DELFASE : a Deep Learning Method for Fault Space Exploration

DELFASE: A Deep Learning Method for Fault Space Exploration

Ali Sedaghatbaf*, Mehrdad Moradi†, Jaafar Almasizadeh‡, Behrooz Sangchoolie*,
Bert Van Acker† and Joachim Denil†
*RISE Research Institutes of Sweden, Borås, Sweden
Email: {ali.sedaghatbaf, behrooz.sangchoolie}@ri.se
†University of Antwerp and Flanders Make, Belgium
Email: {mehrdad.moradi, bert.vanacker, joachim.denil}@uantwerpen.be
‡University of Isfahan, Isfahan, Iran
Email: j.almasizadeh@sci.ui.ac.ir

***Abstract*—Cyber-Physical Systems (CPSs) are increasingly used in various safety-critical domains; assuring the safety of these systems is of paramount importance. Fault Injection is known as an effective testing method for analyzing the safety of CPSs. However, the total number of faults to be injected in a CPS to explore the entire fault space is normally large and the limited budget for testing forces testers to limit the number of faults injected by e.g., random sampling of the space. In this paper, we propose DELFASE as an automated solution for fault space exploration that relies on *Generative Adversarial Networks* (GANs) for optimizing the identification of critical faults, and can run in two modes: *active* and *passive*. In the active mode, an active learning technique called ranked batch-mode sampling is used to select faults for training the GAN model with, while in the passive mode those faults are selected randomly. The results of our experiments on an adaptive cruise control system show that compared to random sampling, DELFASE is significantly more effective in revealing system weaknesses. In fact, we observed that compared to random sampling that resulted in a fault coverage of around 10%, when using the active and passive modes, the fault coverage of DELFASE could be as high as 89% and 81%, respectively.**

***Index Terms*—Fault injection, cyber-physical systems, generative adversarial networks, safety assessment.**

I. Introduction

Cyber-physical systems (CPSs) are complex systems that integrate cyber components (i.e., computing hardware, network hardware, and software) with mechanical components (e.g., sensors and actuators). CPSs have proven to be vital in a range of safety-critical domains, including health-care, smart grids, aerospace, energy and transportation [1]. Due to the increasing application of CPSs, their safety has attracted a lot of attention in the computing community. Over the last years, a variety of testing methods have been developed for analyzing the safety of CPSs. The safety assurance of these systems is challenging due to reasons such as the high connectivity and heterogeneity of these systems, the dynamicity of their environments and several real-time constraints that they need to satisfy.

An effective testing technique that facilitates safety assessment of CPSs is fault Injection (FI). With the help of FI, one can evaluate the safety of a CPS by accelerating the occurrence of possible faults [2]. Three attributes are typically considered for every injected fault, namely fault type/model, location of injection, and fault activation time [3]. Each attribute might be supplied with one or more parameters that should be evaluated. Typically, several values could be selected for evaluating each of these parameters whose combination leads to an exponential growth of the fault space size.

Traditional FI methods rely on expert knowledge and historical data from system failures to decide on the type of faults that should be injected as well as their location and timing [4]. We in this paper are, however, interested in identification of critical faults through automatic exploration of the fault space using Machine Learning (ML). Note that, with critical faults, we refer to those that reveal system weaknesses leading to the violation of safety requirements. The method we propose for exploration of the fault space is based on supervised learning. This method establishes a model between the injected faults and the execution behavior of the System Under Test (SUT). This model represents the knowledge learned automatically through interaction with the SUT, and helps us locate more critical faults with less effort. Note that, effective exploration of the fault space has also been studied in the past using other techniques such as those that are deterministic [5], [6] or model-based [7], as well as those that are based on evolutionary optimization [8] or reinforcement learning [9].

In this paper, we propose to apply Generative Adversarial Networks (GANs) [10] to the problem of fault space exploration. GANs have been successfully used in several applications (e.g., image generation [11], anomaly detection [12] and performance testing [13]). However, *to the best of our knowledge, this work demonstrates the first application of GANs in fault space exploration*.

Using GANs, we take advantage of a generator model for intelligent fault generation and a discriminator model to predict the impact of each generated fault on safety requirements, based on the knowledge acquired through interactions with the SUT. Following this approach, we are able to identify several critical faults without the need to do an exhaustive search in the fault space and examine the impact of all candidate faults on an SUT. The proposed solution, which is named DELFASE (a DEep Learning method for FAult Space Exploration), trains

a GAN model in an online manner with no need for a pre-existing training dataset. DELFASE can work in two modes: *active* and *passive*. In the active mode, ranked batch mode sampling [14] as an Active Learning (AL) technique is utilized to build the training dataset in an intelligent way so that the GAN model learns more about the fault space with less training data. In the passive mode, the faults to be included in the training dataset are selected randomly.

