complexity, an executing CPS can – maliciously or inadvertently – steer system behavior in unanticipated ways through emergent behaviors that have the potential to affect lives and compromise large investments or critical assets. Emergent behaviors could result from several factors [27], such as incomplete verification, inadequate modeling, hardware or software failures, or by malicious agents (i.e., an adversary or an insider) and attacks that can compromise the safety and integrity of these systems.

The CPS must address not only the issues of emergent behaviors of computer systems but also external influences, such as sensors and actuators, that affect the system. The bigger problem is that CPS systems – such as autonomous vehicles and medical devices – have an overall impact on human life and safety. Besides insufficient system-level analysis, malicious attacks also contribute to the emergence of emergent behaviors such as cyber-physical (e.g., side-channel) attacks, which can often cause irreversible damage in system execution.

Since emergent behavior manifests through a complex execution profile of the system, formal verification techniques such as model checking and runtime verification (RV) can be deployed to check system correctness. In model checking, a complete model allows one to take into account any location of the trace. On the other hand, RV – especially when dealing with on-line monitoring – takes into account finite executions of increasing size. For this, the monitor must be designed to take into account the execution incrementally. RV can respond nicely to predefined and predictable system states but may pose difficulty in detecting emergent behaviors.

Anomaly detection (AD) techniques can find an anomaly or outlier that is significantly different from the remaining data and thus flag anomalous execution in computer systems. For example, network intrusion detection, credit card fraud detection, sensor network error detection, medical diagnostics, and many other fields are well-known areas utilizing the AD technique [1]. AD can be particularly useful at runtime to detect an execution out of the normal range, and thus can play a significant role in detecting emergent behavior. However, since the important system state is not preserved and is only determined based on stateless input values, further optimization and backtracking of the system state are almost impossible by using AD alone.

As such, runtime verification and AD are useful in their respective domains but previously have not been used in combination effectively to detect emergent behaviors. In this dissertation, I first propose SAFER: a novel methodology that complements runtime verification, anomaly detection, and system recovery resulting in successful emergent behavior detection and resolution. The SAFER methodology is illustrated through four case studies, including a life/safety-critical example – a collision avoidance implementation of autonomous vehicles.

However, RV for autonomous distributed embedded systems (particularly in the automotive context) becomes even more challenging due to the need to support failover, resource sharing, and data-centric design flows. First, each node operates independently and has its own trace. Second, it is difficult to have a centralized monitor, and thus traces may have an undefined global state. Third, it is hard to express the properties of such distributed autonomous systems in existing formal languages. Fourth, many emerging systems can self-reconfigure and self-organize at runtime, increasing the difficulty in defining an expected global behavior. Lastly, many of these systems use AI and ML algorithms (e.g., object detection in autonomous driving pipelines [52]) which can generate unpredictable behavior at runtime. Besides, as systems complexity increases, so does the challenge of performing runtime verification at the architecture level. All of these issues point to a critical gap in the

applicability of RV to distributed self-reconfigurable autonomous cyber-physical systems.

Many types of self-reconfigurable autonomous CPSs have emerged, e.g., smart traffic intersections [42, 78] and truck platooning [43]. These systems are made from smaller individual ones and need to quickly reconfigure as their structure changes with new participants joining and old ones leaving. For the rest of the dissertation, we will focus on truck platooning as the exemplar of such a distributed autonomous system that requires runtime self-configuration [43].

Truck platooning allows multiple trucks to drive cooperatively to improve, e.g., safety [43] and fuel consumption [88]. To this end, truck platooning must guarantee safe collective behavior in addition to individual vehicle safety in dynamic settings, which intrinsically requires runtime verification. However, prior work in RV has mostly focused on verifying individual systems or statically configured distributed systems, with a gap in the literature for monitoring distributed systems that can self-reconfigure at runtime. Additionally, some of the work that has studied RV applied to distributed systems cannot handle self-reconfiguration (e.g., [60, 84]) or consider changes as the system degrades and treat them as slow and more permanent (e.g., [56]). Thus, there is a need to apply RV for distributed autonomous CPSs that can quickly self-reconfigure at runtime in emerging use cases for safe, collective autonomy, as exemplified by truck platooning. Furthermore, distributed RV critically requires efficient mechanisms to maintain global state and parallel execution traces, enable non-intrusive monitoring, and perform spatio-temporal queries to check safety properties.

To address these challenges, this dissertation then presents DRIVE, a novel methodology that combines Runtime Verification with distributed data sensing and storage, to overcome the challenges of coordinating distributed runtime verification that must verify overall system safety properties by combining the local traces and data distributed across multiple parallel embedded platforms. We validate our approach through a case study using a state-of-the-art inspired truck platooning scenario, where random events are inserted to disturb the steady-

state platooning of the observed vehicles.

The combination of the DRIVE and SAFER frameworks allow both individual and collective CPSs to monitor their runtime execution and detect emergent behaviors. However, as the literature shows that similar emergent behaviors can be appropriately resolved with the same reaction [22], it is essential for systems to understand the type of emergent behaviors they face in order to choose the best reaction. Thus, the last contribution of this dissertation is LOCoCAT, a framework that can classify particular types of vehicular emergent behavior allowing systems to understand the threats they face. We validate the feasibility of LOCoCAT using a low-power resource-constrained platform and compare its results with the few related works that analyze similar concerns using three widely-explored datasets.

The rest of this dissertation is structured as follows: Chapter 2 introduces background concepts and discusses related work for the proposed frameworks; Chapter 3 depicts the SAFER methodology for single systems monitoring and recovery and evaluates it using multiple case studies, including an autonomous driving example; Chapter 4 details the DRIVE methodology for collective system monitoring using a real-world-inspired case study to show its applicability; Chapter 5 discusses the LOCoCAT framework for characterizing detected emergent behaviors; and, lastly, Chapter 6 summarizes all contributions presented in this dissertation and suggests paths for future work.

Chapter 2

Background

2.1 Correctness vs. Normalness

According to [27], any system software can be represented with a Finite State Machine (FSM). The FSM used to represent the software shows the input and output needed for each state of the program. Also, the transition between the states is illustrated with the FSM. Besides all the expected states of each software, some meaningless weird states can occur unintentionally. Once one of these states has been reached, a new computing device, also named a weird machine, is needed to sneak away from the unexpected states and return to the predictable ones.

Moreover, the states occurring for a system's FSM are divided into two categories, predicted states (behavior) and unpredicted ones [27]. In both categories, having both desired and undesired behaviors are probable. The predicted desired behaviors are the ones included in the system's design. The predicted undesired behaviors are known to the designer as existing problems. The surprisingly pleasant behaviors are unpredictable manners of the designed system. It is worth understanding why the system is acting in a way that it was

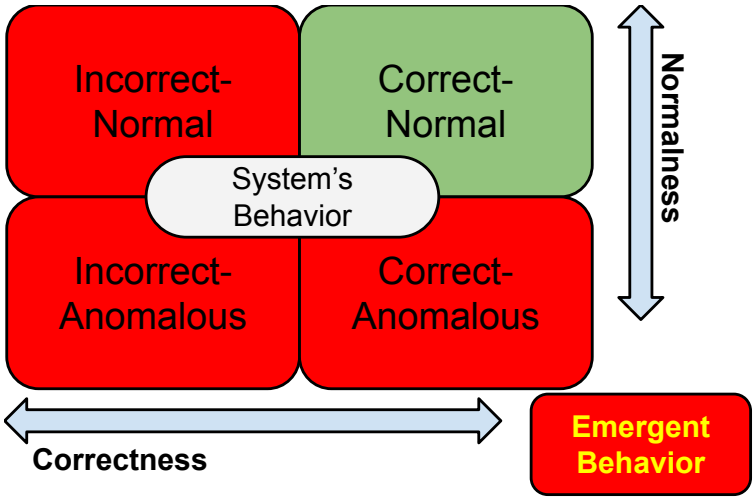


Figure 2.1: Classifying a Cyber-Physical System's Behavior.

not designed. The last category is the unpredictable undesired behavior, which may cause severe problems for the system, and is also the source of emergent behaviors of the system.

Consequently, it is essential to distinguish between correct normal behavior and abnormal behavior of a system. This dissertation distinguishes between correct vs. incorrect and normal vs. anomalous behaviors. The correct/incorrect dichotomy is orthogonal to the normal/anomalous dichotomy, as shown in Figure 2.1. By correct, we mean behavior that satisfies system specifications, and by normal, we mean behavior that follows (in a statistical sense) frequently observed patterns of the system. Note that not every correct behavior is normal. Even if the system is implemented correctly (according to some specification), some correct behaviors can still be anomalous. Many computing systems, whether embedded or general-purpose, receive user inputs that can, maliciously or inadvertently, alter or steer system behavior in unanticipated ways, resulting in emergent behaviors.

2.2 Emergent Behavior

Emergent behavior refers to a system behavior that is unexpected and sometimes hard to manage. Based on [59], nearly all emergent behavior instances have some characteristics in common, like an amplification of seemingly minor behaviors. Researchers have identified emergent behaviors of a system as one of the reasons for system malfunctions.

Given the nature of cyber-physical systems, emergent behaviors in such systems can happen in both computer systems and the physical processes they control. As an illustrative example, consider the problem of a single-camera-based collision avoidance example shown in Figure 2.2. The physical laws of vehicles inform the design of a controller system, which is a finite state machine (FSM) with four states that monitors safe braking distance and either "Increase Speed" or "Decrease Speed" through the collision avoidance (CA) state. As described in [27], emergent behavior can be attributed to the implementation of the FSM and the controller system complexity (e.g., operating system, processor, memory). If unforeseen circumstances (e.g., environmental changes) yield an emergent state in the controller system (i.e., CameraCtrl in Figure 2.2), these emergent states will also manifest in the physical system (i.e., Camera and Stop in Figure 2.2). An important insight given by this example is that an attack on the physical system can expose latent states to emergent software states and vice versa. That is, physical laws allow for a vehicle to turn into the camera malfunction due to the weather or an unexpected "Stop" such as a crash or engine failure. Even though these were not taken into account in the FSM, they can still arise at runtime. In other words, physical systems are an integral part of unintended programming models that can play an enormous role in emergent execution.

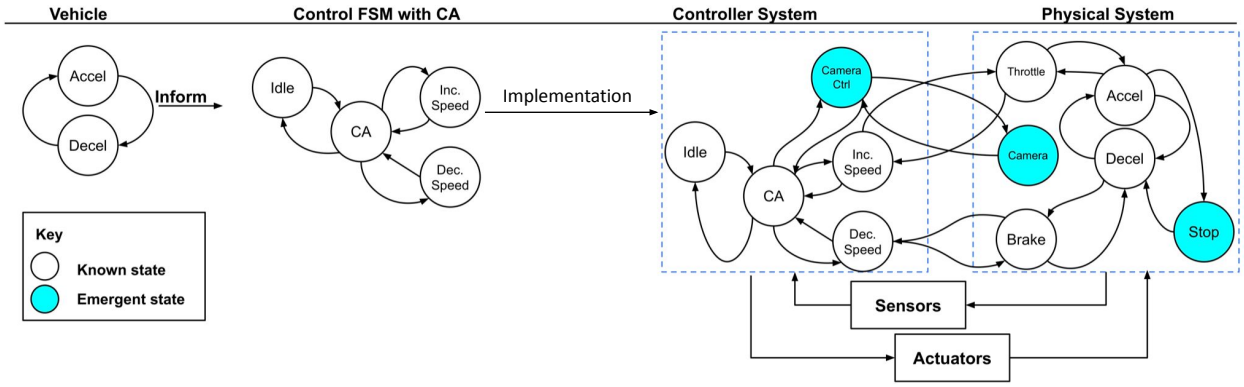


Figure 2.2: Emergent Cyber and Physical States in an Automotive Collision Avoidance Cyber-Physical System.

2.3 Anomaly Detection

Due to the problems caused by undesired emergent behaviors, it is essential to distinguish between anomalous behaviors and normal ones. A possible way to detect violations in system behavior (i.e., anomalous behaviors) is through anomaly detection. Anomaly detection techniques can be split into three categories [1]: (i) statistical-based methods, (ii) proximity-based methods, and (iii) deviation-based methods.

In statistical-based techniques (e.g., [54]), the input data will be represented as a statistical distribution. To determine whether a new datapoint is an anomaly, the model will try to fit the new data to that specific distribution. If the model can successfully fit the new data, it will be considered as normal. On the other hand, the model will detect data as abnormal.

In proximity-based methods, the model will try to find whether a datapoint is close to the majority of input data or not. If data is close to the majority, it will be considered normal; if not, it will be deemed abnormal. Clustering methods (e.g., [30]) are a well-known technique that is considered a proximity-based anomaly detection model.

For deviation-based anomaly detection, a method is first used for dimensionality reduction

like Principal Component Analysis (PCA) (e.g., [86]) or autoencoders (AE) (e.g., [72]). In the second step, these data with the reduced dimensionality will be reconstructed. Finally, the model will calculate the reconstruction error between the original data and its reconstructed one. This reconstruction error determines whether the data is an anomaly or not [2]. Based on the results shown in [72], by using dimensionality reduction with autoencoders (AEs), the model can detect more anomalies compared with the dimensionality reduction using PCA-based models.

All the aforementioned categories can be viewed as supervised or unsupervised learning algorithms, and there are some hybrid methods that are a mixture of these two types of algorithms [2] (e.g., [31]). Nonetheless, out of these three categories, deviation-based anomaly detection methods have recently received significant attention in the research community with promising results.

The PCA anomaly detection approach has been widely used for monitoring systems with highly correlated variables. As mentioned in [86], PCA can find the latent variables of the data with high dimensions. In other words, not only will it reduce the dimension of the input data, but also, it has the power to retain a large portion of the characteristics of the data. PCA will compress the input data to the latent variables, which will then be used to classify the test data into normal or abnormal behavior of the system.

Using an autoencoder as a non-linear way of dimensionality reduction shows its effectiveness in representing input data's hidden features. Hence, it has been widely considered in the anomaly detection problem. In [72], the dimensionality reduction of both real and artificial data is being made by autoencoders. The authors have claimed that by using AEs, the model could detect anomalies that were not seen by the PCA-based models. This idea motivated others to pursue doing more experiments on autoencoders. In more recent years, authors in [89] have utilized the effectiveness of deep autoencoders by considering the low possibility of having access to clean data. They have shown that the model is still capable of detecting

anomalies regardless of not having clean training data. Another novelty of the authors was using the idea of robust principal component analysis, which splits the data into two parts, the one that can be reconstructed by AEs and the anomalies and outliers existing in the real data.

The autoencoder-based anomaly detection methods have been improved using a more sophisticated neural network model called Variational AutoEncoders (VAE) [5]. The two advantages of using VAE are: (i) it reduces the dimensions in a probabilistically sound way; and, (ii) the definition of anomaly score is related to the probability measure rather than reconstruction error, which was used in PCA-based and AE-based anomaly detection. The proposed anomaly detection in [5] uses normal instances and applies the semi-supervised method using a probabilistic encoder and decoder. For testing the generated model, some samples are obtained from the probabilistic encoder, then fed to the probabilistic decoder to gain the required parameter. Then, the average probability of original data generated from the distribution is calculated as the reconstruction probability.

Motivated by successful applications of VAEs, as a proof of concept, we choose to use a VAE-based anomaly detection method in the SAFER implementation (Chapter 3). Thus, we will take a deeper look into the specifics of this method.

A variational autoencoder (VAE) is a directed probabilistic graphical model consisting of an autoencoder-like structure, and the model is approximated by a neural network[5]. In Figure 2.3, the VAE represents the complex model resembling a pair of an encoder and a decoder structure.

The VAE has an encoder $q_{\varnothing}(z|x)$ and a decoder $p_{\theta}(x|z)$ where z is the latent variable, which is the encoder output through sampling from $q_{\varnothing}(z|f(x,\varnothing))$ where f is a neural network representation. Also, the reconstruction x, which is reconstructed from z, is sampled from $p_{\theta}(x|g(z,\theta))$ where g is a neural network representation. In a reduced space (dimension) z,

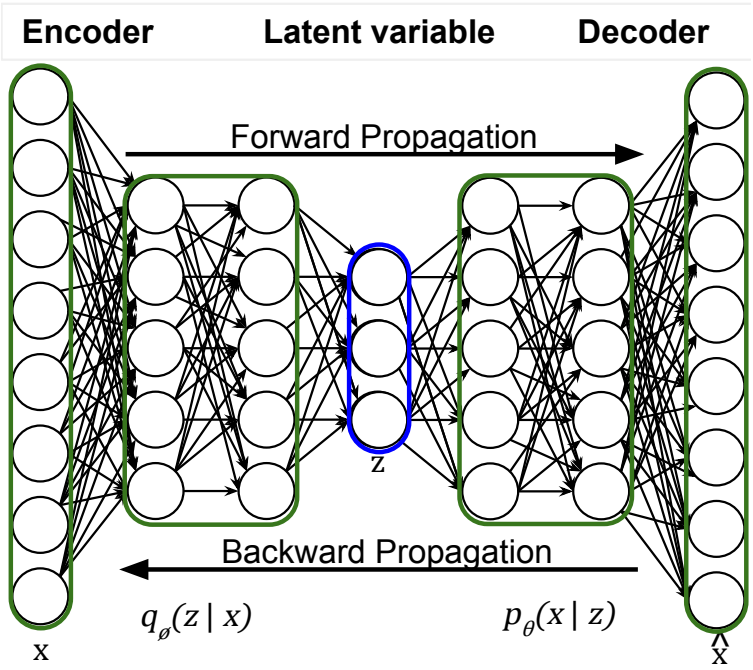


Figure 2.3: Variational autoencoder consisting of 1) an encoder, 2) a latent space, and 3) a decoder.

we can use a Multivariate Gaussian distribution function to analyze trace data that will be in continuous form.

