SAFER: Safety Assurances For Emergent Behavior

Caio Batista de Melo, Marzieh Ashrafiamiri, Minjun Seo, Fadi Kurdahi, *Fellow, IEEE*, Nikil Dutt, *Fellow, IEEE*

***Abstract*—Emergent behavior haunts the reliability and safety of complex software systems. Such behavior consists of unexpected operations that can arise at runtime and lead the system to potentially unsafe conditions. The SAFER framework allows users to model systems using Periodic State Machines and automatically generates code from a specification. It then creates a model that learns safe execution based on execution traces from the code. The applicability of SAFER is demonstrated through four case studies: Producer-Consumer, Collision Avoidance, Integer Overflow, and Stack Overflow. These case studies showcase the framework's ability to deal with emergent behaviors for different classes of applications.**

***Index Terms*—System modeling, runtime emergent behavior detection, runtime system recovery.**

I. Introduction

EMERGENT behavior often means complex systems operating in unexpected ways that are not easily predictable from the behavior of components [1]. This happens not only in components at different levels of abstraction such as hardware, software, operating systems, and 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, but the vibrations generated by adjacent drives adversely affect the stability of the entire system. In another example, well-known issues such as priority inversion in operating systems can result in blocking of higher-priority tasks because they require resources held by low-priority tasks, which can also be considered as emergent behavior.

These problems are exacerbated when computer systems are integrated into Cyber-Physical Systems (CPS) that sense environmental signals and perform computations and control for actuations in the physical world. Furthermore, CPS are often systems-of-systems, resulting 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 isolated parts of the system, and this requires a holistic system-level analysis.

A CPS is a non-terminating system continuously reacting 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 [2], 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 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 occurrence 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 you to take into account any location of the trace. On the other hand, RV – especially when dealing with online monitoring – takes into account finite executions of increasing size. For this, the monitor must be designed to take into account the execution in an incremental way. 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 [3]. 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 use of AD alone.

As such, RV and AD are useful in their respective domains but previously have not been used in combination effectively to detect emergent behaviors. This work proposes 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.

II. Correctness vs. Normalness

Our work distinguishes between correct vs. incorrect and normal vs. anomalous behaviors using the dichotomy shown

Caio Batista de Melo, Marzieh Ashrafiamiri, Minjun Seo, Fadi Kurdahi, and Nikil Dutt are with the Center for Embedded and Cyber-physical Systems at UC Irvine, Irvine, CA, USA. Email: cbatista@uci.edu, mashrafi@uci.edu, minjun.seo@uci.edu, kurdahi@uci.edu, dutt@ics.uci.edu.

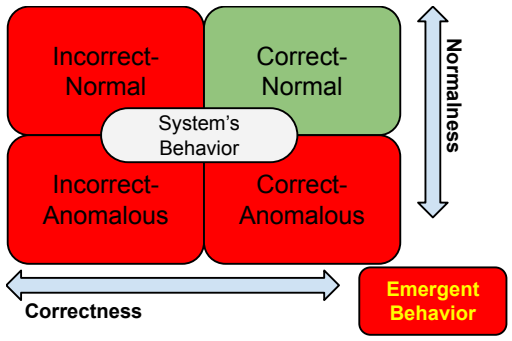

Fig. 1. Classifying a Cyber-Physical System's Behavior.

in Figure 1. Consequently, it is essential to distinguish between correct-normal and emergent behaviors of a system.

Any software component can be represented as 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 execution, some meaningless weird states can occur unintentionally. If we reach one of these weird states, we need a new computing device, also named as a weird machine [2], to sneak away from the unexpected states and move back to safe and predictable states.