To evaluate DELFASE, we consider an Adaptive Cruise Control (ACC) with sensor fusion as the SUT. The purpose of ACC is to keep a safe distance between two cars. We have performed FI experiments on the MATLAb/Simulink model of this system with the goal of measuring fault coverage (FC) indicating the percentage of critical faults; these faults are those that lead to an accident between the two cars. We limited our experiment to a two-car scenario, while earlier studies have shown that the entire traffic might be affected by a fault in a single car [15]. The results of our experiments show that after a few iterations, DELFASE identifies a higher number of critical faults compared to when the fault space is randomly sampled. Furthermore, the results confirm that when using the active mode, DELFASE achieves a desired FC faster than when the passive mode is used. This, however, comes with a higher overhead of model training, which is around 0.4 seconds for each training step (or epoch). In summary, this paper makes the following contributions:

- Introduces DELFASE as an automated ML solution for fault space exploration when conducting FI campaigns (see Section V).
- Implements two modes of operation for DELFASE, namely the *active* and *passive* modes where the former uses ranked batch mode sampling to train the dataset [14], while the latter includes faults in the training dataset randomly.
- Explores the fault space using the active and passive modes of DELFASE and conducts a comparison of the results obtained for these modes with when the fault space is randomly sampled with respect to *fault coverage*, *labeling effort*, and *execution overhead* (see Section VI-B).

II. Background

A. Fault Injection

Fault Injection (FI) is a testing method suitable for observing systems' behavior under small perturbations [2]. In this method, engineers manipulate a real or a virtual system to make it fail in order to test the system's robustness or analyze the system's dependability attributes such as safety. In FI, there are typically three attributes that would need to be specified for each fault. These attributes are (1) fault type/model, (2) fault activation time, and (3) fault location [3]. Some typical fault models are as follows:

- *Data fault*: In this model, the data transmitted between the SUT components are manipulated. For example, we can inject data faults by manipulating sensor data. In real world, sensor data could change due to faults in sensors or changes in the external environment (e.g., rainy or foggy weather).
- *Hardware fault*: Bit-flip and stuck-at-value are examples of hardware faults. In the bit-flip model, one or several bits of the value stored in a location (e.g., a CPU register) are flipped. In the stuck-at-value model, a signal/pin is tied to a specific value.
- *Timing fault*: In this model, the transmission of data between two components of the SUT is delayed. This delay might lead to loss or out of order delivery of data.

FI techniques can be categorized into three groups [16]: model-implemented [17], hardware-implemented [18] and software-implemented [19]. In this paper, we focus on model-implemented FI where faults are injected into software/hardware models as opposed to the other FI categories which target software/hardware implementations. We inject faults into a MATLAB/Simulink model of an SUT. Model-implemented FI eases the process for test engineers as the evaluation is performed in a higher abstraction level without the need to do costly experiments on the real system. In fact, using model-implemented FI, we can conduct thousands of FI experiments in a short period of time. Furthermore, we can apply this method in the early development stages when the system is not implemented yet or is incomplete. However, the accuracy of the experimental results would be closely connected to the accuracy of the model.

B. Generative Adversarial Networks

Generative Adversarial Networks (GANs) [10] are generative models that can learn the distribution of input data to generate realistic data. In a typical GAN architecture (see Figure 1), there are two models: a generator and a discriminator. The generator model is trained to generate dummy data, and the discriminator model is trained to discriminate dummy data from real data. In other words, the generator model and the discriminator model compete with each other in a game during model training. In such a game, the generator constantly learns to generate convincing examples to fool the discriminator such that it cannot distinguish them from real data. Simultaneously, the discriminator is trained by real and dummy data, and constantly learns to identify real examples from dummy ones. The ultimate goal of the game is to make the generator so powerful that it can generate dummy data that the discriminator cannot distinguish from real data.

C. Active Learning

Passive learning is the typical learning scenario followed in most supervised learning applications. In this scenario, a labeled dataset is provided in advance for an ML model to learn from. However, in Active Learning (AL), we let the ML model decide which data to learn from [20]. AL is suitable for cases that data labeling is expensive. For instance, for training deep learning models (e.g., the GAN model in this paper) we usually need a large labeled dataset. Experimental studies in several domains e.g., intrusion detection [21], and natural language processing [22] confirm the effectiveness of AL in

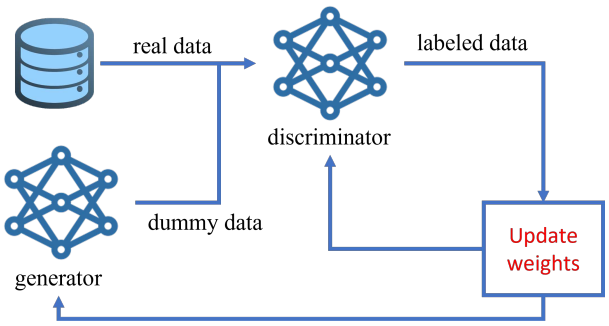


Fig. 1: A typical GAN model

reducing the cost of data labeling. In this paper, we introduce the first application of AL to the FI domain and demonstrate how its integration with GANs can reduce the cost of fault labeling. Note that, we assign a binary label to each injected fault, which indicates whether the fault is critical (i.e., leads to safety failure) or not.