Backpropagation updates $\varnothing$, θ by a deterministic transformation $h(\epsilon, x)$ where ϵ is from a standard normal distribution. The objective function of a VAE (the marginal likelihood) is:

$$\log p_\theta(x^{(i)}) = D_{KL}(q_\varphi(z|x)||p_\theta(z)) + L(\theta, \varphi; x^{(i)}) \tag{2.1}$$

where D_{KL} Kullback–Leibler divergence and L is the variational lower bound. The algorithm for training the VAE is shown in Algorithm 1. After this encoder/decoder pair is trained, the detection process consists of (1) encoding the original data x_o into x_e; then, (2) decoding x_e into x_d; and finally, (3) comparing the difference (i.e., reconstruction error) between x_o and x_d. If the difference between x_o and x_e is above a threshold, this datapoint is considered

anomalous.

Algorithm 1: Variational autoencoder training

Input: Trace Dataset $x^{(1)}, ..., x^{(n)}$
Output: encoder $f_{\varnothing}$, decoder g_{θ}

initialize $\varnothing, \theta$;
do
  for *i=1 to n* **do**
    draw L samples from $\epsilon \sim N(0, 1)$;
    $z^{(i,l)} = h_{\phi}(\epsilon^{(i)}, x^{(i)})$;
  end
  $E = \Sigma_{i=1}^{N} - D_{KL}(q_{\varnothing}(z|x^{(i)})||p_{\theta}(z)) + \frac{1}{N}\Sigma_{l=1}^{L}(\log\, p_{\theta}(x^{(i)}|z^{(i,l)}))$; // Equation (2.1)
  update $\varnothing, \theta$ using gradients of E;
while *not convergence of parameters* $\varnothing, \theta$;

2.3.1 Anomaly Classification

Anomalies can either be correct or incorrect behavior, as discussed in Section 2.1. When an anomaly is a correct behavior, it usually means a lack in the system specification [22], and including this behavior in the overall system spec should be enough to resolve the issue. However, when an anomaly is an incorrect behavior, the system needs to fix it. Even more so, often, an incorrect-behavior anomaly means a malicious agent is attacking the system.

Thus, anomaly classification is vital to systems as it allows for better reactions to ongoing events and prepares for future ones [80]. Anomaly (and attack) classification has been widely discussed in different fields, e.g., CPS [45], IoT [23], Automotive [77]. Additionally, it has been shown in the literature (e.g., [22]) that similar anomalies can be resolved with the same fix, allowing systems to manage multiple issues easier. However, as Chapter 5 discusses, there are some specific types of anomalies in automotive systems that have not been appropriately analyzed for a classification problem in a real-time system.

2.4 Formal Verification

Another possible manner to detect abnormal behaviors is through formal verification. Let us look at the difference between two typical formal verification techniques in (i) model checking and (ii) runtime verification (RV). In model checking – typically through offline analysis – all executions of a given trace are checked to investigate whether they satisfy the given accuracy attribute φ explicitly or implicitly when using symbolic techniques, causing the language inclusion problem.

Conversely, RV, mostly for online analysis, addresses a given word, requiring much less complexity than the inclusion problem [76, 55]. Model-checking, such as a symbolic model checking technique [19], considers infinite traces, so it understands the position of a trace. However, because RV handles a small window of finite traces, a monitor for RV must consider executions in an incremental fashion [14]. Given the focus of this dissertation in observing the runtime of systems, there is a need for online analysis; thus, we consider Runtime Verification for our proposed approaches.

Runtime verification can be used to detect that a property does not hold by constructing only part of the system model [32]. A popular way to achieve this is using the Linear Temporal Logic (LTL) language. LTL defines a formula as an infinite word of elements which can have operations applied to them. It is possible to state safety properties using such formulas. Furthermore, we can use data from the system to verify if these properties hold or were violated [32]. Multiple extensions and alternatives to LTL have been proposed.

Unlike LTL, regular expressions are more symmetric with respect to the arrow of time. The semantics of regular expressions are defined using the satisfaction relation $(w, t, t') \models \varphi$ which holds whenever the subsequence of w starting at t and ending in t satisfies expression φ [10].

Temporal logic is often good for describing *global behaviors* and the expected relations between the events and the states that evolve over time. In contrast, regular expressions are convenient for expressing *local temporal patterns* (consecutive sequences of events at states) that happen in a behavior [10]. The necessary expressiveness and succinctness of the specification language are only truly achieved when temporal logic is combined with regular expressions.

Signal Temporal Logic (STL) [53] is an extension of (propositional) LTL by moving from discrete to dense time and using predicates on numerical values in addition to basic (atomic) propositions. STL [53] extends the continuous-time Metric Temporal Logic (MTL) [47] with numerical predicates over real-valued variables. In particular, STL enables reasoning about real-time properties at interfaces that exhibit both discrete and continuous dynamics.

The pure time-domain nature of STL limits its applicability to cyber-physical systems. In particular, abnormal signal behaviors such as undesirable oscillations and complex topological requirements (i.e., the spatial distribution of the entities generating signals) are very challenging to capture using only the time domain. For example, the *time-frequency logic* (TFL) is a unified formalism to express the time-frequency properties of a signal. TFL extends STL with predicates evaluating the magnitude of a particular frequency range at a point in time. The semantics of TFL operates over a spectrogram generated using a short-time Fourier transform (STFT).

2.4.1 Spatial-Temporal Properties

Because of the limitations of timed formalism and the importance of spatial awareness to some domains, spatial-temporal properties are introduced. Thus, new extensions, languages, and methods that consider both time and space are developed. Particularly, multiple extensions to STL are proposed. Spatial-Temporal Logic (SpaTeL) [35] extends STL to reason

over quadtrees, spatial data structures that are constructed by recursively partitioning the space into uniform quadrants. Signal Spatio-Temporal Logic (SSTL) [17] extends STL with three spatial modalities `somewhere`, `everywhere` and `surround`, which can be nested arbitrarily with the original STL temporal operator. *Spatio-Temporal Reach and Escape Logic* (STREL) [8] extends STL with two novel spatial operators *reach* and *escape* from which is possible to derive other spatial modalities such as *everywhere*, *somewhere* and *surround*. These operators enable a monitoring procedure where the satisfaction of the property at each location depends only on the satisfaction of its neighbors, opening the way to future distributed online monitoring algorithms.

Additionally, different methods for monitoring such properties are introduced (e.g., [63, 9, 87, 79]). The authors of [63] use the spatio-temporal logic STREL and its expressivity to specify and monitor spatio-temporal behaviors over complex dynamical and spatially distributed systems. The design of a spatial logic is strictly related to the description of the space in which the dynamics take place. The monitor introduces Boolean and quantitative semantics: *Reach, Escape, Somewhere, Everywhere, Surround.*

2.4.2 Runtime Verification for Distributed Systems

The openness of CPS with the possibility for new actors to join or to leave the system, the local interactions among the system components, and the unknown environment in which they operate may cause undesired spatio-temporal emergent behaviors (i.e., congestion) often impossible to predict at the design-time. Indeed, their complexity restricts the exhaustive verification of their models' runtime only to relatively small examples. Many works (e.g., [11, 8, 9, 13, 61, 12, 49]) have tried to solve this problem.

Spatio-Temporal Reach and Escape Logic (STREL) [8] enables the specification of spatiotemporal requirements and their monitoring over the execution of mobile and spatially dis-

tributed CPS. STREL proposes both a qualitative and quantitative semantics based on constraint semirings, an algebraic structure suitable for constraint satisfaction and optimization. STREL is a flexible framework to formulate properties of CPS that allows freely mixing spatial and temporal operators to build complex queries, and to automatically construct monitoring algorithms.

Basin et al. propose a framework [12] that uses temporal structures (TSs) and a centralized specification language to provide monitoring across the system. The system uses a *slicer* to send the observed events to different submonitors, which can detect violations of global properties for that subset. Individual results of each submonitor are sent into a *merger*, which combines all violations in the system.

The authors extend their work by allowing the system to observe multiple data-sources [11]. Each data-source demands a separate slicer, which separates its data and sends it to the appropriate submonitor. Due to the time nature of TSs, it is necessary to introduce some synchronization so the submonitors can know the order of events coming from different slicers. This is accomplished by implementing a low-resolution global clock, which allows the different data-sources to watermark their data so they can be partially ordered for the submonitors. The final framework provides scalability by allowing the system to increase the number of data-sources, slicers, and submonitors as demand increases.

Similarly, Momtaz [60] proposes a framework that allows different components to synchronize their local clocks into a bound-global one. Their framework implements a rewriting-based method, where local timestamps are converted into bounded-global global ones, which allow for a partial ordering of the event traces. The bounded-clock synchronization allows the system to verify properties as a single cohesive entity.

Mostafa and Bonakdarpour [61] propose a sound and complete method for runtime verification of asynchronous distributed programs. Their proposed method uses the 3-valued

semantics of LTL defined over the global state of the program. Similar to [11], their proposal enables distributed computing of predicates through event slicing. The methodology uses distributed components that verify parts of the event data for violations and then determine their local state. The combination of all local states is used to determine the global state, which verifies global predicates.

Ye et al. [84] focus on self-adaptive systems. Their proposed framework models the external factors as a probabilistic model, allowing a single system to reason about all possible outcomes. On top of the probabilistic reasoning of the external environment, the system performs runtime verification in a formal way to enable the system to regulate itself.

Marmsoler and Petrovska [56] propose a methodology to perform runtime verification in dynamic architectures. Their framework monitors a single system for architectural erosion (i.e., modifications) throughout the system's lifetime. It observes architectural events that can impact the system negatively and uses architectural constraints so the system monitor can adapt its behavior as the system changes. By adapting the monitor to system changes, their proposal allows continuous runtime verification even as system properties change.

However, none of the literature has proposed a methodology that is able to verify distributed cyber-physical systems that can constantly self-reconfigure at runtime, e.g., trucks in a platoon that can transition from leader to follower.

2.5 Distributed Databases

A main premise of DRIVE (Chapter 4) is that safety properties of a distributed cyber-physical system can be checked at runtime as continuously running queries over the set of all event traces generated by its nodes. Such traces can be persisted in a database distributed across the entire system. A distributed database (DDB) is a collection of multiple logically

interrelated databases distributed over a computer network, and a distributed database management system (DDBMS) is a software system that manages a distributed database *while making the distribution transparent to the user* [29, Chapter 25].

The concept of transparency extends the general idea of hiding implementation details from end users. The following types of transparencies are possible:

- **Data organization transparency** (also known as distribution or network transparency) refers to freedom for the user from the operational details of the network and the placement of the data in the distributed system. It may be divided into:

  1. **Location transparency** refers to the fact that the command used to perform a task is independent of the location of the data and the location of the node where the command was issued.
  2. **Naming transparency** implies that once a name is associated with an object, the named objects can be accessed unambiguously without additional specification as to where the data is located.

- **Replication transparency**: Copies of the same data objects may be stored at multiple sites for better availability, performance, and reliability. Replication transparency makes the user unaware of the existence of these copies.

- **Fragmentation transparency**. Two types of fragmentation are possible:

  1. **Horizontal fragmentation** distributes a relation (table) into subrelations that are subsets of the tuples (rows) in the original relation.
  2. **Vertical fragmentation** distributes a relation into subrelations where each subrelation is defined by a subset of the columns of the original relation.

  A *global query* by the user must be transformed into several fragment queries. Fragmentation transparency makes the user unaware of the existence of fragments.

- Other transparencies include design transparency and execution transparency, referring to freedom from knowing how the distributed database is designed and where a transaction executes.

In the generic schema architecture of a pure DDBMS, a consistent, unified view, represented by the global conceptual schema (GCS), shows the logical structure of underlying data across all nodes, which provides *network transparency*. To accommodate potential heterogeneity in the DDB, each node has its own local internal schema (LIS) based on physical organization details at that particular site. The logical organization of data at each site is specified by the local conceptual schema (LCS). The GCS, LCS, and their underlying mappings provide the fragmentation and replication transparency.

The component architecture of a pure DDBMS is an extension of its centralized counterpart:

- The global query compiler references the GCS to verify and impose defined constraints.
- The global query optimizer references both GCS and LCS and generates optimized local queries from global queries. It evaluates all candidate strategies using a cost function that estimates cost based on response time (CPU, I/O, and network latencies) and estimated sizes of intermediate results. The optimizer then selects the candidate with the minimum cost for execution.
- The global transaction manager is responsible for coordinating the execution across multiple sites in conjunction with the local transaction manager at those sites.
- Each local DBMS would have its local query optimizer, transaction manager, and execution engines.

One of the most popular DDBMS available is BASE [68]. BASE consistency model is basically the subsumption of properties resulting from a system that provides availability

and partition tolerance but no strong consistency. While a strong consistency model is provided by ACID, which implies that all subsequent reads after a write yield the new, updated state for all clients and on all nodes of the system, this is weakened for BASE. Instead, the weak consistency of BASE comes up with an inconsistency window, a period in which the propagation of the update is not yet guaranteed.

- Basically available: The availability of the system, even in case of failures, represents a strong feature of the BASE model.
- Soft state: No strong consistency is provided, and clients must accept a stale state under certain circumstances.
- Eventually: consistent consistency is provided in a "best effort" manner, yielding a consistent state as soon as possible.

2.5.1 Distributed Query Processing

A DDBMS that supports full distribution, fragmentation, and replication transparency, must have a query decomposition module that breaks up or decompose a query into subqueries that can be executed at the individual sites. [29, Chapter 25] Additionally, a strategy for combining the results of the subqueries to form the query result must be generated. Whenever the DDBMS determines that an item referenced in the query is replicated, it must choose or materialize a particular replica during query execution.

To determine which replicas include the data items referenced in a query, the DDBMS refers to the fragmentation, replication, and distribution information stored in the DDBMS catalog. For vertical fragmentation, the attribute list for each fragment is kept in the catalog. For horizontal fragmentation, a condition sometimes called a guard, is kept for each fragment. This is basically a selection condition that specifies which tuples exist in the fragment (only

tuples that satisfy this condition are permitted to be stored in the fragment). For mixed fragments, both the attribute list and the guard condition are kept in the catalog.

A distributed database query is processed in the following stages:

1. **Query Mapping**. The input query on distributed data is specified formally using a query language. It is then translated into an algebraic query on global relations in the global conceptual schema.

2. **Localization**. This stage maps the distributed query on the global schema to separate queries on individual fragments using data distribution and replication information.

3. **Global Query Optimization**. A list of candidate queries can be obtained by permuting the ordering of operations within a fragment query. DDBMS query optimization algorithms consider the goal of reducing the *amount of data transfer* as an optimization criterion in choosing a distributed query execution strategy.

4. **Local Query Optimization**. This stage is common to all sites in the DDB. The techniques are similar to those used in centralized systems.

A semijoin operation $R \ltimes_{A=B} S$, where A and B are domain-compatible attributes of R and S, respectively, produces the same result as the relational algebra expression $\pi_R(R \bowtie_{A=B} S)$. Notice that the semijoin operation is not commutative; that is, $R \ltimes S \neq S \ltimes R$

In a distributed environment where R and S reside at different sites, the semijoin is typically implemented by first transferring $F = \pi_B(S)$ to the site where R resides and then joining F with R, thus leading to the strategy behind distributed query processing using the semijoin operation, which *reduces the number of tuples in a relation before transferring it to another site.*

Intuitively, the idea is to send the joining column of one relation R to the site where the other

relation S is located; this column is then joined with S. Following that, the join attributes, along with the attributes required in the result, are projected out and shipped back to the original site and joined with R. Hence, only the joining column of R is transferred in one direction, and a subset of S with no extraneous tuples or attributes is transferred in the other direction. If only a small fraction of the tuples in S participate in the join, this can be quite an efficient solution to minimizing data transfer.

2.6 System Recovery

When evaluating the efficiency of the existing methods in system recovery, four different aspects should be considered. (i) The standard operation overhead, which refers to the extra latency added to the system by running the system recovery method when failure will not happen. (ii) The extra latency added to the system in the occurrence of failure. (iii) The amount of similarity between the recovered state and the state before failure, and (iv)The resource efficiency of the system recovery method.

System recovery techniques can be divided into three different categories [51]: (i) Amnesia, (ii) Active Replication, and (iii) Rollback Recovery mechanism. In amnesia, a new copy of the failed process will start execution from the point of recovery. The low complexity of the method results in high efficiency as one of its benefits. On the other hand, the loss of data or state can occur, which will result in low recovery precision.

In the active replication technique, multiple copies of the tasks will be executed in the parallel systems. The main issue with this method is the trade-off between consistency and availability. Techniques used in the systems like Borealis [7], provide some flexibility in the system by configuring the consistency trade-off. In comparison to Borealis, mechanisms like Flux [75] will provide more relaxed synchronization methods. Although the active replication

technique suffers from low resource efficiency due to the parallel execution of the same tasks, these techniques have high recovery precision and low overhead.

Rollback recovery techniques are considered the third category of system recovery methods. In these techniques, the system will reset to some previous state and start execution from that state. The two strategies are (i) logging and replay and (ii) checkpointing. In the logging and replay strategy, just the system's changes will be logged, not the whole state. In the checkpointing method [16], the entire state (snapshot of the system) will be recorded within the predetermined time slots. The rollback strategy has been used in various systems, such as MillWheel [3], and Flink [21]. Although this strategy is good at recovery precision and resource efficiency, it suffers from high latency overhead.