We can classify the states occurring for a system's FSM into two categories: predicted (i.e., normal behavior) and unpredicted (i.e., anomalous behavior) [2]; and we may observe both desired (i.e., correct) and undesired (i.e., incorrect) behaviors for each category. The predicted desired behaviors are the ones included in the system's design (i.e., correct-normal). The predicted undesired category is the behaviors known to the designer as the existing problems (i.e., incorrect-normal). The unpredicted desired behaviors show a lack in the specification of the designed system (i.e., correct-anomalous). It is worth understanding why the system is acting in a way that it was not designed to. The last category is the unpredictable undesired behavior, which may cause severe problems for the system, and is also the source of unexpected behaviors of the system (i.e., incorrect-anomalous). Thus, we define an emergent behavior to be any system behavior that is incorrect or anomalous.

III. Anomaly Detection

A promising approach to detect emergent behaviors is through anomaly detection. Anomaly detection techniques can be split into three categories [3]: (i) statistical-based methods, (ii) proximity-based methods, and (iii) deviation-based methods.

Statistical-based techniques represent the input data as a statistical distribution; Proximity-based methods try to find whether a data is close to the majority of input data or not; and Deviation-based methods use dimensionality reduction as a first step to reduce the size of the input data, then use the reduced data to reconstruct the original data and calculate the reconstruction error between the original data and its reconstructed version.

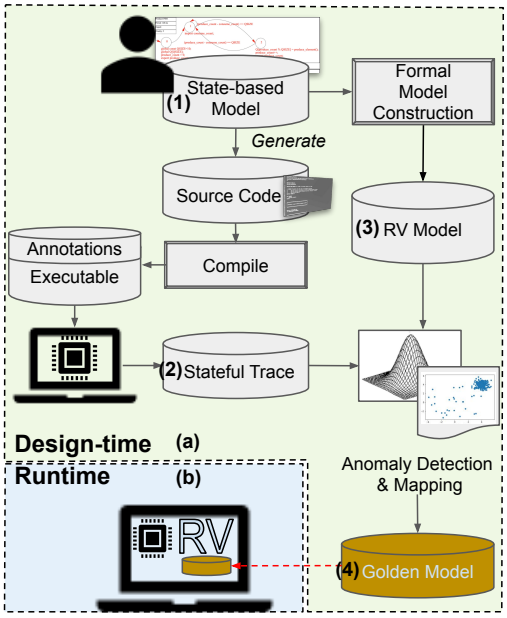

Fig. 2. Overview of SAFER methodology for (a) design-time, and (b) runtime.

Out of these three categories, deviation-based anomaly detection methods have recently received significant attention in the research community with promising results. In particular, autoencoder based anomaly detection methods have been enhanced using a more sophisticated neural network model called Variational AutoEncoders (VAE) [4]. VAEs can detect anomalies by first doing a dimensionality reduction, where the dataset is compressed to have a smaller number of features, called the latent space. Next, it tries to recreate the original dataset from the latent space using its system knowledge from training. Finally, it can find the probability that this original dataset depicts normal system behavior.

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.For instance, a recent VAE-based anomaly detection scheme [4] 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, we deploy a VAE-based anomaly detection method as a part of our SAFER methodology described next.

IV. SAFER Methodology

The SAFER (Safety Assurances For Emergent Behavior) methodology shown in Figure 2 consists of (a) a design-time methodology, and (b) online monitoring using non-intrusive hardware or software instrumentation.

The design-time methodology (Figure 2 (a)) first develops a state-based model using Periodic State Machines (PSMs) (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 body of the functions they defined. The compiled binary contains instrumentation based annotations, and running it on a real machine produces one or more sets of execution 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 execution trace (2) and the RV model (3) cross-reference each other to perform AD and model updates, respectively. This gives a golden model for online monitoring, including monitorable properties.

The online monitoring methodology (Figure 2 (b)) performs RV based on the golden model. The monitor can be non-intrusive hardware-based such as TAL hardware [5], NUVA hardware [5], or intrusive middleware. Finally, we enable the system to trigger recovery techniques when an emergent behavior is detected through our analysis.

A. 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 communication and synchronization between state machines in hardware and between hardware and software components. SAFER uses a conceptually simplified PSM model compared to [6]; the PSM model used in SAFER is defined below.

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.

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, $V_e \in EX$.