Now, the question is how data is labeled in AL. *Membership query synthesis* and *pool-based sampling* are examples of data-labeling scenarios typically followed in AL [20]. In the first scenario, the ML model generates data instances by sampling from an underlying distribution. However, in the latter, the learner uses a query strategy to select data instances from a pool of observed data. *Least Confidence* (LC) is an example of such query strategies [20], where instances for which the ML model is least confident in label assignment are selected for labeling. In this paper, we use a combination of membership query synthesis and pool-based sampling for data labeling. In particular, we take advantage of GANs to synthesize new faults (or queries in AL terminology) and use ranked batch-mode sampling [14] as a novel variant of pool-based sampling to prioritize them for injection. Ranked batch-mode sampling has two advantages over traditional pool-based sampling methods: (1) it allows us to select more than one instance from the pool in each iteration, and (2) its ranking mechanism helps us avoid injecting redundant faults. The query strategy that we use in the above labeling scenario is LC.

III. Related Work

FaultCheck [23] is a FI tool that facilitates generation of fault models by a property-based testing tool called QuickCheck. By integrating these two tools, the authors of the work show how in addition to normal circumstances, we can validate safety requirements in circumstances that certain faults are present in the SUT. MODIFI [16] is another FI tool which is suitable for injecting faults into Simulink models. This tool receives as input the fault model and can inject a user-defined set of faults and cybersecurity attacks [24] into the SUT both sequentially and concurrently. As another Simulink-based tool, ErrorSim [25] allows injecting different types of faults (e.g., hardware fault, network fault, sensor fault, etc.) into Simulink models, specifying their occurrence and duration, and analyzing the propagation of errors throughout the models. In [26], the authors combine fault trees with simulation-based FI to analyze the propagation of failures throughout the SUT. Hereby, the faults to be injected are automatically extracted from fault trees. Fault trees contain expert knowledge about SUT or heuristics on systems failure. Including this knowledge in FI would help fault injector to analyze frequent failures faster and assure that all known failures have been covered in the testing process.

Despite being useful, the above contributions do not address the challenge of fault space exploration. In other words, the set of faults to be injected is either predefined or selected by random sampling. Sangchoolie et al. [6], [27] propose to use prior knowledge about the outcome of bit-flip faults to prune the fault space. In particular, they observed that the impact of injecting faults into certain bit positions could be identified a priori, hence no need to inject faults into those locations. SEInjector [28] is a FI tool for analyzing transient faults. This tool ignores the faults whose outcome is known without the need for injection. To further accelerate fault space exploration, SEInjector classifies faults based on their outcomes, so that the faults with similar outcomes go to the same equivalence class. Then, from each class only one instance would be injected into the SUT.

In DriveFI [29] Bayesian Networks (BNs) are used to find critical faults that may lead to safety hazards. The authors use domain knowledge and the safety model of the SUT to build a BN. The BN built is then analyzed to identify critical faults. The experimental results show that using BNs, more critical faults can be found within a shorter period of time compared to random sampling. Both SEInjector and DriveFI rely on domain knowledge about the SUT to prune the fault space. Maldini et al. [8] propose an evolutionary algorithm for optimizing the search in the fault space and use the algorithm to attack a cryptography algorithm. The proposed algorithm relies less on prior knowledge and is more suitable for black-box testing scenarios. However, the algorithm supports only discrete fault spaces. So, if we have continuous-value fault parameters, we have to discretize them first. DELFASE can work with both discrete and continuous fault data.

As an example of ML solutions, reinforcement learning is used for fault space exploration [30]. The solution has a better performance compared to random sampling. However, as emphasized by the authors, defining an appropriate reward function is a significant challenge when using reinforcement learning for fault space exploration. In this paper, we propose a supervised learning solution for fault space exploration which takes advantage of AL to reduce the labeling cost, without the need to manually define a suitable reward function.

IV. System Under Test

In this paper, we use an Adaptive Cruise Controller (ACC) as the SUT. ACC is a driving assistance system that can be used to regulate the speed of a car and maintain a safe distance from cars ahead. This system prevents the controlled car, i.e., ego car, from getting too close to the car in front, i.e., lead car. ACC is equipped with both vision and radar sensors to