Chapter 3

Monitoring a Single System

3.1 The SAFER Framework

This chapter will discuss the SAFER (Safety Assurances for Emergent Behavior) framework. SAFER allows users to model systems using state machines and automatically trains and deploys AD and RV models for runtime emergent behavior detection. Figure 3.1 depicts an overview of SAFER, which consists of (a) a design-time methodology and (b) runtime verification (online monitoring) using non-intrusive hardware or software instrumentation.

The design-time methodology (Figure 3.1 (a)) first develops a state-based model (1). The state-based model automatically implements the skeleton of the system (e.g., PSM code) and key functions (e.g., import/export) and includes instrumentation. It provides the user with complete source code, and all that is required is for the user to complete the user-defined functions. The compiled binary contains instrumentation-based annotations, and running it on a real machine produces one or more sets of stateful traces (2). The state-based model (1) performs formal model construction. This constructs a runtime verification model (3), which is platform-independent, as its output. The stateful trace (2) and the RV model (3)

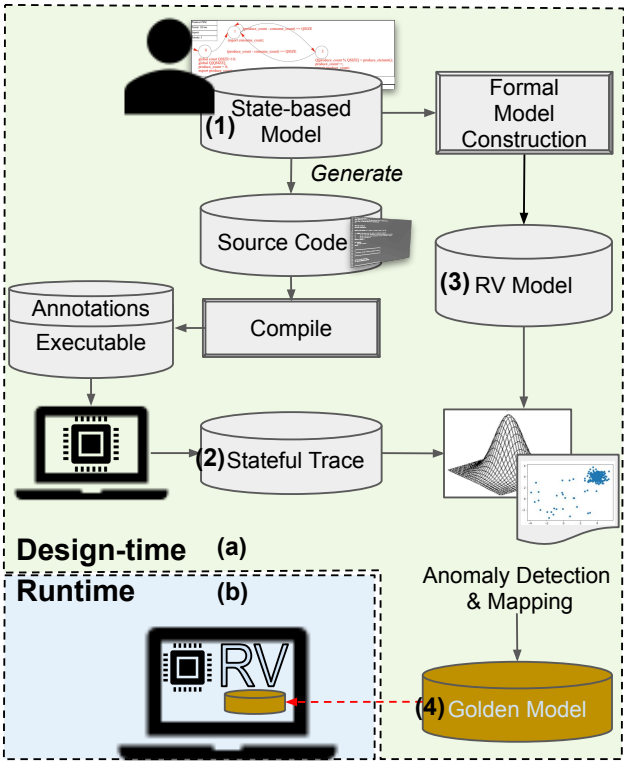


Figure 3.1: Overview of SAFER methodology for (a) design-time, and (b) runtime.

cross-reference each other to perform AD and model updates, respectively. This gives a golden model for runtime verification, including monitorable properties.

The runtime methodology (Figure 3.1 (b)) performs runtime verification based on the golden model. The monitor can be non-intrusive hardware-based such as TAL hardware [70], NUVA hardware [70], or intrusive middleware. Finally, we enable the system to trigger recovery techniques when an emergent behavior is detected through our analysis.

The remainder of this section details each step in the SAFER methodology.

3.1.1 Periodic State Machines

SAFER allows users to model systems using Periodic State Machines (PSMs). PSMs integrate the concepts of periodic finite state machines with explicit definitions of data com-

munication and synchronization between state machines in hardware and between hardware and software components. SAFER uses a conceptually simplified PSM model compared to [46]; the PSM model used in SAFER is defined below.

Definition 3.1. *A periodic state machine (PSM) is a tuple* $G =< P, S, S_0, \delta, IM, EX >$ *where:*

- P *is the period at which the state machine executes. The period* P *is defined as an explicit time unit, e.g., 200 ns.*
- S *is a finite and non-empty set of vertices, where each vertex represents a state. A state is fired at multiples of* P *time units.*
- S_0 *is an initial state, an element of* S*.*
- δ *is the state-transition function: where* Σ *is a transition condition.*
- IM *is a set of shared variables from other PSMs that are imported into this PSM.*
- EX *is a set of shared variables that are exported from this PSM that can be imported to other PSMs.*

Definition 3.2. *A state* S *in a PSM* G *is a tuple* $S =< V_i, A, V_e >$ *where:*

- V_i *is a shared variable that can be imported,* $V_i \in IM$*;*
- A *is a set of actions (statements) executed in state* S*;*
- V_e *is a shared variable that can be exported,* $V_e \in EX$*.*

3.1.2 Runtime Monitorable Properties

Runtime verification handles verification techniques that can check whether the execution of the system being monitored satisfies or violates given properties [74]. Based on the PSM

definition and architecture, SAFER checks the system at runtime for the following properties:

1. Only one state can be executed in a PSM.
2. Export statement queues value of the variable into the message queue.
3. Import statement waits until the message queue has a relevant Export variable.
4. Execution time in a state in a PSM cannot exceed time defined in period p.
5. Each state will evaluate transition condition(s) every period p.
6. Each state in different PSMs can be executed simultaneously.
7. Execution of statement(s) in a state must be done within period p.
8. A state S must be moved to one of state(s) in $\delta(S, *)$ after transition, where * = any.
9. Variable used in a PSM must be allocated locally and cannot be used outside of the PSM.

The above properties are divided into two categories: *safety properties* and *liveness properties*. A safety property (p1, p3, p4, p5, p7, p8, and p9) states that the attribute must be true for all paths for the system to work safely. A liveness property (p2, p3, and p5) ensure the progress of the workflow. Note that p3 and p5 exhibit both safety and liveness properties. As a proof of concept, SAFER uses a Python framework called pyModelChecking[1] to verify the RV properties outlined above. Additionally, SAFER uses timed linear-time temporal logic3 ($TLTL_3$) [14] to formally define these properties, which is a language able to verify timed conditions and supports three truth values: true, false, and inconclusive.

[1]https://github.com/albertocasagrande/pyModelChecking

3.1.3 Source Code Generator & Trace Analyzer

PSM models can generate skeleton code that satisfies the SAFER properties. The pthread library creates concurrent execution threads, one for each PSM, and each PSM is structured as a switch-case statement, where we consider what state it should execute and statements from the PSM model as is. Functions are generated with a boilerplate code that returns a random number and adds delays corresponding to the PSM's period. This is to (i) emulate the expected timing after the full implementation is included in those functions and (ii) ensure that all states have a fair chance to execute before the correct code is implemented.

Later, when the user implements each PSM's complete behavior, these delays and boilerplate code can be removed, and the generated models will still work if the expected timing was estimated correctly.

The skeleton code is instrumented so that each PSM exports its execution trace with its current state when that starts executing, as well as which state it is transitioning to after it is done. Traces are then generated using this code, which produces a sequence of time-stamped messages indicating which state each PSM is executing, and what is the next state and the reason that triggered the transition to this new state (i.e., the transition context). From these traces, we can extract a stateful trace that has information regarding the global state (i.e., the aggregated local current state of each PSM) and data from that state (e.g., how long it has executed).

Finally, the global trace is also parsed to extract a stateful trace that contains current global state, next global state, execution time for the current state, time taken to transition to the next state, and the context that led to the transition. The final stateful trace is a dataset where each message from the global trace is converted into a datapoint that has five sets of features: (1) the current global state, (2) the next global state, (3) the current state execution time, (4) the time to transition, (5) and the transition context (i.e., what system

conditions are triggering the next transition).

3.1.4 Anomaly Detection Runtime Model

After collecting traces from the correctly generated software, SAFER trains an anomaly detection model using the stateful trace as input for a variational autoencoder (VAE). We use Tensorflow 1.15.2 to create a VAE model based on the stateful data we extracted from the implementation model.

The goal of using the autoencoder is to reduce the dimensionality from the stateful data (which could vary based on the global state length) into five dimensions. Originally, each datapoint has $2N + 3$ features, where N is the number of PSMs of the specification: we consider the current global state, the next global state, the execution time of the current state, the time to transition to the next state, and the transition context.

Reducing the dimensionality of our data helps to combine relevant information in the latent space. The model then tries to decode the latent space variables back to the original number of dimensions. Later, the reconstruction probability of each state is used to determine if it is an abnormal state or not. The reconstruction probability represents how well the state can be recreated based on the values of the latent variables. If this probability is below a certain threshold, then the datapoint is considered abnormal.

Once the model is trained with the stateful traces, we embed it into the generated code with the frugally-deep library[2]. A new supervisor thread is created that can monitor the state of the other PSMs as a whole and use this embedded model to detect anomalies as the system executes.

[2]https://github.com/Dobiasd/frugally-deep

3.1.5 System Recovery

The final part of the SAFER infrastructure consists of recovering the system after a property is violated or an anomaly is detected. If the system is deployed with a recovery method enabled, the supervisor thread that runs the anomaly detection model will have access to all data from the system.

When using the Amnesia recovery method, the supervisor thread will reset all system data to its starting values when an anomaly is detected. If using the Checkpointing recovery method, the supervisor thread will make a copy of the current execution context when a normal state is detected; then, when it detects an anomaly, it will restore the stored system context from the last successful check.

Finally, it is important to note that, due to the way the system is instrumented from scratch, we assume that error-free sensing of the system state is possible. This assumption might not hold true in a real-world scenario. In such cases, it would be possible to incorporate existing literature that can estimate the system state [85], which can increase the robustness of the SAFER implementation. However, these exceptional situations are topics for future research.

3.2 Application Classes

To validate our framework's applicability, we conducted four case studies: Producer-Consumer, Integer Overflow, Stack-based Overflow, and Collision Avoidance. These four case studies highlight different aspects of the proposed methodology. The Producer-Consumer is a small example to demonstrate rate-based systems, i.e., systems with periodic behavior.

The Integer Overflow[3] and Stack-based Overflow[4] examples were chosen to represent real-world systems that suffer from emergent behaviors, and there are many dangerous weaknesses related to them, which represent systems that can have unexpected critical behaviors. Additionally, the integer overflow example showcases the need for anomaly detection on top of runtime verification.

Lastly, the Collision Avoidance is an event-driven system, i.e., its behavior is not periodic; instead, it changes based on specific events. It depicts a subsystem of an autonomous vehicle's collision avoidance logic, which represents a safety concern if the system misbehaves. Thus, with our goal of providing safety assurances in mind, we give more focus and conduct further experiments for the Collision Avoidance system due to the potentially catastrophic nature of its emergent behaviors.

3.3 Evaluation

We conducted five experiments for each case study, where in each instance, we corrupted one of the stateful trace features. That is, after training the model using the training data, we evaluated the performance of the model using test data in 5 adversarial conditions: (1) corrupted current global state, (2) corrupted next global state, (3) corrupted state execution time, (4) corrupted transition time, and (5) corrupted transition context. These corruptions could indicate problems in the system, such as an imminent hazard [70] or an attacker trying to exploit a vulnerability of the system. In both cases, it is vital to detect corruption in a timely manner so that the system can respond appropriately. To achieve this, we duplicated the test data and corrupted half of it accordingly.

Table 3.1 shows the average F1 score results we obtained for the test data on each experiment.

[3]https://cwe.mitre.org/data/definitions/190.html
[4]https://cwe.mitre.org/data/definitions/121.html

Table 3.1: F1 Scores for PSM Anomaly Detection Experiments

Case Study	Current State	Next State	Execution Time	Transition Time	Transition Context	Average
Producer Consumer	80.73%	84.35%	90.21%	90.21%	66.96%	82.49± 8.57%
Integer Overflow	76.61%	77.25%	96.47%	96.47%	80.45%	85.45± 9.09%
Stack-based Overflow	83.29%	77.67%	97.52%	97.52%	95.52%	90.70± 8.53%
Collision Avoidance	74.55%	75.18%	83.76%	80.73%	69.02%	76.65± 5.13%
Average	78.79± 3.20%	78.61± 3.45%	91.99± 5.51%	91.23± 6.68%	78.48± 12.13%	83.82± 9.47%

We chose the F1 score as the evaluating metric, as it takes both false positives and false negatives into account. By doing so, we consider both normal behaviors that were incorrectly deemed anomalous and anomalous behaviors that were not detected.

Each row on the table shows the F1 scores obtained for each case study, where each column represents the type of anomaly that was introduced. The last column shows the average F1 score observed for all anomaly types for each case study. Similarly, the bottom row shows the average F1 score for each anomaly type across all case studies, and the last column on the bottom row shows the average F1 score across all experiments.

For all average results, we included the average F1 score ± the standard deviation across the results, e.g., the average F1 score for the Producer-Consumer case study is 82.49%, and the standard deviation for the F1 scores across the different experiments for this case study is 8.57%.

Across all experiments, we observed an average F1 score of over 83% when considering the best threshold for each system. In addition, the average false positive rate across the four case studies was approximately 16%, and the false negative rate was approximately 18%.

Table 3.2: Comparison of Anomaly Detection F1 Scores in Other Domains

Approach	Dataset	Supervised or Unsupervised	F1 Score
SVM [31]	KDD99	Supervised	79.11%
SVM with PCA [31]	KDD99	Combination of Both	90.51%
Robust Deep AEs [89]	MNIST	Semi-supervised	$\sim 75\%$
Outlier Detection [36]	Real-time network traffic	Unsupervised	29.82%
SVM [36]	Real-time network traffic	Supervised	57.25%
CkNN [36]	Real-time network traffic	Supervised	42.99%
VAE [5]	MNIST	Semi-supervised	$\sim 40\%$
SAFER	Real-time execution trace	Semi-supervised	83.82%

These results show great promise for the applicability of the SAFER framework. The F1 scores show that this approach can detect the desired anomalous behaviors and not erroneously flag normal system execution. In particular, it is worth pointing out that the transition time was the most accurate measure overall. This makes sense since the transition time is generally short, so our approach should straightforwardly detect an unusually long transition.

Additionally, the execution time also proved to be a useful metric, performing very well in all studies. This can be related to the PSMs; since they have a defined period, a state's execution time should not change a lot, and it has an upper threshold. Thus, our anomaly detection model accurately detected the corrupted times that deviated from the expected system behavior.

Furthermore, the average false negative rate across all case studies when considering only Execution Time and Transition Time anomalies was under 5%. This result reinforces our hypothesis that, due to the time-sensitive nature of PSMs, such anomalies would be easily detected.

Lastly, the results obtained are encouraging when compared to existing anomaly detection techniques. Because – to the best of our knowledge – there is no existing work that is directly comparable to SAFER, we chose to compare anomaly detection works from other domains.

Table 3.3: Methodology overhead

Overhead	Amnesia				Checkpointing			
	Producer Consumer	Integer Overflow	Stack-based Overflow	Collision Avoidance	Producer Consumer	Integer Overflow	Stack-based Overflow	Collision Avoidance
binary size	2.498x	2.501x	2.498x	2.495x	2.501x	2.502x	2.501x	2.496x
- detection	99.99%	99.88%	99.99%	99.85%	99.86%	99.86%	99.86%	99.80%
- recovery	0.01%	0.12%	0.01%	0.15%	0.14%	0.14%	0.14%	0.20%
memory footprint	2.759x	2.759x	2.758x	2.752x	2.760	2.759x	2.760x	2.753x
- detection	99.94%	99.94%	99.94%	99.88%	99.91%	99.92%	99.90%	99.83%
- recovery	0.06%	0.06%	0.06%	0.12%	0.09%	0.08%	0.10%	0.17%
performance	1.090x	1.065x	1.154x	1.041x	1.107x	1.077x	1.183x	1.113x
- detection	98.77%	97.28%	94.84%	98.25%	97.24%	96.18%	92.58%	91.91%
- recovery	1.23%	2.72%	5.16%	1.75%	2.76%	3.82%	7.42%	8.09%

Table 3.2 shows that through the PSM modeling and stateful trace generation, SAFER is well equipped to detect anomalies in a wide range of applications. The only approaches that obtained comparable results [31, 89] were geared towards, and applied to a single dataset, whereas SAFER can demonstrably be applied to diverse systems and still maintain a high F1 score.

Thus, these results support our hypothesis that SAFER can achieve great anomaly detection results while providing extreme flexibility by allowing users to define their applications. Additionally, although SAFER is semi-supervised, the anomaly detection part of the framework is unsupervised since it does not require any user input after the user has modeled their custom application. In comparison, the existing work that provides unsupervised anomaly detection [36] achieves much lower results than our proposed approach.

3.4 SAFER Overhead

The SAFER methodology does incur overheads that need to be assessed in a cost-benefit analysis. Table 3.3 shows the overhead added by our approach in terms of binary size, memory footprint, and performance using the two different approaches of system recovery. For each of these metrics, we consider three different overheads: (i) the total overhead; (ii)

the overhead added only by the anomaly detection model; and (iii) the overhead added only by the system recovery technique deployed.

The binary size overhead for both recovery techniques is similar at around 2.5x across all case studies. However, it is interesting to notice that the Collision Avoidance system had a slightly lower overhead. Further investigation showed that the added binary size has around 2.5MB extra due to the AD runtime model and libraries it requires. Moreover, as the original system's binary size grows, this extra size will represent a smaller relative increase. Additionally, the systems with Checkpointing also had a slightly higher overhead, as in order to enable the system to save and restore checkpoints, it needs to have extra copies of the variables it wants to recover, whereas restarting will reset variables to their start value.

The memory footprint overhead is also similar between all case studies at around 2.7x. Since the most significant factor in increasing memory is the anomaly detection model, the smaller original system shows a higher overhead due to the size of the AD model being relatively bigger. Furthermore, the Checkpointing technique exhibits a higher overhead once again since it needs extra memory to store checkpoints.