B. Runtime Monitorable Properties

Runtime verification handles verification techniques that can check whether the execution of the system being monitored satisfies or violates given properties [7]. Based on the PSM definition and architecture, SAFER can verify that the PSM specification is executing correctly by using RV to check the following properties at runtime:

- P1 Only one state can be executed in a PSM.
- P2 Export statement queues value of the variable into message queue.
- P3 Import statement waits until message queue has a relevant Export variable.
- P4 Execution time in a state in a PSM cannot exceed time defined in period p.
- P5 Each state will evaluate transition condition(s) every period p.
- P6 Each state in different PSMs can be executed simultaneously.
- P7 Execution of statement(s) in a state must be done within period p.
- P8 A state S must be moved to one of state(s) in $\delta(S,$ *) after transition, where * = any.
- P9 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. These properties allow SAFER to check all PSM-designs with the same set of properties, i.e., the deployed RV model checks only for the properties listed above. As a proof of concept, SAFER uses a Python framework called pyModelChecking[1] to verify the RV properties outlined above.

C. 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 only returns a random number and states add 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 full 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 their execution trace with its current state when that starts executing, as well as which state it's 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).

Finally, the global trace is also parsed to extract a stateful trace that contains current global state, next global state, execution time for 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

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

global trace is converted into a datapoint that has five sets of features: (1) the current global state, (2) next global state, (3) current state execution time, (4) time to transition, (5) and transition context (i.e., what system conditions are triggering the next transition).

D. Anomaly Detection Runtime Model

After collecting traces from the correct software generated, 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 trace we extracted from the implementation model.

The goal with the use of the autoencoder is to reduce the dimensionality from the stateful trace (which could vary based on the global state length) into 5 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 anomalous 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 anomalous.

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.

E. 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, effectively restarting the system to its starting state. 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 [8], which can increase the robustness of the SAFER implementation. However, these exceptional situations are topics for future research.

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

V. 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 critical unexpected behaviors. The examples were designed as minimal working examples that suffer from the specific weaknesses they are intended to illustrate. 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.

VI. Evaluation

We conducted five experiments for each case study, wherein 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. The corrupted data was injected directly into the AD supervisor, i.e., we did not observe what adverse situations could arise, instead we were interested if the AD supervisor could identify situations where an unexpected stateful trace was measured. However, these corruptions could indicate problems in the system, e.g., an imminent hazard [5]. Thus, 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 I shows the average F1 score results we obtained for the test data on each experiment. 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

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

TABLE I
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%

TABLE II
COMPARISON OF ANOMALY DETECTION F1 SCORES IN OTHER DOMAINS

Approach	Dataset	Supervised or Unsupervised	F1 Score
SVM [9]	KDD99	Supervised	79.11%
SVM with PCA [9]	KDD99	Combination of Both	90.51%
Robust Deep AEs [10]	MNIST	Semi-supervised	$\sim 75\%$
Outlier Detection [11]	Real-time network traffic	Unsupervised	29.82%
SVM [11]	Real-time network traffic	Supervised	57.25%
CkNN [11]	Real-time network traffic	Supervised	42.99%
VAE [4]	MNIST	Semi-supervised	$\sim 40\%$
SAFER	Real-time execution trace	Semi-supervised	83.82%

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%. It is also important to note that these results are only for the AD model. Since SAFER also has an RV model that can detect other property violations, this result shows that SAFER can detect up to 83% of the emergent behaviors that are not detectable with RV only.

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 II 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 [9] [10] 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 [11] achieves much lower results than our proposed approach.

VII. SAFER OVERHEAD

The SAFER methodology does incur overheads that need to be assessed in a cost-benefit analysis. Table III 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. All comparisons are done to a base binary that is generated from a PSM design and include only RV. As RV can be achieved via software, hardware, or a combination of both, with different levels of size and performance overheads [12], we chose to include it in the base binary and focus on the overhead added by the AD and recovery.

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 the restarting will reset variables to their start values.

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

TABLE III
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%

how long each of them took to finish. For the performance overhead, there is a bigger 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.