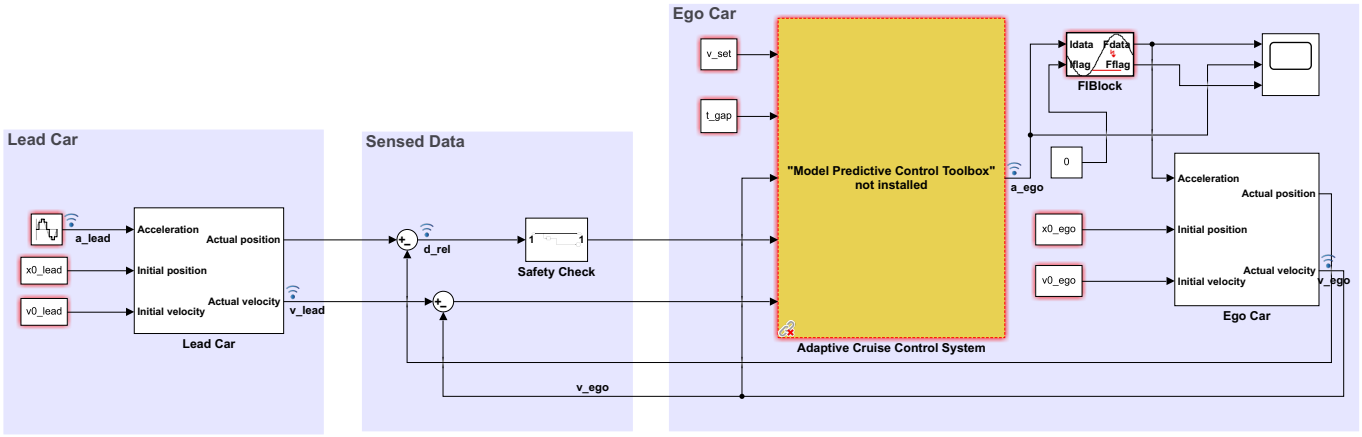


Fig. 2: The Simulink model of the ACC with a fault injector

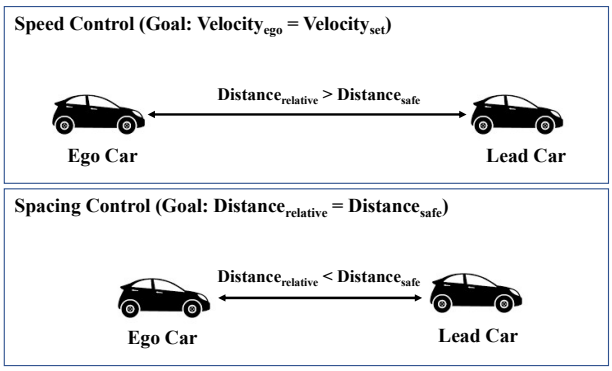


Fig. 3: The operation modes of ACC

detect the lead car, and to measure the position and velocity of the cars accurately.

The Simulink model of ACC is shown in Figure 2. This model is an extension of the original model [31]. The model includes three main modules. The first module (named *Adaptive Cruise Control System*) models the ACC functionality which is to control and generate the acceleration of the ego car. The other two modules (named *Ego Car* and *Lead Car*) model the car dynamics, steering wheel controller, roads and environment actors for the ego and lead cars. The sensor data synthesized by these modules are sent to the first module to determine the appropriate acceleration for the ego car. As shown in Figure 3, ACC works in two modes depending on the relative distance between the cars. In the *speed control* mode, ACC makes the ego car travel at the driver-set speed, but it will switch to the *spacing control* mode whenever the cars get too close. In this mode, ACC reduces the speed of the ego car until there is a safe distance between the cars.

ACC is a *safety-critical* system, since any malfunction of ACC can endanger environment or human life. Therefore, the safety of this system should be analysed carefully. To this end, a block for checking safety requirements ((*Safety Check* in Figure 2)) is added to the original ACC model. In this paper, we consider one safety requirement for the ACC as defined by Equation 1. This requirement specifies $distance_{unsafe}$ as a lower bound for the distance between the two cars. This lower bound is $4m$ (length of a car in the simulation) [32], and if the relative distance between the two cars gets less than this bound, the block sets a requirement violation flag and stops the simulation immediately.

$$distance_{relative} \geq distance_{unsafe} \tag{1}$$

In addition to *Safety Check*, we have added *FIBlock* to the original ACC model to inject faults into the acceleration signal transferred between ACC and the ego car. This block is an instance of the Fault Injection Block introduced in [7] as a MATLAB extension. This extension supports six common types of faults (i.e., stuck-at-value, packet-loss, bias/offset, bit-flip, delay and noise), and has the following configuration parameters [7]:

1) *fault type*: the type/model of the fault being injected (e.g., stuck-at-value), which may be supplied with a seed value. For example, an offset fault requires a seed value which specifies the magnitude of the offset.
2) *fault event*: how the fault occurs (e.g., probabilistic) and the corresponding seed value. For example, if the fault event is deterministic, this value specifies the point of time for fault injection.
3) *fault effect*: how long the fault will affect the SUT (e.g., infinite) and a seed value for specifying the duration. For example, if the fault affects the SUT for a constant period of time, then this value specifies the length of the time interval the fault will persist in the SUT.

In the Simulink model shown in Figure 2, *FIBlock* is configured to inject bias/offset faults at deterministic points of time and with a constant duration. However, the magnitude of offset and the exact time/duration of injection are the fault parameters to be explored by DELFASE (see Section VI).