Lastly, to compare performance overhead, we executed each binary for a total of 1,000,000 PSM transitions and measured how long each of them took to finish. For the performance overhead, there is a more considerable variance across the different case studies. However, the AD model inference still appears as the bottleneck, accounting for at least 91% of the added overhead. Although the recovery part accounted for only a small portion of the performance overhead, it is relevant to note that the Checkpointing method included a higher overhead than Amnesia. This makes sense since Checkpointing executes before it has to recover by taking snapshots of the system to prepare for a rollback, whereas Amnesia only executes when it has to recover the system.

In summary, while SAFER adds overheads to the final system in providing emergent behavior

detection and recovery at runtime, the designer can adjust the overheads to meet the requirements for each application. In particular, the fixed-size overhead of binary size and memory footprint due to the anomaly detection model embedding does not vary significantly between different case studies; and thus can be considered at design time. Additionally, SAFER's performance overhead (due to runtime anomaly detection inferencing) can be controlled by adjusting how often the supervisor thread should run the inference process; this enables a trade-off between performance and safety. Overall, SAFER provides excellent benefits in terms of safety due to emergent behavior detection and recovery while giving users ways to adjust the framework's overhead to an acceptable level for their use case.

Chapter 4

Monitoring Collective Systems

4.1 The DRIVE Framework

This chapter will introduce and discuss the DRIVE (Distributed Runtime Verification for Monitoring Safe Collective Autonomous Systems) framework. DRIVE aims to allow distributed collective systems to perform runtime verification of global properties through distributed data processing. Figure 4.1 depicts the DRIVE methodology, which is split into two layers. The application layer is in charge of overseeing the interaction between vehicles and recording distributed execution traces (e.g., real-world data sensor inputs, software state, vehicle state), local to a single vehicle. Thus, in the application layer, each vehicle stores data corresponding to its own sensing and actuation without knowledge of other vehicles in the platoon. The distributed runtime verification layer is in charge of combining the local data for each vehicle, which is distributed into *local datastores*, and providing mechanisms to perform *global queries* with local data to verify the system's safety properties. By doing so, the application layer enables the platoon to observe the global platoon behavior, even though this data is spread out across all vehicles. Figure 4.1 highlights the interactions between the

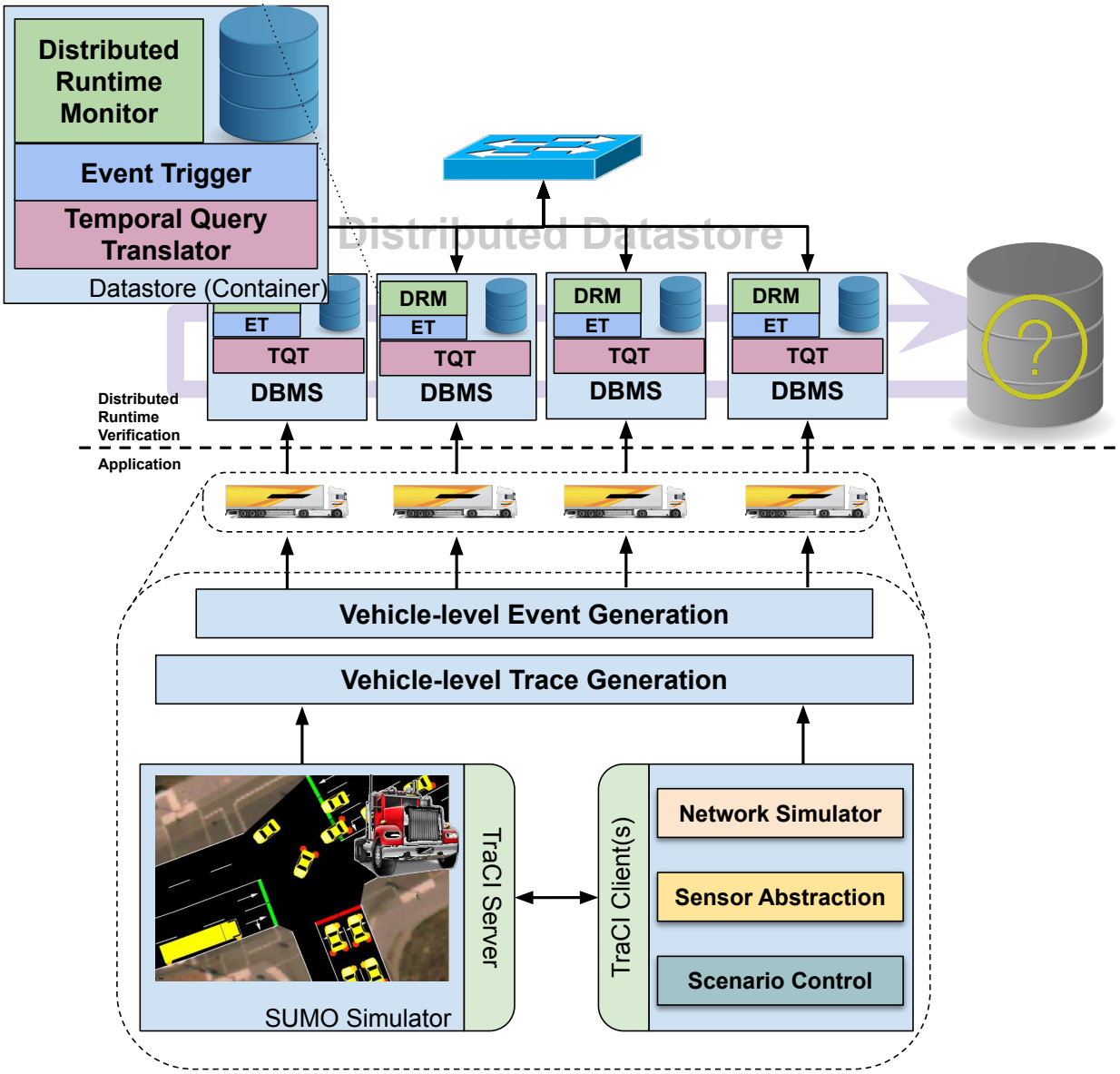


Figure 4.1: DRIVE Methodology Overview.

two layers and how the platoon-level data is distributed across the vehicles in our proposed DRIVE methodology. The remainder of this section will detail the implementation of each layer and how they can be used to achieve distributed runtime verification. Section 4.1.1 details the platform used to simulate the Application layer, and Section 4.1.2 introduces challenges faced and how they can be mitigated with our proposed Distributed Verification layer.

4.1.1 Application Platform

As a proof of concept, we use the SUMO simulator [50] in the application layer. SUMO is a traffic simulator that allows traffic systems modeling, enabling users to define diverse simulation scenarios by creating elements such as road networks and vehicles of different types. SUMO has a significant focus on the interactions between vehicles due to traffic circumstances. Therefore it has a more limited feature set in regards to graphical elements (e.g., object detection), which are supported through other simulators (e.g., CARLA [26]). However, this allows for faster simulation runs due to reduced resource utilization. As our experiments focus on the safety properties defined, SUMO proves to be a good fit with its quicker simulations and ability to observe, calculate, and export the data we require to verify that each vehicle meets its desired properties.

Additionally, SUMO provides an extension point called Traffic Control Interface (TraCI) [82], which allows for retrieving and injecting values during simulations. This interface is illustrated at the bottom of Figure 4.1. Through TraCI, SUMO provides extended features with existing plugins and allows developers to interact and modify the simulation during its execution. One such plugin is *simpla* [25], which enables vehicle platooning. Simpla allows users to define groups of vehicles that can form platoons to drive together. It does so by user-defined parameters that control the maximum gap between vehicles and the speed each

vehicle should follow based on its role in the platoon (e.g., leader, follower). However, we need to modify simpla with new features for our purposes; these additions will be described below with the other required SUMO modifications.

Since SUMO provides a global view of the simulation to all vehicles, they can observe all events without constraints. However, in a real-world situation, vehicles are limited by their sensors with physical limitations (e.g., sensing range). Thus, we extend SUMO using TraCI by adding simulated sensors to vehicles (i.e., Sensor Abstraction). These sensors enable limiting each vehicle's field of view (i.e., range, angle) to depict the sensors a vehicle could have available in a real-world situation (e.g., lidar, camera). At each time step of the simulation, we retrieve all the relevant information for each vehicle, modify the data to reflect what is observable based on the equipped sensors, and then send SUMO this calculated data back. Also, we modify specific properties (e.g., lane change available) based on this sensed data instead of the actual data measured by SUMO. By these means, we allow SUMO to simulate vehicle behavior based on data that depicts what would be available with real-world sensors (i.e., Scenario Control).

In addition to including sensors, we add two new features with TraCI that are not available with simpla regarding vehicle platooning. First, we enable vehicles to sense when a platoon is nearby in a different lane, which is not supported out-of-the-box – simpla only supports sensing platoons in the same lane. Second, after detecting a nearby platoon, we allow the vehicle to coordinate a merge in the middle of the platoon (i.e., Network Simulator) – simpla only supports joining at the front or back of the platoon. These two additions help improve simulations, as both actions are usually included in truck platoons [43].

Finally, in order to simulate the multiple local datastores, we use Docker [57] containers to deploy an individual DBMS instance to each vehicle. For each vehicle inserted into the simulation, the SUMO extension will create a new container connected to this vehicle. At each simulation step, vehicles log data concerning their sensed environment (e.g., open lanes,

sensed distances) and physical changes (e.g., speed changes) to their connected containers (i.e., Vehicle-level Trace generation), and when they exit the simulation, we export the data stored in their individual DBMS. Based on the exported data, we are able to extract vehicle-level Events (e.g., joined/exited a platoon, changed lane), which are used in the Runtime Verification layer to verify safety properties. Through this process, the entire platoon-level data is effectively distributed through the vehicles' datastores, similar to a traditional DDBMS horizontal fragmentation, as each vehicle stores only values pertaining to itself. Additionally, the combination of vehicle-level data allows new features to be queried at a platoon level (e.g., violations of defined properties), similar to a DDBMS vertical fragmentation.

4.1.2 Distributed Runtime Verification Platform

Distributed RV poses several challenges, such as specification, communication, security, higher rates of errors, non-determinism, and suitable specification languages.

First, it is a well-known fact that certain specifications cannot be appropriately verified at runtime. Due to the distributed clock, partial alignment for specific distributed events prevents the monitoring of time specifications that require a specific relative order of these events. Thus, since there is no global system view, there are far fewer specifications that can be monitored at runtime. Other works in RV (e.g., [60, 12, 11]) assume there is a low-resolution global clock that is able to order these events.

Second, the runtime verification performed by the monitor must be distributed and managed across multiple execution nodes. The decomposition and placement of property checkers are crucial decisions that actually affect the overhead that arises, such as the number and size of messages, communication delay, and computational complexity between monitors. Such placement also affects the admin domain where the event data is analyzed, compromises confidentiality restrictions, and can lead to security breaches due to the communication

required for the monitor to reach a verdict.

Third, the opposite aspect of security and confidentiality constraints translates into additional observability constraints that limit the specifications that can be monitored in practice. Distributed monitors may have to compete with traces where event data may be obfuscated or removed to maintain confidentiality, which affects the nature of the verdicts that may be provided.

Fourth, runtime verification must compete with errors that occur in distributed systems. Fault tolerance of each runtime monitor can be improved using techniques involving replication and dynamic reconfiguration of the monitor. More interestingly, fault tolerance monitoring algorithms can provide stability to the monitor. A theory that can determine the specifications combined with a monitoring algorithm can determine the warranties that need to be investigated.

Fifth, since monitoring has to be done in a distributed architecture, it essentially leads to non-deterministic calculations. Nonetheless, monitoring analysis and reported verdicts are characterized by solid consistency or observational verdict determinism and need to hide internal non-determinism. In practice, this cannot be easily achieved. Non-deterministic monitor behavior can also compromise the accuracy of RV settings and the validity of reported verdicts. To go around this issue, as mentioned above, some existing work relies on the assumption that there is a synchronous global clock available. (e.g., [60, 12, 11]).

Sixth, distributed systems impose additional restrictions on the class of properties that can be detected. Regarding the challenge of navigating a new specification language to monitor decentralized systems, it needs to be able to identify violations in a dynamic manner, as limits and properties might change as the system restructures itself.

In face of all these challenges, for the Distributed Verification layer, we propose the extension of the local datastores so they are split into *(i) Distributed Runtime Monitor, (ii) Event Trig-*

ger, and (iii) Temporal Query Translator as shown in Figure 4.1. These three components work together as follows:

Distributed Runtime Monitor (DRM). The monitor's primary role is to perform a given query specification and return a result of whether it was violated. Also, it lowers its overhead by using functions and stored procedures. These are compiled internally according to the DBMS implementation and shorten the execution time. Normally, timed monitor M returns True (i.e., satisfied) if observation O (Result of query statements) satisfies knowledge base K (Distributed Metadata) and property P (Property) shown in Figure 4.2. If O does not satisfy K and P, the monitor returns False to indicate that the property was violated or the check was inconclusive.

Event Trigger (ET). In case that a row related to a property is inserted/changed, the verification is performed using a database trigger, i.e., the database checks new data as it is inserted for a defined condition, and when the condition is satisfied, it triggers a second procedure. The trigger-based method uses fewer resources than the periodic method, which provides an opportunity to verify more properties. An event trigger is equivalent to an event emitter used in an RV system, where a validation monitor takes as input a series of events e from the event emitter (the system being monitored) and produces a result as an output.

Temporal Query Translator (TQT). It enables temporal queries using window-based SQL-99 queries and timestamped data and enables spatial queries by combining the user-defined queries with a knowledge of the system (e.g., mapping node names to physical entities). The temporal query translator can generate three different kinds of queries to support the event-based verification mentioned above: (i) a query statement creates a user-level procedure (i.e., a user-defined function to execute arbitrary code), (ii) a trigger statement creates an ET, and (iii) a monitor statement creates a DRM for the property. This process is exemplified in Figure 4.2. Additional examples of the conversion process from properties into SQL queries are provided later in Section 4.3.

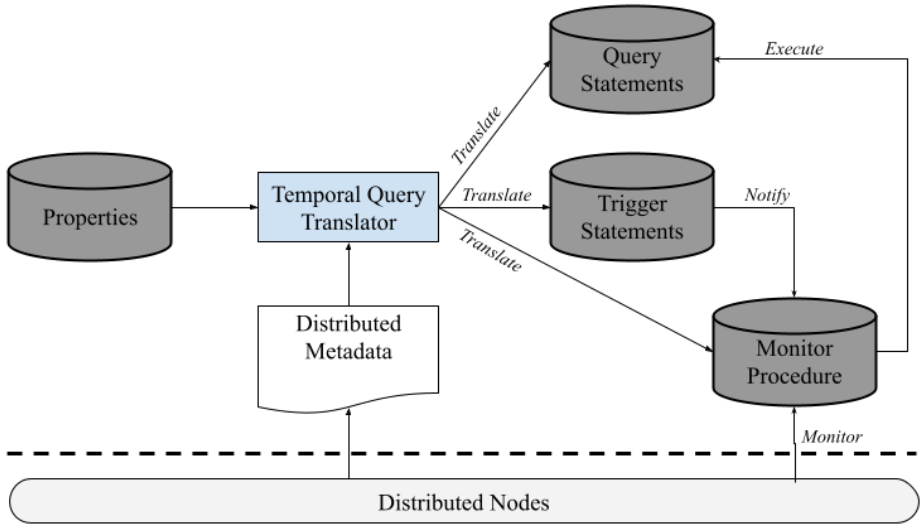


Figure 4.2: The process of generating each query statement using distributed metadata and temporal translator

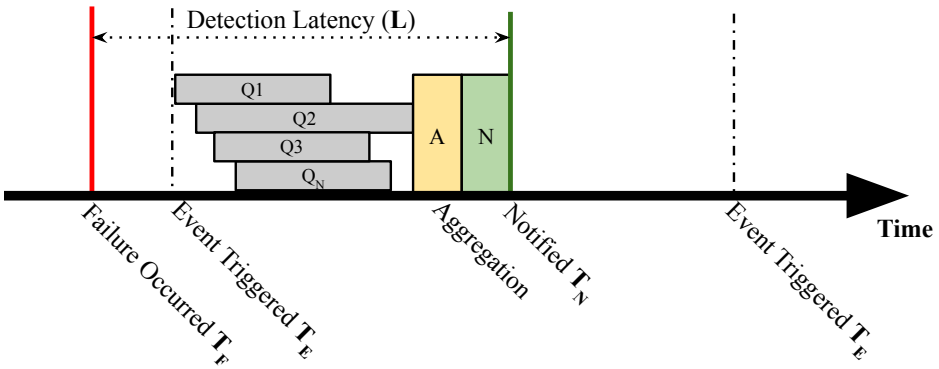


Figure 4.3: Detection Latency of a Distributed Query.

The query is executed for each distributed node, which runs in parallel ($Q_1 - Q_N$). When all query executions are finished, it goes through aggregation and reports whether the property is violated. The detection latency (L) is the time difference between the moment of actual failure (T_F) and the time at which the system detects a property violation (T_N) as shown in Figure 4.3. A distributed query is executed when data is inserted into a column registered in *Insert Trigger*. In RV, this latency plays an important role depending on the situation. In the case of a safe-critical system, the smaller this value is, the faster the action required for recovery can be taken.