VIII. CONCLUSION

We presented the SAFER methodology to detect emergent (mis)behaviors to ensure the reliability and safety of complex systems. The emergent behaviors are unexpected operations that can still arise at runtime and lead the system to potentially unsafe conditions. To address this problem, we developed a formal model, the periodic state machine (PSM), and demonstrated how the model could be branched into runtime verification and anomaly detection methods. These two separate models can be integrated into one at the end to achieve emergent behavior detection. We illustrated SAFER's utility 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-art, with the advantage of being applicable to diverse systems. Lastly, due to its flexible overhead and the fact that AD models can detect emergent behaviors early, we believe that SAFER can work as a first line of safety checks for systems to prevent them from reaching critical states. In such cases, systems could still have more taxing last-case solutions in case an emergent behavior goes undetected by the AD and RV models.

SAFER lays the groundwork for a rich set of future work, including complex verification with anomaly detection via unmodified open-source projects that have been affected by Common Weakness Enumeration (CWE) issues, detection latency analysis, and hardware implementation for non-intrusive monitoring for detection.

ACKNOWLEDGMENTS

This work was partially supported by the following NSF grants: IPF grant CCF-1704859 and SARE EAGER grant ECCS-2028782.

REFERENCES

[1] J. C. Mogul, "Emergent (mis) behavior vs. complex software systems," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 293–304, 2006.

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

[3] C. C. Aggarwal, "Outlier analysis," in *Data mining*. Springer, 2015, pp. 237–263.

[4] J. An and S. Cho, "Variational autoencoder based anomaly detection using reconstruction probability," *Special Lecture on IE*, vol. 2, no. 1, pp. 1–18, 2015.

[5] 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)*. IEEE, 2019, pp. 1–10.

[6] 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*. IEEE Computer Society, 2007, pp. 10–20. [Online]. Available: https://doi.org/10.1109/ISORC.2007.47

[7] 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*, 2019, pp. 295–298.

[8] 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*, vol. 39, no. 11, pp. 3555–3565, 2020.

[9] A. George and A. Vidyapeetham, "Anomaly detection based on machine learning: dimensionality reduction using PCA and classification using SVM," *International Journal of Computer Applications*, vol. 47, no. 21, pp. 5–8, 2012.

[10] 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*, 2017, pp. 665–674.

[11] 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*, vol. 92, pp. 390–402, 2018.

[12] M. Seo and R. Lysecky, "Non-intrusive in-situ requirements monitoring of embedded system," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 23, no. 5, aug 2018. [Online]. Available: https://doi.org/10.1145/3206213

Caio Batista de Melo is an Assistant Teaching Professor in the Department of Computer Science at North Carolina State University. He received his Ph.D. in Computer Science from the University of California, Irvine in 2023. His research interests include reliable systems, emergent behavior reasoning, and computer science education.

Marzieh Ashrafiamiri earned her master's degree in Electrical Engineering and Computer Science from the Henry Samueli School of Engineering at UC Irvine. She is working as a software engineer at Snap, Inc. on the Content Understanding team. Her primary focus was on solving content-related issues in the 4th tab of the Snapchat app.

Minjun Seo earned a Ph.D. in Electrical and Computer Engineering from the University of Arizona in 2018. Currently, he is a Postdoctoral Researcher at UC Irvine's CECS. His research interests include embedded systems, computer architecture, runtime verification, and safety-critical systems.

Fadi Kurdahi is a Professor of EECS and CS, and the director of the Center for Embedded and Cyber-Physical Systems at UC Irvine. He received his PhD from USC in 1987 and currently conducts research in EDA, embedded and cyber-physical systems, and neuromorphic computing. He is a Fellow of the IEEE and the AAAS.

Nikil Dutt is a Distinguished Professor of CS, CogSci and EECS at UC Irvine, and received his PhD in CS from UIUC in 1989. His research interests are in embedded systems, EDA, computer architecture, healthcare IoT, and brain-inspired architectures and computing. He is an ACM and IEEE Fellow.