V. Proposed Method

This section presents an overview of DELFASE as an ML method for fault space exploration. As shown in Figure 4, fault

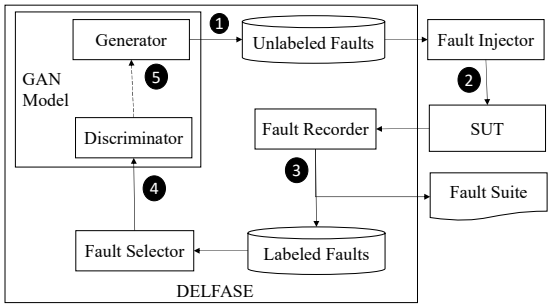


Fig. 4: Overview of DELFASE

space exploration in DELFASE is an online learning process which includes the following steps:

1) **Fault Synthesis**: *Generator* synthesizes a pool of unlabeled faults. For example, considering the example in Section IV, each synthesized fault includes three features corresponding to the three value parameters of *FIBlock1*. The seed values provided for these parameters are numeric and can be initialized automatically e.g., by an ML model.
2) **Fault Injection**: *Fault Injector* iteratively takes a fault out of the pool received from *Generator* and injects it into the SUT. After each injection, it waits until a feedback is received from the SUT, and then injects the next fault. The feedback indicates whether the fault was critical (i.e., led to the violation of a safety requirement) or not.
3) **Fault Recording**: After injection of faults and receiving the SUT outputs for the whole fault pool, *Fault Recorder* divides the injected faults into two groups according to the injection outcomes: (1) critical faults (or real data in GANs terminology) which led to the violation of safety requirements (e.g., the one expressed by Equation 1 for the ACC example), and (2) non-critical faults (or fake data in GANs terminology). *Fault Recorder* assigns the labels 1 and 0 to critical and non-critical faults, respectively, and records the critical faults in the fault suite. This component stops the execution of DELFASE if the size of the fault suite has reached the limit specified by the user. Otherwise, it records the labeled faults in the labelled dataset that will be used for training the GAN model in step 5.
4) **Fault Selection**: In this step, a batch of faults are selected by *Fault Selector* for training the GAN model. In the passive mode, *Fault Selector* selects a random batch of faults from the pool of faults labeled by *Fault Recorder*. However, in the active mode, this component uses ranked batch-mode sampling together with the LC query strategy (see Section II-C for details about active learning) to rank the labeled faults synthesized by *Generator*, and then selects the superior ones for model training. Following the ranked batch mode sampling algorithm [14], *Fault Selector* takes the following steps to rank the synthesized faults: (a) asks *Discriminator* to predict the label of each fault, (b) uses the LC strategy to estimate an uncertainty score for each fault based on the label predicted by *Discriminator*, (c) uses Euclidean distance to estimate the dissimilarity of each fault to the other faults stored in the labeled dataset, and (d) ranks the faults by evaluating a weighted sum of the uncertainty and the lowest dissimilarity estimated for each one. According to the ranked batch mode sampling algorithm [14], for the initial iterations that we have fewer labeled faults, a higher weight would be assigned to the dissimilarity score (to increase diversity among the labeling candidates). However, in latter iterations, uncertainty gets a higher weight.
5) **Model Training**: The GAN model is trained in two steps. First, *Discriminator* is trained using the faults selected by *Fault Selector*. Then, the weights of *Generator* are updated based on the knowledge learned by *Discriminator*. In particular, the parameters of *Discriminator* will be frozen in the second step, and the whole GAN model will be trained to update the weights of *Generator*. After training the GAN model, execution of DELFASE moves to Step 1.

TABLE I: Configuration of FIBlock

Parameter	Type	Value
fault type	offset	[0,7]
fault event	deterministic	[0,7]
fault effect	constant	[0,7]

VI. Evaluation

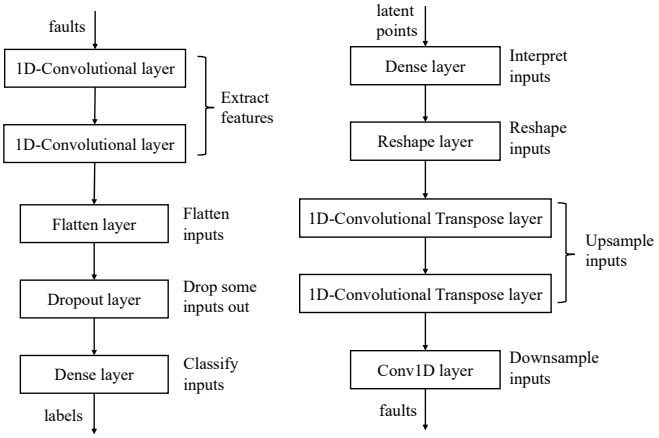
In this Section, we provide details about the implementation of DELFASE, and present the results of the fault injection experiments performed. The ACC elaborated in Section IV is used as the SUT, and the Python engine of MATLAB is used to facilitate interactions between DELFASE and the MATLAB script running the Simulink model of ACC.

A. Experimental Setup

We have implemented DELFASE using Python 3.8 and TensorFlow platform [33] (the source code is available online [1]). We utilized Keras [34] as a practical ML library to implement the GAN model and took advantage of the implementation provided in modAL [35] for the AL technique used in DELFASE (i.e., ranked batch-mode sampling).