Although the distributed queries across datastores serve a similar purpose to existing event-streaming frameworks, our proposal has a significant difference: it allows the system to reorganize itself. Popular event-streaming frameworks (e.g., Apache Kafka [1]) use a client-server model, where the server is deployed as a central point of processing power for all clients. DRIVE proposes a methodology that allows processing queries in-platoon without the need for a powerful remote resource available. This local processing allows for runtime self-organization as vehicles join and leave the platoon since any vehicle can act as the server that combines the global data. Such re-organization would not be possible with a resource-intensive server requirement, as all vehicles would need to have the whole server capabilities.

Additionally, existing event-streaming frameworks might not be applicable in the embedded context. Dunne et al. [28] show that popular event-streaming clients might not be able to perform well in an embedded context with limited resources. In fact, out of the tested frameworks, only two had clients that produced acceptable latency: Redis [2] and RabbitMQ [3]. However, Redis and RabbitMQ still require a centralized service that demands significant resources for processing all events, which hinders the deployment of the entire pipeline in-

[1] https://kafka.apache.org/
[2] https://redis.io/
[3] https://www.rabbitmq.com/

platoon as DRIVE does. An onboard RV system can detect events and trigger responses with minimal latency, which is critical for a real-time application. A client-server model, where the server is remote, cannot work reliably within these real-time constraints due to the largely unpredictable latency.

4.2 Case Study

In order to verify the applicability of DRIVE in a real-world scenario, we simulate a case study based on a popular truck route between Los Angeles and Seattle, shown in Figure 4.4. A platoon is formed with a given number (n) of trucks in a parking lot right before a highway entrance in Los Angeles. Out of the n trucks, n_a (front) trucks are destined to San Francisco (SF), while n_b (rear) trucks are destined to Seattle. New trucks join from behind (if going to Seattle) or in the front (if going to SF). The platoon splits into two platoons at the junction of I-5 (Seattle) and I-580 (SF). Sometimes rogue vehicles cuts-in through the platoon or slow down in front of it, requiring gap adaptations for safety. In such situations, emergency braking may be enforced to prevent an accident. Figure 4.4 shows the platoon's route, highlighting the key points where the platoon splits and arrives.

Additionally, we must note that we make the following assumptions for the experiments. First, we assume that there is a reliable and safe communication channel for the platoon. Second, we assume the latency of the communication channel is near-instantaneous. And third, we choose the leader truck as the platoon vehicle to aggregate the local results. Therefore, to verify the real-world applicability of DRIVE, a complete overhead analysis, including communication latency, would be required. First, a state-of-the-art communication method needs to be considered and factored in the results with, e.g., security threats, latency, and dropped packets [6]. Additionally, more work is needed to decide which platoon node should be responsible for aggregating the results. Since one of the goals of platooning is to conserve

Figure 4.4: Route Used for the Case Study (Map Taken from [65]).

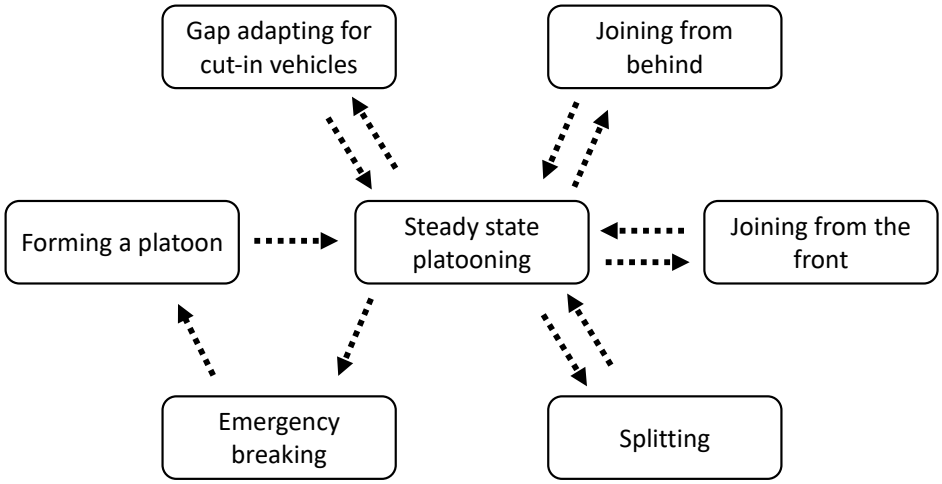


Figure 4.5: State Diagram of Platoon States

the followers' energy [43], removing the aggregation responsibility from the lead truck would make sense. Another factor in this decision, if the communication latency is considered, is how long the platoon is and how long it would take for the violation notification to reach all nodes. If the platoon is considerably large, it can make sense to have multiple aggregation nodes, resulting in a hierarchical structure. In that case, the verification problem would become much more complex, and a significant extension of DRIVE would be required.

4.2.1 System States and Safety Properties of Example Scenario

During this scenario, the platoon follows the state diagram shown in Figure 4.5. This state machine describes the possible states that the platoon can operate and which orderings could happen. Based on these possible states, the following safety properties are defined. Note that $TLTL_3$ [15] grammar is chosen to show properties in a formal language.

Additionally, in order to describe the scenario and the properties below, we use the following notation:

- $T[i]$: each truck ($1 \leq i \leq n$, $i = 1$: leading truck, $2 \leq i < n$: following trucks, $i = n$:

trailing truck)

- $v[i]$: each truck's velocity (measured by $T[i]$)
- $a[i]$: each truck's acceleration (measured by $T[i]$)
- $da[i]$: each truck's deceleration (measured by $T[i]$)
- $r[i]$: each truck's lateral offset to the lane center (measured by $T[i]$)
- $d[i][j]$ is the distance between $T[i]$ and $T[j]$
- $G[i][j]$: the time gap between $T[i]$ and $T[j]$, defined by $\frac{d[i][j]}{v[j]}$, with $i < j$.

Properties for steady-state platooning:

- **Property 1**. Time gaps between trucks should be within t_{min} and t_{max}; $(\forall i \in [1, n-1] \Box t_{min} \leq T(i, i+1) \leq t_{max})$
- **Property 2**. Platoon velocity v should be within v_{min} and v_{max}; $(\Box(v_{min}.i < v \wedge v_{max}.i > v\ ))$
- **Property 3**. Platoon acceleration a should be within a_{min} and a_{max}; $(\Box(a_{min}.i < a \wedge a_{max}.i > a\ ))$
- **Property 4**. Lateral offset difference between adjacent trucks should be within r_{min} and r_{max}. $(\Box(r_{min}.i < r \wedge r_{max}.i > r\ ))$

Properties for joining from behind:

- **Property 5**. Time gap between $T[n]$ and $T[new]$ should be within t_{min} and t_{max} for k seconds before the join is confirmed; $(\Box BackJoinRequest(new) \wedge t_{min} \leq T(n, new) \leq t_{max} \in [k, k] \rightarrow BackJoinConfirm(new)\ )$

- **Property 6**. Lateral offset difference between $T[n]$ and $T[new]$ should be within r_{min} and r_{max} for k seconds before the join is confirmed.

Properties for joining from the front:

- **Property 7**. Time gap between $T[new]$ and $T[1]$ should be within t_{min} and t_{max} for k seconds before the join is confirmed; $(\Box FrontJoinRequest(new) \wedge t_{min} \leq T(new, 1) \leq t_{max} \in [k, k] \rightarrow FrontJoinConfirm(new))$

Properties for splitting (between $T[i]$ and $T[i+1]$):

- **Property 8**. Time gap between $T[i]$ and $T[i+1]$ should be within ts_{min} and ts_{max} for k seconds before the split is confirmed and $T[i+1]$ immediately takes the role of a leading truck.

  $(\Box SplitRequest(i, i+1) \wedge ts_{min} \leq T(i, i+1) \leq ts_{max} \in [k, k] \rightarrow SplitConfirm(i, i+1))$

Since a platoon can self-reconfigure itself at runtime, these properties are dynamically instantiated with appropriate values to consider all vehicles that are a part of the platoon at the verification time. Additionally, we assume that a truck($T[i]$) can only measure the distance from its immediate follower ($d[i-1][i]$). If we need a distance over multiple trucks, it has to be collectively estimated by the trucks in between.

4.2.2 Scenario Implementation in SUMO Simulator

Figure 4.6 depicts an example of this scenario in SUMO with three trucks leaving from Los Angeles. Two trucks have San Francisco as their destination, while one is going to Seattle. Figure 4.6a shows this original platoon, with the leader marked with the color

green. Figure 4.6b shows a new truck joining in the front to go to San Francisco; the new truck is shown as yellow, which illustrates that it is not a part of the platoon yet. The old leader is shown as red, depicting that it is catching up with the solo platoon ahead of it. Figure 4.6c shows a new truck joining from behind to go to Seattle; the new truck is shown as red to illustrate that it is catching up with the platoon. Finally, Figure 4.6d shows the platoon splitting at the appropriate exit. The two last trucks take the exit heading to Seattle, while the front three trucks stay on the road heading to San Francisco.

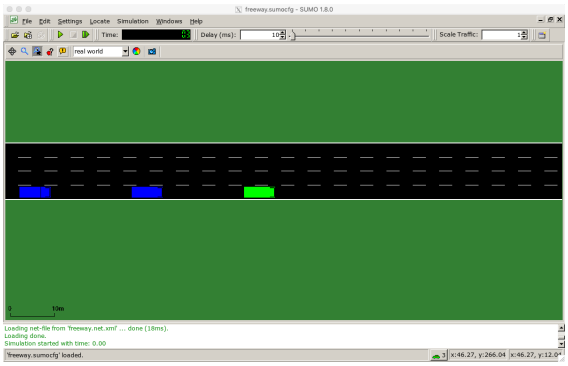
To validate the safety properties verification in these different states, we run this case study scenario multiple times in SUMO, and each run is unique due to the addition of random rogue vehicles to disturb the platoon. Two types of rogue vehicles are considered: (i) slow vehicles that cause an emergency braking situation (Figure 4.6e), and (ii) fast vehicles that cut in between the platoon to exit the highway (Figure 4.6f).

4.3 Experimental Results

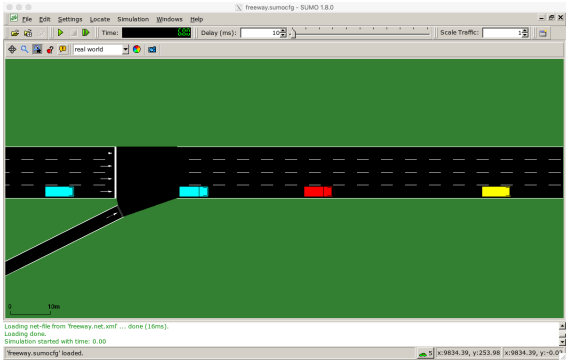
We run experiments using the scenario described in Section 4.2, with an initial platoon of three trucks and one truck joining on each side for a total of five trucks. To emulate heavy traffic conditions, we insert up to 40 random vehicles, with up to 20 simultaneously, during each simulation run. These vehicles are added to the simulation in random starting locations and speeds, but in the platoon's route so the chance of interaction is maximized.

The Runtime Verification (RV) is run in a non-intrusive manner. This means that the RV results are not fed back to the vehicles for appropriate measures, and they are used solely for verification purposes. Additionally, we consider imperfect sensing to the platoon vehicles, where they are limited by the physical limitations of their sensors.

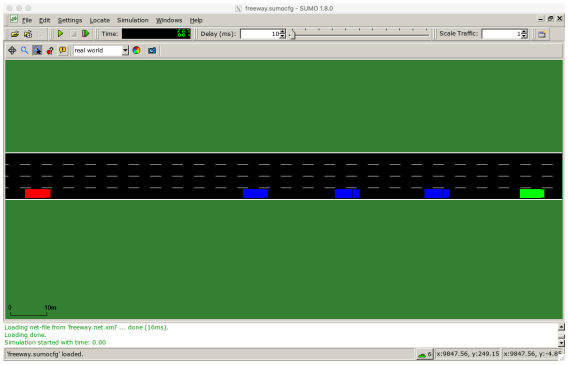
Section 4.3.1 will discuss the performance of the distributed queries and their effectiveness



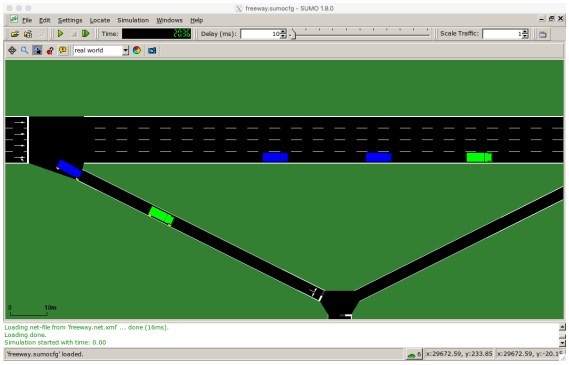
(a) Original three-truck platoon.



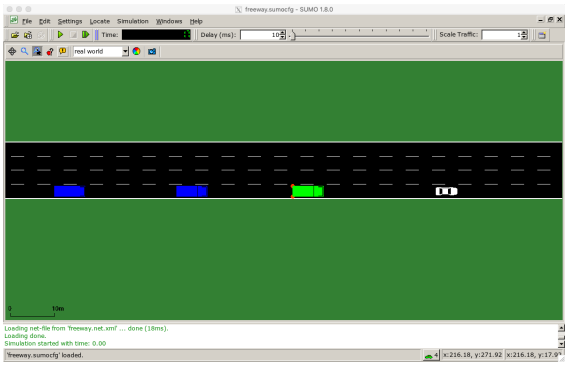
(b) New truck joins in the front to go to San Francisco.



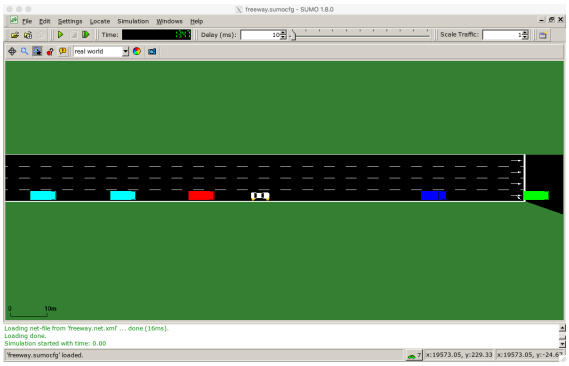
(c) New truck joins from behind to go to Seattle.



(d) platoon splits to the different destinations.



(e) Slow rogue vehicle ahead of the platoon.



(f) Rogue vehicle splits the platoon.

Figure 4.6: Scenario implementation in SUMO with three trucks departing Los Angeles.

in detecting safety property violations. Section 4.3.2 discusses promising preliminary results showing the benefits of the distributed verification for an example use case.

4.3.1 Distributed Query Performance

This subsection shows an example of how we handle properties and introduces the properties used in the experiment. Trigger, function, join status, detection rate, and latency are displayed as a result and analyzed. This experiment was performed using PostgreSQL version 13 as DBMS (acting as the datastores) and an Intel(R) Xeon(R) E-2176G CPU @ 3.70GHz. We will first look at a local property (i.e., a property that depends on data from a single truck) that does not require distributed queries (Property 3), and then extract and verify global properties that require distributed queries.

Local property: Deceleration of any truck in a platoon never exceeds A.
$\forall\Box(da[i] < A)$

We are interested in the occurrence of an event, i.e., when a new trace is added to each local table, and we create a trigger to handle the event. The queries processed by Temporal Query Translator are as follows.

```
CREATE TRIGGER loc_prop_trigger
    BEFORE INSERT OR UPDATE ON "speed_change"
    FOR EACH ROW
    EXECUTE PROCEDURE loc_prop_function();
```

Listing 4.1: Trigger to create an event

Listing 1 defines an event that checks the predicate in the user-defined function and calls the defined loc_prop_function() function when the property is violated.

```
CREATE FUNCTION loc_prop_function() RETURNS trigger AS
```

```
BEGIN
  IF (NEW.data->>deceleration > A) THEN
    Notify()
  END IF;
  RETURN NULL;
END;
```

Listing 4.2: A function to handle an event

Listing 2 defines an event that checks the predicate in the user-defined function and calls the defined Notify() function defined as follows in Listing 3.

```
CREATE FUNCTION notify(integer) RETURNS integer
     AS 'c_impl_notify', 'notify'
     LANGUAGE C STRICT;
    -- C native implementation notifies all nodes
```

Listing 4.3: A function written in native language to notify violation of property

The local property above has a detection latency of $< 1ms$. From this property, the extracted global properties related to the entire platooning are shown below. For properties 9 and 10, we introduce the detailed queries as follows.

Property 9. A deceleration event initiated by any truck X in the platoon never results (during the next T seconds) in an acceleration event of any truck Y in the platoon (behind or ahead of X).

$\forall X \Box DecelEvent_{X,t_0} \implies \nexists (AccelEvent_{Pred(X),t_1} \vee AccelEvent_{Succ(X),t_1}$ where $t_0 \leq t_1 \leq t_0 + T$.

```
-- for each distributed node
CREATE TEMPORARY TABLE tbl_prop_accel;
CREATE TEMPORARY TABLE tbl_prop_decel;

```

```
-- for each distributed node
CREATE TRIGGER prop_decel
    ON "speed_change"
    EXECUTE PROCEDURE prop_decel_function();

-- for each distributed node
CREATE FUNCTION prop_decel_function() RETURNS trigger AS
BEGIN
  IF (NEW.data->speed > cur_speed) THEN
    UPDATE prop_accel
  ELSE IF (NEW.data->speed < cur_speed) THEN
    UPDATE prop_decel
  END IF;
END;

-- function to perform global query (lead truck)
CREATE FUNCTION monitor ()
RETURNS void AS $$
    -- global query
    IF EXISTS (SELECT ((t1.isBraking OR ... OR tn.isBraking) AND (t1.
   isAccelerating OR ... OR tn.isAccelerating))
    -- distributed linked datastore
    FROM LINKED t1...tn
    ) THEN
        notify()
    END IF;
$$ LANGUAGE SQL;
```

Listing 4.4: Distributed query details for property 9

Listing 4.4 shows how Property 9 can be verified as a SQL query. First, each truck keeps track of its speed changes by (1) creating a temporary table to store changes on this property and (2) updating its property tracker when it accelerates. Then, this local data is combined

to verify the global property. The truck in charge of combining the results uses the *monitor* function to (3) verify the status of the local property of each truck. If the distributed query returns a result where a violation is detected (i.e., a truck decelerated and other trucks accelerated), (4) it notifies the platoon of this violation.