Figure 5 shows the architectures of the GAN model in DELFASE. In this model, *Discriminator* takes as input a fault and outputs a binary value indicating whether the fault is critical ($label = 1$) or non-critical ($label = 0$). For feature extraction, *Discriminator* includes two one-dimensional convolution layers with 64 neurons, a kernel size of 4, and LeakyReLU [36] as the activation function. The extracted features are then used by the next three layers for classification. The classification task is performed by the last layer which is a fully connected (or Dense in Keras terminology) layer with

[1] https://github.com/alisedaghatbaf/DELFASE



(a) Discriminator model (b) Generator model

Fig. 5: Architecture of ML models in DELFASE

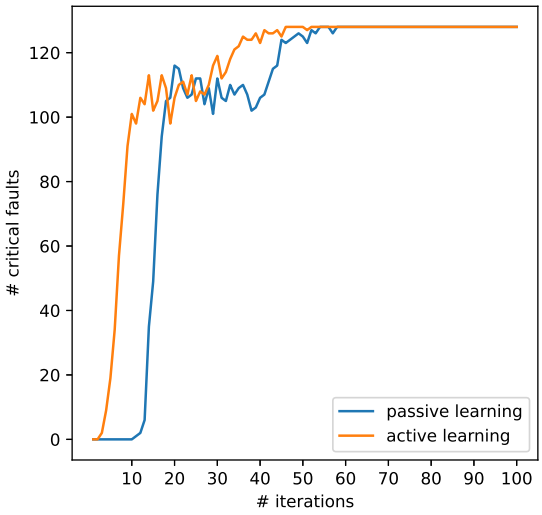


Fig. 6: Performance of DELFASE

one neuron and a Sigmoid [37] activation function. However, since that layer takes only 1-dimensional inputs, we need to reduce the dimensionality of the feature vectors using a Flatten layer. In particular, each feature vector is 2-dimensional where time is one of the dimensions, and the Flatten layer removes this dimension. Finally, the purpose of adding a Dropout layer between the last two layers is to randomly drop out a ratio of neurons (i.e., 0.4) during training, and thereby avoid overfitting [38]. For training, *Discriminator* relies on binary cross-entropy [39] as the loss function and RMSprop [40] as the optimizer.

Generator on the other hand, takes as input a point in the latent space and generates a point in the fault space. Latent space is a 100-sphere where each of the 100 variables is drawn from a Gaussian distribution $G(0,1)$. We consider latent points as abstract representations of faults. Therefore, in the first layer of *Generator* shown in Figure 5, we use a fully connected layer to interpret the input latent points. This layer has 3×128 neurons, where 3 is the number of fault parameters (see Table I). In the second layer, we reshape the outputs of the first layer to fit the dimensionality of the fault space. Then we upsample them via two deconvolution (or transposed convolution) layers to make each point four times the size of a point in the fault space. For each deconvolution layer, the kernel size is 4 and the number of neurons is 128. Finally, we generate faults by downsampling those points using a convolution layer, which has only one neuron and a kernel size of 4, and uses hyperbolic tangent as the activation function. The GAN model is formed by putting *Generator* on top of *Discriminator*. For training Generator, the GAN model relies on RMSprop and binary cross-entropy as the optimizer and loss function, respectively.

B. Experimental Results

We ran our experiments on a computer with an AMD Ryzen 7 (3.8 GHz) processor. In these experiments we configured *FIBlock1* such that it would inject offset faults, in a deterministic time during the simulation, and for a constant period of time, and assumed that the seeds of those parameters would take any floating point value between zero and seven (see Table I). Note that, the motivation for choosing seven as the upper bound for the fault parameters was the observation that for the majority of the values higher than seven, the safety requirement expressed by Equation 1 would be violated. In particular, we injected 100 faults with at least two values higher than seven and 87 of them were found to be critical.

We ran DELFASE for 100 iterations assuming an infinite size for the fault suite. In each iteration, Generator was asked to synthesize a fault pool of size 128 such that 32 of them were selected by Fault Selector for training the GAN model in each iteration. Figure 6 shows how many of the faults synthesized in each iteration were found to be critical after injection. The figure shows that DELFASE can explore the fault space and generate many critical faults after a few iterations. Furthermore, the figure shows that the active mode would require fewer iterations, when compared to the passive mode, to learn the fault space and generate a high ratio of critical faults.

To investigate the effectiveness of DELFASE, we performed a comparison with uniform random sampling. We used $random.uniform()$ function of Python to generate random values between zero and seven for each fault parameter and injected the generated faults to the SUT and used Fault Recorder for labeling them. We repeated random fault generation for 100 iterations such that similar to the experiment on DELFASE, 128 faults were generated in each iteration, and we recorded the number of critical faults generated in each iteration. The results are summarized in Table II. The table shows that, the FC of random sampling is around 10% irrespective of the fault suite size. However, after 100 training iterations, the FC of DELFASE rises up to 81% and 89% for the passive and active modes, respectively. These results also indicate that for very small fault suites, random sampling has a better FC compared

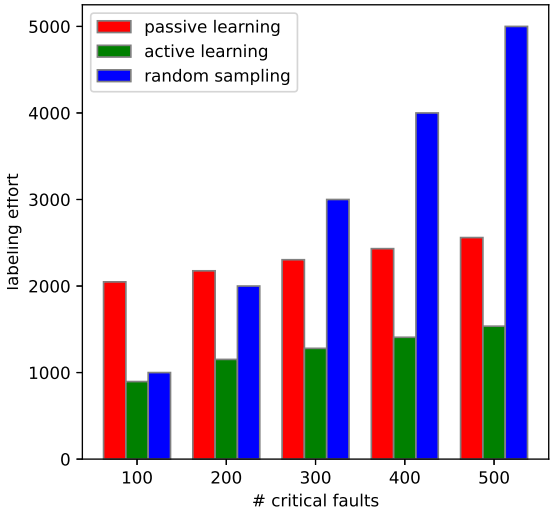


Fig. 7: Labeling effort of DELFASE and random sampling

to passive DELFASE.

Figure 7 compares DELFASE with random sampling from the labeling effort (i.e., the number of faults labeled) perspective. In this regard, for small fault suites with less than 300 critical faults, the labeling effort of passive DELFASE is comparable with random sampling. However, the effort of random sampling can be much higher for bigger fault suits. For example, since the FC of random sampling is around 10% (see Table II), for a fault suite of size 1000, we need to label around 10000 faults. which is a considerable effort. These results also highlight the low labeling effort of active DELFASE compared to both passive DELFASE and random sampling.

From the execution time (overhead) perspective, the results presented in Table II indicate that active DELFASE results in the least overhead. This is an interesting conclusion as before conducting the experiments, one could expect that DELFASE would require a higher execution time in the active mode compared to the passive mode and random sampling, considering the overhead of model training, and the fault selection algorithm. To investigate this further, we measured the average execution time of the system under test (SUT). While for non-critical faults, each run of the SUT takes around 2.4 seconds, our investigations revealed lower execution times for critical faults due to the fact that the *Safety Check* ends the simulation as soon as the safety requirement expressed by Equation 1 gets violated (see Section IV for more details). Therefore, higher FC implies an overall lower execution time of the Simulink model. Furthermore, to analyze the impact of the time it takes to train the models on the overall overhead, we measured the average execution time of each model training step of DELFASE (i.e., step 5 in Section V) and found it to be approximately 1 and 1.4 seconds for the passive and active modes, respectively. As mentioned above, 128 faults were labeled in each iteration of DELFASE, which means that an approximate overhead of $2.4 \times 128 = 307.2$ seconds for simulation run in each iteration could be expected (in the worst case where none of the injected faults are found to be critical). Therefore, it is easy to notice the low impact of model training on the overall execution time, compared to fault injection and simulation run.

VII. Conclusion and Future Work

In this paper, we introduced DELFASE as a supervised learning solution for fault space exploration. This solution takes advantage of Generative Adversarial Networks and an active learning technique called ranked batch-mode sampling. Effective identification of critical faults (faults that lead to safety issues) with low test budget is the main challenge addressed by this solution. Here, by test budget we mean the number of faults that should be injected into the SUT in order to identify critical ones.

Our experiment on the Simulink model of an adaptive cruise control system (as a safety-critical cyber-physical system) confirms that the proposed solution can identify more critical faults with less test effort compared to random sampling. Furthermore, we found that using active learning would lead to faster learning and less execution overhead compared to passive learning where the machine learning model does to take part in the selection of training data. However, our experiments were focused only on one fault model and as part of our future work, we plan on conducting experiments using other fault models. Furthermore, we plan on conducting experiments on more complex traffic scenarios as well as investigating the impact of a fault on other cars in the traffic.

TABLE II: Results of the experiments

Iterations	Generated Faults	DELFASE (passive learning)			DELFASE (active learning)			Random Sampling		
		Critical Faults	FC(%)	Time(s)	Critical Faults	FC(%)	Time(s)	Critical Faults	FC(%)	Time(s)
10	1280	0	0.0	3146.3	386	30.1	3017.2	135	10.5	3088.7
20	2560	590	23	6085.5	1440	56.2	5786.3	254	9.9	6183.1
30	3840	1677	43.7	8851	2546	66.3	8558.9	389	10.1	9271.8
40	5120	2742	53.5	11624.3	3755	73.3	11285.7	524	10.2	12360.6
50	6400	3938	61.5	14351.7	5026	78.5	13990.9	671	10.5	15445.1
60	7680	5208	67.8	17053.2	6305	82.1	16693.2	772	10	18545.8
70	8960	6488	72.4	19751.2	7585	84.6	19492.2	901	10	21636.6
80	10240	7768	75.8	22449.2	8865	86.6	22097.2	1035	10.1	24725.7
90	11520	9048	78.5	25147.2	10145	88.1	24799.2	1170	10.1	27814.5
100	12800	10328	80.7	27845.2	11425	89.2	27501.2	1305	10.2	30903.2

ACKNOWLEDGMENT

This work was partially funded by Flanders Make vzw research centre, and the European IVVES (grant agreement No 18022) and VALU3S (grant agreement No 876852) projects.