Property 10. A lane change is never initiated by the lead truck (more specifically, its driving software) unless there is enough gap G in the target lane per every other vehicle in the platoon.

$\exists TargetLane \Box LaneChangeStarted(TargetLane) \implies$

$\forall X \Box_T (LateralGap(X, TargetLane) > MinGap)$

```
-- for node t1 (leading truck)
CREATE TEMPORARY TABLE tbl_prop_lanechange

-- insert initial value
INSERT INTO tbl_prop_lanechange VALUES (0, true)

-- for node t1
CREATE TRIGGER prop_lanechange ON "lane_change"

-- for node t1
CREATE FUNCTION prop_lanechange_function() RETURNS trigger AS
BEGIN
  IF (NEW.data->>lane_changed) THEN
    UPDATE tbl_prop_lanechange SET isLaneChanged
  RETURN NULL;
END;

-- for all nodes but t1 (follower trucks)
CREATE TEMPORARY TABLE tbl_prop_distchange

-- insert initial value
```

```
22 INSERT INTO tbl_prop_distchange VALUES (0, true)
23
24 -- for all nodes but t1
25 CREATE TRIGGER prop_distchange ON "distance"
26
27 -- for all nodes but t1
28 CREATE FUNCTION prop_distchange_function() RETURNS trigger AS
29 BEGIN
30   IF (NEW.data->>lane_change_gap < G) THEN --G: Minimum gap for lane
     change
31     UPDATE tbl_prop_distchange SET isDistTooClose
32   END IF;
33 END;
34
35 -- function to perform global query
36 CREATE FUNCTION monitor ()
37     -- global query
38     IF EXISTS (SELECT (t1.isLaneChanged AND ( t2.isDistTooClose OR ... OR
    tn.isDistTooClose))
39     -- distributed node query
40     FROM LINKED t1...tn
41     ) THEN
42         notify()
43     END IF;
```

Listing 4.5: Distributed query details for property 10

Similarly, Listing 4.5 shows how Property 10 can be verified as a SQL query. First, the leading truck creates a new temporary table to store lane change actions, while the follower trucks create temporary tables to store the gaps for possible lane changes. Then, the truck in charge of combining the results uses the *monitor* function to combine the local data and verify the global property. It queries the distributed datastores to find an event where the

leading truck (T_1) had a lane-change action while a follower truck ($T_2 \dots T_n$) did not have enough space to change lanes. If there is such an event that satisfies the proposed violation, the truck notifies the rest of the platoon.

We believe the description of how these two properties can be verified is a good illustration of the applicability of our framework. The conversion from property to SQL is simple and allows for the distribution of complex queries to local nodes. Therefore, the authors decided to omit the implementation details for properties 11 15, as they would not add essential information in addition to the details provided for properties 9-10. Thus, only the definitions are presented for the remaining global properties.

Property 11. A lane change is never initiated by the lead truck unless all trucks have already been in the same lane for at least T seconds.

$\Box(\exists Lane_1, Lane_2, LaneChangeStarted(Lane_1, Lane_2) \implies \forall T \in Trucks(\Box_{t_{min}} InLane(T, Lane_1)))$

Property 12. A lane change initiated by the lead truck is followed by a lane change of every other truck in the platoon within T seconds.

$\Box(\exists Lane_1, Lane_2, LaneChangeStarted(Lane_1, Lane_2)$

$\implies \forall T \in Trucks(\Diamond_{t_{max}} LaneChangeComplete(Lane_1, Lane_2)))$

Property 13. A truck X spontaneously leaving the platoon (by switching lanes) always results in the gap ahead of all remaining vehicles narrowed down to the specified following distance (within tolerance).

$\Box(\exists X \in Trucks, LeavesPlatoonComplete(X) \implies (\Diamond_{t_{max}} TimeGap(Pred(X), Succ(X))))$

Table 4.1: Performance of Runtime Verification on Distributed datastore for local (1-8) and global (9-15) properties.

Property	Triggers	Functions	Tables	Detection %	Latency
p1	4	4	0	100%	<1 ms
p2	4	4	0	100%	<1 ms
p3	4	4	0	100%	<1 ms
p4	4	4	0	100%	<1 ms
p5	4	4	0	100%	<1 ms
p6	4	5	0	100%	<1 ms
p7	4	5	0	100%	<1 ms
p8	4	5	0	100%	<1 ms
p9	4	8	4	100%	<5 ms
p10	4	12	4	100%	<10 ms
p11	4	12	4	100%	<11ms
p12	4	8	4	100%	<5ms
p13	4	12	4	100%	<10 ms
p14	4	8	4	100%	<3 ms
p15	4	8	4	100%	<5 ms

Property 14. No truck X can leave the platoon while another Y is trying to join (i.e., after Y has signaled its intent to join but before it physically joins).

$\Box(\exists X \in TrucksJoinsPlatoonStart(X)$

$\implies (\forall Y \neq X \Box \neg LeavesPlatoonStart(Y) \mathcal{U} JoinsPlatoonComplete(X)))$

Property 15. A platoon X can only join another platoon Y, of size n, by adjoining its leader to the rear end of Y (no interleaving).

$\Box(MergedPlatoons(X,Y) \implies Succ(Y_n) = X_1 \wedge \forall T \in X, Succ(T) \in X \wedge \forall T \in Y, Pred(T) \in Y$

Table 4.1 shows the performance results for all queries executed, both for local and global properties. The detection latency across all queries has excellent results and demonstrates the applicability of DRIVE in the real world. Additionally, as we are able to verify all properties (i.e., detected 100% of violations for all properties), our framework enables the

platooning vehicles to verify their local properties by themselves and, after joining a platoon, they can contribute to the verification of the global behavior. By doing that, the platoon does not have the need to collect new information, only the sharing of the existing data is required.

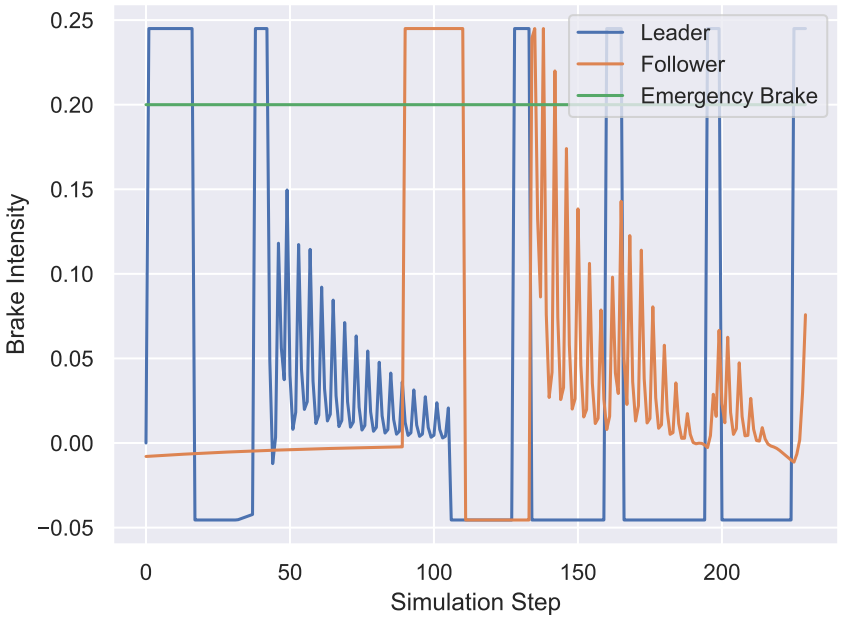
Table 4.1 shows the number and latency of trigger, function, and table creation according to the complexity of the property: n triggers are required to detect n state changes (e.g., a change in a specific value). If checks of sub-properties, nested, or higher-level properties are needed, it is possible to create functions that support them. Finally, a stored procedure for global-level monitoring is required, and it takes computing power equal to the Latency of Table 4.1, calculated as shown in Figure 4.3 as: $Time_{Detected} - Time_{Failure}$.

In this experiment, the system is also verified by periodically executing queries (not by event trigger), and in the worst case, 11 ms requires 1.1 % out of the 1000 ms periodic query, assuming that we have an empirically-defined 1-second period. Since several independent queries benefit from the database system's compiled query cache and parallel execution, the actual execution of all properties p_1-p_n converges within 30 ms latency on average. The average is close to 1/3 of the shortest latency required by tasks in truck platooning applications, meaning that latency does not affect application performance.

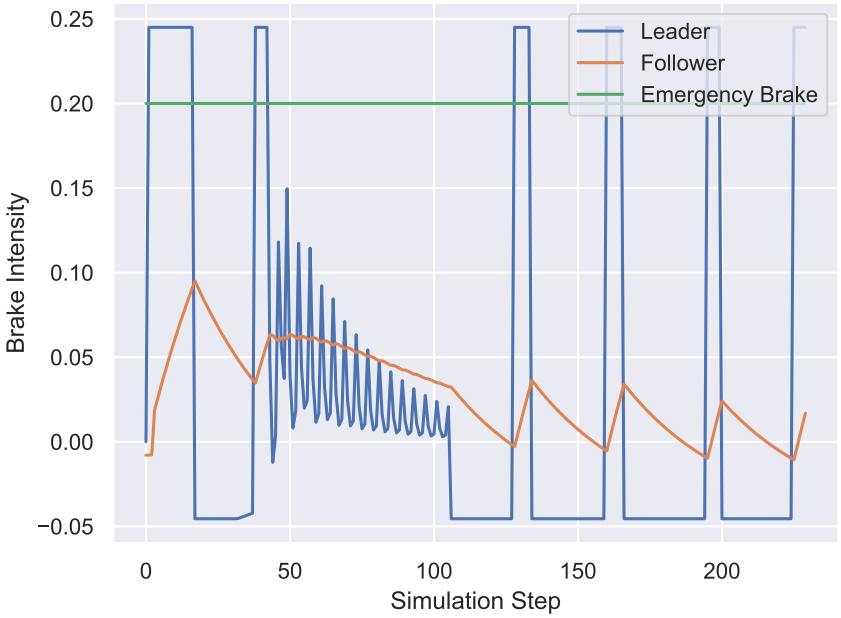
4.3.2 Preliminary Result from A Use Case

While not the primary focus of DRIVE, which aims to provide a methodology for performing distributed runtime verification, we conducted preliminary experiments to showcase how applying this distributed runtime verification can be beneficial to autonomous systems.

In this experiment, we show that DRIVE can smooth out the brake intensity curve of a follower truck in an emergency braking scenario. Vehicles can significantly benefit from



(a) Brake Intensity without RV.



(b) Brake Intensity with RV.

Figure 4.7: Benefits from Distributed Runtime Verification.

starting braking actions earlier, in a smooth lower-intensity manner. For instance: (i) it has a significant negative correlation with the number of crashes [71], which can help vehicles drive more safely; (ii) it can reduce their fuel consumption considerably [62] and gain up to 17% increase in energy recovery for regenerative-braking-equipped vehicles [83], which can increase the driving range; and, (iii) it can reduce brake and tire wear by decreasing the particulate matter emission (PME). A recent study reveals that the brake intensity has a significant role in particular matter emission: the higher the brake intensity, the more the PME [66]. Moreover, heavier vehicles, such as trucks, tend to emit more particles and can benefit greatly from this decrease in particle emission.

The experimental setup consists of a two-truck platoon and a slower passenger car (i.e., the obstacle). We use a two-truck platoon without loss of generality; we discuss how this experiment can scale up at the end of this section. The platoon's driving speed is double the obstacle's, and all cars use the state-of-the-art Cooperative Adaptive Cruise Control (CACC) car-following model [24]. The obstacle starts at a large distance ahead of the platoon, so the platoon drives at a fast speed until it suddenly has to brake harshly to avoid a collision with the obstacle due to its sensing limitations. We then perform this experiment twice, once with distributed runtime verification and once without it. Finally, we considered a detection delay of 1 simulation step, as we set the simulation step to match the worst distributed query latency measured in Section 4.3.1.

Figure 4.7 shows the brake intensity of the platooning vehicles, measured as the negative speed change in the simulation, over the course of the simulations. On the left plot (Figure 4.7a), the data depicts the brake intensity of the vehicles without distributed runtime verification. In this situation, both vehicles have to do emergency braking once the platoon gets close to the obstacle and the CACC model adjusts its behavior.

The right plot (Figure 4.7b) shows how the vehicles would act with distributed RV. In this scenario, once the leader performs the initial emergency braking, it sends a notification to

the follower on the next simulation step, which accounts for the property violation detection latency. The follower then can start braking earlier to open a gap between itself and the leader. This proactive measure allows for less intense braking action in future simulation steps, as the vehicle has already started slowing down earlier and has opened an extra gap between the vehicle ahead.

The brake intensity reduction is significant, as in the second experiment, the follower vehicle has to brake at less than 50% of the original intensity from the first experiment. This reduction allows the follower-vehicle to reduce the amount of torque used in braking, decreasing the chance of violating the safe limit defined by the vehicle's safety properties.

These results are promising for the applicability of distributed runtime verification, which could also produce good results for different scenarios and properties. Finally, although these results only depict the behavior of a two-truck platoon, the change to a scaled version with a larger platoon is trivial: the leader would broadcast the notification to all vehicles. Since each vehicle processes this notification and acts individually, the results for additional trucks added at the end would be similar to those for the follower truck above.

Chapter 5

Reasoning About Emergency

Through the deployment of SAFER and DRIVE together, a user can enable the runtime monitoring of individual and collective systems. However, both frameworks aim to detect emergent behaviors, and while SAFER proposes recovery methods, it uses blanket actions for all anomalies. Thus, there is still a lack in the overall system design, as it does not reason about the detected unexpected behaviors.

In this chapter, I will analyze a particular type of anomaly that can be detected at runtime: CAN (Controller Area Network) bus attacks. Modern vehicles are a distributed system by design, where all their functionality is spread out to different electrical component units (ECUs) that have specific goals. Due to this distributed design, some vehicles need over 70 ECUs to operate correctly [39]. In order to enable the many ECUs to communicate necessary information with each other, most vehicles use the CAN bus. Although the industry standard since 1996 [4], the CAN bus is still vulnerable to attacks from bad actors. Since messages are neither encrypted nor signed, if a malicious agent can gain access to the CAN bus, it can read all shared messages and send messages that other ECUs will receive [39, 38, 73, 37]. Such attacks happen in our day-to-day lives outside of academic and industry experiments,

e.g., [34].

Much work has been proposed in the literature to detect intrusions in the CAN bus, e.g., [48, 38, 40, 4, 73, 37]. However, similar to SAFER and DRIVE, most Intrusion Detection Systems (IDSs), e.g., [38, 48, 73, 37], stop the analysis at the detection stage, so there is no analysis after an intruder has been detected, and thus do not present pathways to facilitate safe responses. These proposed frameworks block the messages from being processed at ECUs but do not analyze if there is a better response for the ongoing attack. Attack type classification is vital to safety-critical systems as it allows systems to better react to ongoing attacks and prepare for future ones [45]. The few works that have approached the issue of attack classification on the CAN bus [40, 4], do not analyze how much overhead their complex classification models add to the system runtime; and high overheads can compromise safe responses.

With this lack in the literature in mind, I introduce LOCoCAT (Low-Overhead Classification of CAN bus Attack Types), an attack-type classification system that can identify what type of attack the intruder is executing with very little overhead. The proposed approach can extend any state-of-the-art IDS and adds minimal power, memory, and processing requirements to the overall system. It works by (i) grouping detected malicious messages into attack blocks, (ii) calculating discrete features for each attack block, and (iii) using a lightweight classifier model to identify the type of each attack. We evaluate the applicability of our proposal using three of the most popular available CAN bus datasets in the literature [38, 73, 37], observing excellent classification results with low latency in an energy- and resource-constrained environment.

The rest of this chapter is structured as follows: Section 5.1 discusses the CAN bus protocol, CAN bus attacks, existing solutions, and their shortcomings; Section 5.2 introduces the LOCoCAT framework in detail; and, finally, Section 5.3 evaluates LOCoCAT and compares its results to existing literature.

5.1 Controller Area Network

The Controller Area Network (CAN) bus is an international standard first defined in 1993, intending to replace complex automotive wiring with a simple two-wire bus [41]. A CAN bus differs from most common networks as it allows nodes to send large amounts of data to a specific destination; instead, a node in the CAN bus broadcasts multiple short messages to all the nodes in the system. The idea is that if a node broadcasts many short messages on the bus (e.g., RPM measurements over time), the data should be consistent throughout the network. However, due to its design, where nodes broadcast their readings to the entire network, and there is no verification or signing of data, the CAN bus is vulnerable to malicious attacks [18].

5.1.1 CAN Bus Attacks

When the CAN bus was initially proposed, security was not its primary consideration since it was used to connect very few ECUs and was not accessible by the user [18]. However, the automotive industry changed drastically over the years, and now there are significantly more ECUs connected to the CAN bus, and it can even be required by law for the bus to be accessible for diagnostics [20].

Therefore, although the CAN bus has some security features in its design (e.g., error checking), it can still be attacked. A lack of authentication allows any node to join the network and communicate with all the other nodes regardless of its intent. A lack of encryption allows any node, malicious or not, to listen and understand the data. The combination of these two also allows an attacker to gain knowledge about the system and inject malicious data back into it.

5.1.2 Attack Datasets

Three main attack datasets in the literature are openly available and explored by novel work that proposes new IDSs. They are:

1. SynCAN: introduced by Hanselmann et al. [38], SynCAN is a synthetic CAN bus dataset, to serve as a benchmark for CAN bus IDSs. SynCAN has up to 4 signal values per message depending on the ECU sending it, and contains five different attack types : (i) continuous, (ii) plateau, (iii) playback, (iv) suppress, and (v) flooding. The length of SynCAN attacks ranges from 4 to 8 seconds. SynCAN is available at `https://github.com/etas/SynCAN`.

2. Survival-IDS: introduced by Han et al. [37], Survival-IDS is a dataset containing real CAN bus data from 3 different vehicles – Hyundai Sonata, Kia Soul, and Chevrolet Spark. Survival-IDS has up to 8 signal values per message depending on the ECU sending it, and contains three different attack types: (i) flooding, (ii) fuzzy, and (iii) malfunction. The length of Survival-IDS attacks is 5 seconds. Survival-IDS is available at `https://ocslab.hksecurity.net/Datasets/survival-ids`.

3. Car-Hacking: introduced by Seo et al. [73], Car-Hacking is a dataset containing real CAN bus data from a Hyundai YF Sonata. Car-Hacking has up to 8 signal values per message depending on the ECU sending it, and contains four different attack types: (i) denial of service, (ii) fuzzy, (iii) drive gear spoofing, and (iv) RPM gauge spoofing. The length of Car-Hacking attacks ranges from 3 to 5 seconds. Car-Hacking is available at `https://ocslab.hksecurity.net/Datasets/car-hacking-dataset`

5.1.3 Intrusion Detection Systems

Many Intrusion Detection Systems (IDSs) have been proposed in the literature to detect CAN bus attacks. Among them, Hanselmann et al. [38] propose an IDS based on a combination of various neural networks, achieving around 87% accuracy in detecting anomalous messages on the SynCAN. Han et al. [37] propose an IDS using heuristic-based algorithms to perform a survival analysis. Their approach achieves an F1-score of over 95% for detecting each attack type. Seo et al. [73] propose a neural network based IDS that achieves 97.725% accuracy. Their model uses a powerful GPU-equipped computer, but no overhead analysis is provided.

Kukkala et al. [48] propose LATTE, an IDS based on a combination of neural networks and support vector machines to detect CAN bus attacks with an average improvement of 18.94% in accuracy compared to other previous IDSs for CAN bus data. Additionally, Kukkala et al. provide a complete overhead analysis of LATTE using an NVIDIA Jetson TX2 as their platform, where it achieves a latency of 193μs. The Jetson TX2 is a powerful GPU-equipped embedded device often used as an additional ECU in the literature, showing the applicability of LATTE in a real-world system.

5.1.4 Attack Types

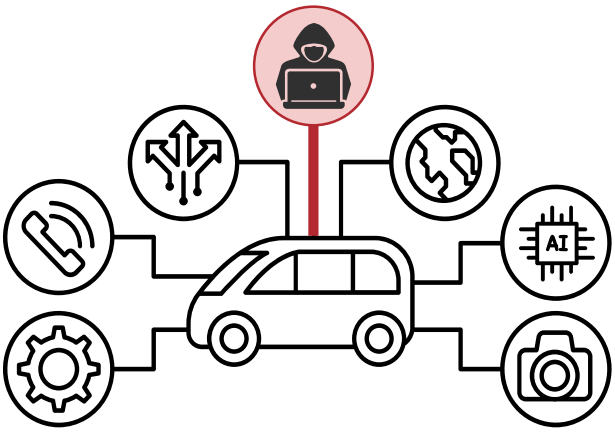
In addition to gaining access to the bus, an attacker can craft malicious messages with different intents and outcomes. If the attacker has domain knowledge about the vehicle, they can spoof the RPM gauge and drive gear [73], make the vehicle malfunction [37], or even steal it by opening its doors and turning it on to drive away [34]. Furthermore, even if the attacker does not have domain knowledge, they can still affect the vehicle by, e.g., flooding the CAN bus to achieve a denial of service attack [38, 37, 73] or sending messages with random values to make actuation fuzzy [73, 37]. Thus, it is essential to detect CAN bus attacks and identify the attacker's intent so the system can react appropriately [45].

Existing literature that identifies the types of CAN bus attacks is limited. To the best of our knowledge, [4] and [40] are the only works that analyze CAN bus attacks as a multi-class classification problem.

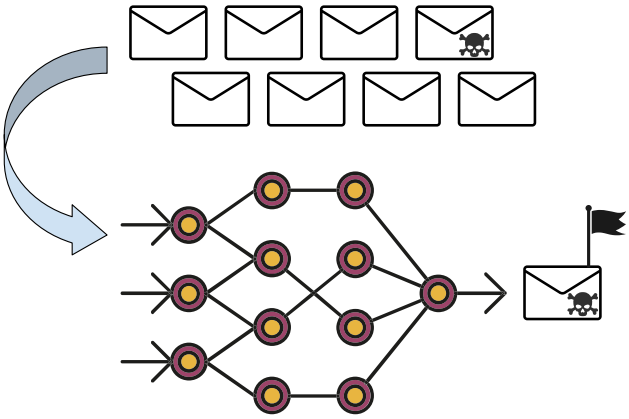
Amato et al. [4] propose an IDS for Car-Hacking that uses a neural network model to classify Car-Hacking attacks in multi-class labels achieving a weighted F1-score of 96.8%. However, the authors do not mention the platform where experiments were run or analyze their approach's overhead. Hossain et al. [40] propose an IDS for Survival-IDS that uses a neural network model to classify Survival-IDS attacks in multi-class labels achieving a weighted F1-score of 98.84%. Their model uses a powerful GPU-equipped desktop computer, but no overhead analysis is provided. Since neither work provides a complete overhead analysis and [40] uses a powerful platform, it is unknown whether their approaches are feasible in the context of a real-time safety-critical system. In Section 5.3, we compare our approach's accuracy results with [4, 40] and provide a complete overhead analysis.

5.2 The LOCoCAT Framework

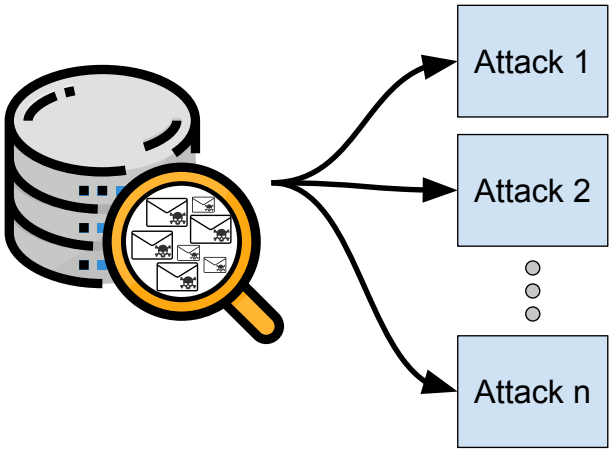
Figure 5.1 outlines the entire process for type classification of attacks on the CAN bus. Figure 5.1a shows an attacker that has gained access to the bus and can send malicious messages. Next, Figure 5.1b depicts a state-of-the-art IDS from the literature that can detect malicious messages among normal data in the CAN bus. As a proof of concept, we use LATTE [48] as the framework's IDS due to its high accuracy and low latency; however, replacing it in the overall framework would be easy as new models appear in the literature. Lastly, Figure 5.1c shows our contribution – a framework that can classify the attack type of groups of malicious messages detected by the IDS into previously known types. As Figures 5.1a and 5.1b are well covered in the literature, the rest of this section will focus on Figure 5.1c, which is our main contribution.



(a) An attacker gains access to the CAN bus and sends malicious messages.



(b) A state-of-the-art IDS flags the malicious messages on the bus.



(c) LOCoCAT analyzes the flagged messages and identifies the attack type.

Figure 5.1: Step-by-step process for classifying CAN bus attacks.

LOCoCAT aims first to quickly, accurately, and non-intrusively identify the type of an ongoing attack so that the system can choose the best reaction available. Additionally, since our approach requires an IDS to flag malicious messages and the most accurate IDSs in the literature (e.g., [48, 38]) use complex models that require a GPU-equipped board to function (e.g., NVIDIA Jetson TX2), LOCoCAT does not significantly affect the overall overhead added to the system.

We achieve both of these goals by exploring models that (i) have good accuracy results, (ii) can run on low-end hardware, (iii) are lightweight, and (iv) have an acceptable latency when considering the usual length of CAN bus attacks.

Our proposed classification methodology analyzes messages flagged as malicious by an existing IDS. Firstly, we group malicious messages with timestamps close to each other into attack blocks. Secondly, we choose a window at the start of the attack blocks to analyze, as we want a classification result before the ongoing attack ends. Thirdly, we calculate discrete features for the attack block window, resulting in a finite set of features for all attack blocks, regardless of how many messages were a part of the attack. Finally, we feed this discrete featureset into a classifier model to get the attack type. The rest of this section describes the steps of our approach in more detail.

5.2.1 Attack Blocks and Features

Each CAN bus attack consists of a sequence of messages that the attacker sends during a short time among the normal messages. Since LOCoCAT already knows which messages are malicious due to the IDS, we group the malicious messages into blocks while an attack occurs. An attack block will then have a finite number of features. We calculate three metrics for each signal in the messages: (i) median, (ii) average, and (iii) standard deviation. We also count how many messages are a part of this attack block. Overall, we have a total

of 13 features for each attack block of the SynCAN dataset [38], as it has up to 4 signals per message, and 25 features for each attack block of the Car-Hacking [73] and Survival-IDS [37] datasets, as they each have up to 8 signals per message.

5.2.2 Window Selection

One of the goals of identifying the type of attack is to give the system more information on how to best react. Thus, it is crucial to have a classification result before the attack is over. Since the length of each CAN bus attack on the literature datasets [38, 73, 37] range from 3 to 8 seconds, we decided to use at most 1s of the attack for classification. This gives enough time for the system to decide and execute a reaction to the ongoing attack. Therefore, we explore models that consider messages up to 50ms, 100ms, 200ms, 500ms, and 1000ms from the start of the attack.

5.2.3 Model Selection

To develop our framework, we use Python3 with the scikit-learn [67] library to train all models and run all inferences. Scikit-learn provides an easy way to train and export models that can run on CPU-only platforms. We consider the following multi-class classification models available in the library: Adaptive Boosting, Decision Tree, Gaussian Naive Bayes, k-Nearest Neighbors, Quadratic Discriminant Analysis, and Random Forest. We perform a parameter grid search for each model to find the best results considering each window length option. The results shown in Section 5.3 are obtained from the best parameter combination of each model.

Table 5.1: Weighted F1-Score for Car-Hacking

Model	1000ms	500ms	200ms	100ms	50ms
AdaBoost	73.75%	73.75%	73.75%	73.75%	73.75%
Decision Tree	79.58%	83.33%	97.50%	98.75%	96.66%
Gaussian NB	84.58%	84.58%	97.08%	98.75%	99.16%
kNN	70.83%	76.25%	94.58%	97.08%	95.41%
QDA	20.41%	20.41%	20.41%	20.41%	20.41%
Random Forest	84.58%	84.16%	97.50%	97.91%	97.91%

5.3 Evaluation

All experiments were executed on a Raspberry Pi Zero W with a 1GHz single-core ARMv6 CPU and 512MB of memory, and all scripts and results are made available online at `https://github.com/cbdm/LOCoCAT`. We use three datasets with malicious CAN bus data from the literature for all experiments: (i) SynCAN [38], (ii) Car-Hacking [73], and (iii) Survival-IDS [37].

We prepare all datasets for our experiments by (i) filtering out the normal messages, (ii) creating attack blocks with malicious messages that are part of the same attack, (iii) shuffling the attack blocks, and (iv) splitting these blocks into 80% for training and 20% for testing. In the following two subsections, we analyze two different aspects of our approach. First, in Section 5.3.1, we analyze our proposal's performance in correctly identifying the ongoing attack. Then, in Section 5.3.2, we analyze the added overhead of the models.

5.3.1 Correct Classification

Tables 5.1 and 5.2 depict the weighted F1-scores for the different models and windows explored for the Car-Hacking and SynCAN datasets, respectively. We chose to omit the complete results for Survival-IDS as there were too few attack blocks to make a thorough analysis. When grouping the malicious messages of each dataset in attack blocks, the Survival-IDS

Table 5.2: Weighted F1-Score for SynCAN

Model	1000ms	500ms	200ms	100ms	50ms
AdaBoost	77.06%	75.22%	58.71%	60.55%	47.70%
Decision Tree	76.14%	77.98%	71.55%	58.71%	44.03%
Gaussian NB	63.30%	65.13%	59.63%	53.21%	45.87%
kNN	73.39%	68.80%	61.46%	55.04%	43.11%
QDA	41.28%	14.67%	27.52%	23.85%	19.26%
Random Forest	76.14%	77.06%	74.31%	57.79%	53.21%

dataset had only 29 separate attacks, while Car-Hacking (1,200) and SynCAN (545) had considerably more. However, it is still worth mentioning that a decision tree model was able to achieve a weighted F1-score of 100% with a 100ms window, which is on par with Hossain et al. [40], whose work uses a neural network running on a GPU to classify the dataset.

As Table 5.1 shows, our proposal works exceptionally well for the Car-Hacking dataset, achieving an F1-score of over 99% with a Gaussian Naive Bayes model considering only the first 50ms of each attack block. These results are extremely promising to the applicability of our approach in a real-world system, as using a resource-constrained platform with very low power requirements, we can still achieve outstanding classification results for a 4-class problem. LOCoCAT is vastly superior to a random classifier, where a scikit-learn DummyClassifier model was able to achieve 27.91%. It also shows better F1 scores than the existing work that analyzes this dataset as a multi-class problem – Amato et al. [4] propose a neural network model that achieves an F1-score of 96.8%.

Although the scores for the SynCAN dataset shown in Table 5.2 are not as high as the ones observed for Car-Hacking, the results are also promising. The SynCAN dataset contains one extra attack, which brought the DummyClassifier F1-score down to 27.52%. Unfortunately, to the best of our knowledge, no other work has analyzed the SynCAN dataset as a multi-class problem, so we cannot compare our results to other existing literature. Nonetheless, LOCoCAT can achieve an F1-score of nearly 78%, using a decision tree and the first 500ms

Table 5.3: Average Inference Latency (ms)

Model	1000ms	500ms	200ms	100ms	50ms
AdaBoost	93.6	25.9	68.7	69.2	68.7
Decision Tree	3.3	3.3	3.3	3.3	3.3
Gaussian NB	9.2	9.2	9.2	9.3	9.3
kNN	15.6	15.5	15.4	15.5	15.6
QDA	15.5	15.5	15.5	15.5	15.5
Random Forest	75.7	52.1	40.4	33.4	59.5

of each attack, which is an improvement of over 2.8X on the DummyClassifier.

5.3.2 Framework Overhead

First, we look at the inference latency, as we want the classification process to conclude before the attack is over to allow the system to react. Table 5.3 shows the average latency in ms for each model-window combination over a total of 174,500 inferences. The two models that perform most accurately, Gaussian Naive Bayes (50ms) and Decision Tree (500ms) (see Section 5.3.1), had very low inference times with 9.3ms and 3.3ms. For the Car-Hacking dataset, this would account for a total classification time of 59.3ms – 50ms of receiving messages and 9.3ms of inference. Since the Car-Hacking attacks range from 3s to 5s, the system would have at least 2.94s to react. For the SynCAN dataset, this would account for a total classification time of 503.3ms – 500ms of receiving messages and 3.3ms of inference. Since the SynCAN attacks range from 4s to 8s, the system would have at least 3.49s to react. In both cases, the system should have enough time to choose and deploy the appropriate reaction for the ongoing attack type.

Next, we analyze the space requirements. Table 5.4 shows the average sizes in KB for each considered model. The most accurate models, Gaussian Naive Bayes (50ms) and Decision Tree (500ms), would only require 1.9 KB and 5.8 KB, respectively. These sizes are much smaller than more complex neural network models in the literature, e.g., LATTE requires

Table 5.4: Average Trained Model Size (KB)

Model	1000ms	500ms	200ms	100ms	50ms
AdaBoost	174.2	56.0	174.2	174.2	174.2
Decision Tree	42.4	5.8	3.4	4.3	13.2
Gaussian NB	1.9	1.9	1.9	1.9	1.9
kNN	1496.5	1496.5	1496.5	1496.5	1496.5
QDA	156.6	156.6	156.6	156.6	156.6
Random Forest	40.2	19.1	228.4	20.3	26.9

1,439 KB [48].

These results show the feasibility of running LOCoCAT under strict resource constraints. Additionally, by using a low-power platform such as the Raspberry Pi Zero W, we can significantly reduce the power consumption we add to the system. A Raspberry Pi Zero W operating at its maximum CPU usage consumes 1.512W [58], whereas an NVIDIA Jetson TX2 can use nearly ten times that at 15W [64].

Chapter 6

Conclusion and Future Work

Autonomous vehicles (AVs) have emerged as the future of transportation and promise many benefits as they become more widely adopted [69]. However, as complex safety-critical cyber-physical systems, AVs face many challenges. In particular, AVs must correctly and safely operate in real-world scenarios that are hard to predict at design time and, in most cases, unfeasible to explore all possibilities. Therefore, much work has to be done to ensure that AVs can operate correctly in the face of uncertainty and unexpected situations. This dissertation introduces three frameworks to help with such concerns: SAFER, DRIVE, and LOCoCAT.

First, the SAFER methodology aims to detect emergent (mis)behaviors to ensure the reliability and safety of complex systems. SAFER's utility is illustrated through four diverse case studies, with a greater focus on a safety-critical Collision Avoidance system. Across all case studies and experiments, SAFER showed an average of 83.82% F1 score for PSM-guided anomaly detection. This score is on par with the current state of the art, with the advantage of being applicable to diverse systems.

Next, this dissertation introduced the DRIVE methodology, which aims to enable distributed

runtime verification in collective autonomous systems. Through a state-of-the-art-based truck platooning case study, the usefulness of the proposed approach is validated. Across the experiments, the distributed runtime verification detected all safety property violations, with most queries resolved under 1 ms and a maximum latency of 11 ms, demonstrating the feasibility of deploying DRIVE in practice. Additionally, preliminary results support the benefits of using DRIVE as a distributed verification method to gain energy efficiency and braking wear-out.

Lastly, this dissertation proposes LOCoCAT, a framework that can identify the type of ongoing attacks on the CAN bus with very little overhead. LOCoCAT's efficacy is evaluated using three existing datasets used in the literature to develop intrusion detection systems for the CAN bus, and we can accurately classify attacks with better results than related works. Additionally, the classification latency is appropriate for the attack lengths, as it allows the system plenty of time to react, and the memory and power requirements are much lower than related works.

Through the combination of SAFER, DRIVE, and LOCoCAT, the work in this dissertation contributes to enabling individual and collective systems to detect unexpected situations and reason about them at runtime. Additionally, it lays the groundwork for a rich set of future work. SAFER enables future work such as complex verification with anomaly detection via unmodified open-source Common Vulnerabilities and Exposures (CVE) cases, detection latency analysis, and hardware implementation for non-intrusive monitoring for detection. DRIVE can be extended to enable vehicles to automatically modify their reactions according to the sensed data with online runtime verification and continue our experiments on the ancillary benefits of energy efficiency and truck lifetime. It would also be interesting to extend the current experiments from SUMO to TruckSim [1], which can produce results with more detailed physics and extended traffic simulations. Combining the TruckSim results

[1] https://www.carsim.com/products/trucksim/

with small-scale simulations would enable a digital twin [44] analysis of DRIVE, as we can connect the physics-rich results from TruckSim with real-world experiments using truck models. LOCoCAT can be expanded to work with attack types that are unknown to the system or are a mix of existing attacks. Additionally, as its overhead results show extreme promise, an analysis to explore the feasibility of running it on a platform with even lower resource requirements, such as the Raspberry Pi Pico, would be interesting.

Bibliography

[1] C. C. Aggarwal. Outlier analysis. In *Data mining*, pages 237–263. Springer, 2015.

[2] S. Agrawal and J. Agrawal. Survey on anomaly detection using data mining techniques. *Procedia Computer Science*, 60:708–713, 2015.

[3] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[4] F. Amato, L. Coppolino, F. Mercaldo, F. Moscato, R. Nardone, and A. Santone. Can-bus attack detection with deep learning. *IEEE Transactions on Intelligent Transportation Systems*, 22(8):5081–5090, 2021.

[5] J. An and S. Cho. Variational autoencoder based anomaly detection using reconstruction probability. *Special Lecture on IE*, 2(1):1–18, 2015.

[6] A. Balador, A. Bazzi, U. Hernandez-Jayo, I. de la Iglesia, and H. Ahmadvand. A survey on vehicular communication for cooperative truck platooning application. *Vehicular Communications*, 35:100460, 2022.

[7] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 13–24, 2005.

[8] E. Bartocci, L. Bortolussi, M. Loreti, and L. Nenzi. Monitoring mobile and spatially distributed cyber-physical systems. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, MEMOCODE '17, page 146–155, New York, NY, USA, 2017. Association for Computing Machinery.

[9] E. Bartocci, L. Bortolussi, M. Loreti, L. Nenzi, and S. Silvetti. Moonlight: A lightweight tool for monitoring spatio-temporal properties. In *International Conference on Runtime Verification*, pages 417–428. Springer, 2020.

[10] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan. Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In *Lectures on Runtime Verification*, pages 135–175. Springer, 2018.

[11] D. Basin, M. Gras, S. Krstić, and J. Schneider. Scalable online monitoring of distributed systems. In *International Conference on Runtime Verification*, pages 197–220. Springer, 2020.

[12] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu. Monitoring metric first-order temporal properties. *Journal of the ACM (JACM)*, 62(2):1–45, 2015.

[13] A. Bauer and Y. Falcone. Decentralised ltl monitoring. In *International Symposium on Formal Methods*, pages 85–100. Springer, 2012.

[14] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):1–64, 2011.

[15] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for ltl and tltl. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):1–64, 2011.

[16] D. M. Blough, F. J. Kurdahi, and Seong Yong Ohm. High-level synthesis of recoverable vlsi microarchitectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(4):401–410, 1999.

[17] L. Bortolussi and L. Nenzi. Specifying and monitoring properties of stochastic spatio-temporal systems in signal temporal logic. In *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*, pages 66–73, 2014.

[18] M. Bozdal, M. Samie, and I. Jennions. A survey on can bus protocol: Attacks, challenges, and potential solutions. In *2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*, pages 201–205. IEEE, 2018.

[19] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.

[20] R. Buttigieg, M. Farrugia, and C. Meli. Security issues in controller area networks in automobiles. In *2017 18th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*, pages 93–98. IEEE, 2017.

[21] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[22] C. B. de Melo, A. L. F. Cançado, and G. N. Rodrigues. Characterization of implied scenarios as families of common behavior. *Journal of Systems and Software*, 158:110425, 2019.

[23] J. Deogirikar and A. Vidhate. Security attacks in iot: A survey. In *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pages 32–37, 2017.

[24] K. C. Dey, L. Yan, X. Wang, Y. Wang, H. Shen, M. Chowdhury, L. Yu, C. Qiu, and V. Soundararaj. A review of communication, driver characteristics, and controls aspects of cooperative adaptive cruise control (cacc). *IEEE Transactions on Intelligent Transportation Systems*, 17(2):491–509, 2015.

[25] DLR, German Aerospace Center, et al. Simpla. `https://sumo.dlr.de/docs/Simpla.html`, 2012. Accessed: 2023-04-11.

[26] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[27] T. F. Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 2017.

[28] M. Dunne, G. Gracioli, and S. Fischmeister. A comparison of data streaming frameworks for anomaly detection in embedded systems. In *Proceedings of the 1st International Workshop on Security and Privacy for the Internet-of-Things (IoTSec), Orlando, FL, USA*, 2018.

[29] R. Elmasri and S. B. Navathe. *Fundamentals of database systems*. Addison Wesley, sixth edition, 2011.

[30] S. R. Gaddam, V. V. Phoha, and K. S. Balagani. K-means+ id3: A novel method for supervised anomaly detection by cascading k-means clustering and id3 decision tree learning methods. *IEEE transactions on knowledge and data engineering*, 19(3):345–354, 2007.

[31] A. George and A. Vidyapeetham. Anomaly detection based on machine learning: dimensionality reduction using PCA and classification using SVM. *International Journal of Computer Applications*, 47(21):5–8, 2012.

[32] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *International Conference on Protocol Specification, Testing and Verification*, pages 3–18. Springer, 1995.

[33] J. B. Goodenough and L. Sha. The priority ceiling protocol: A method for minimizing the blocking of high priority ada tasks. *ACM SIGAda Ada Letters*, 8(7):20–31, 1988.

[34] D. Goodin. There's a new form of keyless car theft that works in under 2 minutes. *Ars Technica*, Apr 2023. Accessed on 2023-04-30.

[35] I. Haghighi, A. Jones, Z. Kong, E. Bartocci, R. Gros, and C. Belta. Spatel: a novel spatial-temporal logic and its applications to networked systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 189–198, 2015.

[36] A. H. Hamamoto, L. F. Carvalho, L. D. H. Sampaio, T. Abrão, and M. L. Proença Jr. Network anomaly detection system using genetic algorithm and fuzzy logic. *Expert Systems with Applications*, 92:390–402, 2018.

[37] M. L. Han, B. I. Kwak, and H. K. Kim. Anomaly intrusion detection method for vehicular networks based on survival analysis. *Vehicular Communications*, 14:52–63, 2018.

[38] M. Hanselmann, T. Strauss, K. Dormann, and H. Ulmer. Canet: An unsupervised intrusion detection system for high dimensional can bus data. *IEEE Access*, 8:58194–58205, 2020.

[39] R. Hegde, G. Mishra, and K. S. Gurumurthy. An Insight into the Hardware and Software Complexity of ECUs in Vehicles. *Communications in Computer and Information Science*, pages 99–106, 2011.

[40] M. D. Hossain, H. Inoue, H. Ochiai, D. Fall, and Y. Kadobayashi. Lstm-based intrusion detection system for in-vehicle can bus communications. *IEEE Access*, 8:185489–185502, 2020.

[41] S. C. HPL. Introduction to the controller area network (can). *Application Report SLOA101*, pages 1–17, 2002.

[42] R. Hult, M. Zanon, S. Gros, and P. Falcone. Optimal coordination of automated vehicles at intersections: Theory and experiments. *IEEE Transactions on Control Systems Technology*, 27(6):2510–2525, 2019.

[43] G. Janssen, J. Zwijnenberg, I. Blankers, and J. de Kruijff. Truck platooning: Driving the future of transportation, 2015.

[44] D. Jones, C. Snider, A. Nassehi, J. Yon, and B. Hicks. Characterising the digital twin: A systematic literature review. *CIRP Journal of Manufacturing Science and Technology*, 29:36–52, 2020.

[45] H. A. Kholidy. Autonomous mitigation of cyber risks in the cyber–physical systems. *Future Generation Computer Systems*, 115:171–187, 2021.

[46] H. Kopetz, C. E. Salloum, B. Huber, and R. Obermaisser. Periodic finite-state machines. In *Tenth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2007), 7-9 May 2007, Santorini Island, Greece*, pages 10–20. IEEE Computer Society, 2007.

[47] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[48] V. K. Kukkala, S. V. Thiruloga, and S. Pasricha. Latte: Lstm self-attention based anomaly detection in embedded automotive platforms. *ACM Trans. Embed. Comput. Syst.*, 20(5s), sep 2021.

[49] T. Li, J. Liu, H. Sun, X. Chen, L. Yin, X. Mao, and J. Sun. Runtime Verification of Spatio-Temporal Specification Language. *Mobile Networks and Applications*, 26(6):2392–2406, 2021.

[50] P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. Wießner. Microscopic traffic simulation using sumo. In *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018.

[51] Y. Luo. Time constraints and fault tolerance in autonomous driving systems. Master's thesis, EECS Department, University of California, Berkeley, May 2019.

[52] B. Maity, S. Yi, D. Seo, L. Cheng, S.-S. Lim, J.-C. Kim, B. Donyanavard, and N. Dutt. Chauffeur: Benchmark suite for design and end-to-end analysis of self-driving vehicles on embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–22, 2021.

[53] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer, 2004.

[54] C. Manikopoulos and S. Papavassiliou. Network intrusion and fault detection: a statistical anomaly approach. *IEEE Communications Magazine*, 40(10):76–82, 2002.

[55] N. Markey and P. Schnoebelen. Model checking a path. In *International Conference on Concurrency Theory*, pages 251–265. Springer, 2003.

[56] D. Marmsoler and A. Petrovska. Runtime verification for dynamic architectures. *Journal of Logical and Algebraic Methods in Programming*, 118:100618, 2021.

[57] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[58] A. G. Millard, R. Joyce, J. A. Hilder, C. Fleşeriu, L. Newbrook, W. Li, L. J. McDaid, and D. M. Halliday. The pi-puck extension board: A raspberry pi interface for the e-puck robot platform. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 741–748, 2017.

[59] J. C. Mogul. Emergent (mis) behavior vs. complex software systems. *ACM SIGOPS Operating Systems Review*, 40(4):293–304, 2006.

[60] A. Momtaz. Runtime Verification for Distributed Cyber-Physical Systems. *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, 00:349–350, 2021.

[61] M. Mostafa and B. Bonakdarpour. Decentralized runtime verification of ltl specifications in distributed systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 494–503. IEEE, 2015.

[62] M. Muñoz-Organero and V. C. Magaña. Validating the impact on reducing fuel consumption by using an ecodriving assistant based on traffic sign detection and optimal deceleration patterns. *IEEE Transactions on Intelligent Transportation Systems*, 14(2):1023–1028, 2013.

[63] L. Nenzi, E. Bartocci, L. Bortolussi, M. Loreti, and E. Visconti. Monitoring spatio-temporal properties (invited tutorial). In *International Conference on Runtime Verification*, pages 21–46. Springer, 2020.

[64] NVIDIA. Jetson TX2 module. *NVIDIA Developer*, Apr 2021. accessed on 2023-04-30.

[65] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . `https://www.openstreetmap.org`, 2017.

[66] F. Oroumiyeh and Y. Zhu. Brake and tire particles measured from on-road vehicles: Effects of vehicle mass and braking intensity. *Atmospheric Environment: X*, 12:100121, 2021.

[67] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[68] D. Pritchett. Base: An acid alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability. *Queue*, 6(3):48–55, 2008.

[69] M. V. Rajasekhar and A. K. Jaswal. Autonomous vehicles: The future of automobiles. In *2015 IEEE International Transportation Electrification Conference (ITEC)*, pages 1–6, 2015.

[70] E. A. Rambo, T. Kadeed, R. Ernst, M. Seo, F. Kurdahi, B. Donyanavard, C. B. de Melo, B. Maity, K. Moazzemi, K. Stewart, et al. The information processing factory: a paradigm for life cycle management of dependable systems. In *2019 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–10. IEEE, 2019.

[71] F. Sagberg, Selpi, G. F. B. Piccinini, and J. Engström. A Review of Research on Driving Styles and Road Safety. *Human Factors: The Journal of Human Factors and Ergonomics Society*, 57(7):1248–1275, 2015.

[72] M. Sakurada and T. Yairi. Anomaly detection using autoencoders with nonlinear dimensionality reduction. In *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*, pages 4–11, 2014.

[73] E. Seo, H. M. Song, and H. K. Kim. Gids: Gan based intrusion detection system for in-vehicle network. In *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, pages 1–6, Aug 2018.

[74] M. Seo and R. Lysecky. Automatic extraction of requirements from state-based hardware designs for runtime verification. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pages 295–298, 2019.

[75] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 827–838, 2004.

[76] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM (JACM)*, 32(3):733–749, 1985.

[77] F. Sommer, J. Dürrwang, and R. Kriesten. Survey and classification of automotive security attacks. *Information*, 10(4), 2019.

[78] P. Tallapragada and J. Cortés. Coordinated intersection traffic management. *IFAC-PapersOnLine*, 48(22):233–239, 2015. 5th IFAC Workshop on Distributed Estimation and Control in Networked Systems NecSys 2015.

[79] Y. Tan, M. C. Vuran, and S. Goddard. Spatio-temporal event model for cyber-physical systems. In *2009 29th IEEE International Conference on Distributed Computing Systems Workshops*, pages 44–50. IEEE, 2009.

[80] M. Uma and G. Padmavathi. A survey on various cyber attacks and their classification. *Int. J. Netw. Secur.*, 15(5):390–396, 2013.

[81] G. Wang, L. Zhang, and W. Xu. What can we learn from four years of data center hardware failures? In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 25–36. IEEE, 2017.

[82] A. Wegener, M. Piórkowski, M. Raya, H. Hellbrück, S. Fischer, and J.-P. Hubaux. Traci: An interface for coupling road traffic and network simulators. In *Proceedings of the 11th Communications and Networking Simulation Symposium*, CNS '08, page 155–163, New York, NY, USA, 2008. Association for Computing Machinery.

[83] L. Xu, X. He, and X. Shen. Improving energy recovery rate of the regenerative braking system by optimization of influencing factors. *Applied Sciences*, 9(18):3807, 2019.

[84] X. Ye, W. Liu, and N. Wang. Runtime Verification of Multi-Agent Self-Adaptive System. *2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 00:12–17, 2021.

[85] F. Yu, R. G. Dutta, T. Zhang, Y. Hu, and Y. Jin. Fast attack-resilient distributed state estimator for cyber-physical systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3555–3565, 2020.

[86] J. Yu. Fault detection using principal components-based gaussian mixture model for semiconductor manufacturing processes. *IEEE Transactions on Semiconductor Manufacturing*, 24(3):432–444, 2011.

[87] E. Zapridou, E. Bartocci, and P. Katsaros. Runtime verification of autonomous driving systems in carla. In *International Conference on Runtime Verification*, pages 172–183. Springer, 2020.

[88] L. Zhang, F. Chen, X. Ma, and X. Pan. Fuel economy in truck platooning: A literature overview and directions for future research. *Journal of Advanced Transportation*, 2020, 2020.

[89] C. Zhou and R. C. Paffenroth. Anomaly detection with robust deep autoencoders. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 665–674, 2017.