REFERENCES

[1] E. A. Lee, "Cyber physical systems: Design challenges," in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, 2008, pp. 363–369.

[2] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: a methodology and some applications," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166–182, 1990.

[3] A. Benso and P. Prinetto, *Fault injection techniques and tools for embedded systems reliability evaluation*. Springer Science & Business Media, 2003, vol. 23.

[4] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[5] J. Fröhlich, J. Frtunikj, S. Rothbauer, and C. Stückjürgen, "Testing safety properties of cyber-physical systems with non-intrusive fault injection–an industrial case study," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2016, pp. 105–117.

[6] B. Sangchoolie, R. Johansson, and J. Karlsson, "Light-weight techniques for improving the controllability and efficiency of isa-level fault injection tools," in *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2017, pp. 68–77.

[7] T. Fabarisov, I. Mamaev, A. Morozov, and K. Janschek, "Model-based fault injection experiments for the safety analysis of exoskeleton system," *arXiv preprint arXiv:2101.01283*, 2021.

[8] A. Maldini, N. Samwel, S. Picek, and L. Batina, "Optimizing electromagnetic fault injection with genetic algorithms," in *Automated Methods in Cryptographic Fault Analysis*. Springer, 2019, pp. 281–300.

[9] M. Moradi, B. J. Oakes, M. Saraoglu, A. Morozov, K. Janschek, and J. Denil, "Exploring fault parameter space using reinforcement learning-based fault injection," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2020, pp. 102–109.

[10] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *arXiv preprint arXiv:1406.2661*, 2014.

[11] H. Huang, P. S. Yu, and C. Wang, "An introduction to image synthesis with generative adversarial nets," *arXiv preprint arXiv:1803.04469*, 2018.

[12] M. Ravanbakhsh, M. Nabi, E. Sangineto, L. Marcenaro, C. Regazzoni, and N. Sebe, "Abnormal event detection in videos using generative adversarial nets," in *2017 IEEE International Conference on Image Processing (ICIP)*. IEEE, 2017, pp. 1577–1581.

[13] A. Sedaghatbaf, M. H. Moghadam, and M. Saadatmand, "Automated performance testing based on active deep learning," *arXiv preprint arXiv:2104.02102*, 2021.

[14] T. N. Cardoso, R. M. Silva, S. Canuto, M. M. Moro, and M. A. Gonçalves, "Ranked batch-mode active learning," *Information Sciences*, vol. 379, pp. 313–337, 2017.

[15] M. Maleki and B. Sangchoolie, "Sufi: A simulation-based fault injection tool for safety evaluation of advanced driver assistance systems modelled in sumo," in *2021 17th European Dependable Computing Conference (EDCC)*, 2021, pp. 45–52.

[16] R. Svenningsson, J. Vinter, H. Eriksson, and M. Törngren, "Modifi: a model-implemented fault injection tool," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2010, pp. 210–222.

[17] R. Svenningsson, H. Eriksson, J. Vinter, and M. Törngren, "Model-implemented fault injection for hardware fault simulation," in *2010 Workshop on Model-Driven Engineering, Verification, and Validation*. IEEE, 2010, pp. 31–36.

[18] L. Antoni, R. Leveugle, and M. Feher, "Using run-time reconfiguration for fault injection in hardware prototypes," in *17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings.* IEEE, 2002, pp. 245–253.

[19] J. Sosnowski, A. Lesiak, P. Gawkowski, and P. Wlodawiec, "Software implemented fault inserters," *IFAC Proceedings Volumes*, vol. 36, no. 1, pp. 293–298, 2003.

[20] B. Settles, "Active learning literature survey," 2009.

[21] K. Yang, J. Ren, Y. Zhu, and W. Zhang, "Active learning for wireless iot intrusion detection," *IEEE Wireless Communications*, vol. 25, no. 6, pp. 19–25, 2018.

[22] Y. Shen, H. Yun, Z. C. Lipton, Y. Kronrod, and A. Anandkumar, "Deep active learning for named entity recognition," *arXiv preprint arXiv:1707.05928*, 2017.

[23] B. Vedder, T. Arts, J. Vinter, and M. Jonsson, "Combining fault-injection with property-based testing," in *Proceedings of International Workshop on Engineering Simulations for Cyber-Physical Systems*, 2013, pp. 1–8.

[24] B. Sangchoolie, P. Folkesson, and J. Vinter, "A study of the interplay between safety and security using model-implemented fault injection," in *2018 14th European Dependable Computing Conference (EDCC)*, 2018, pp. 41–48.

[25] M. Saraoğlu, A. Morozov, M. T. Söylemez, and K. Janschek, "Errorsim: A tool for error propagation analysis of simulink models," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2017, pp. 245–254.

[26] S. Reiter, M. Zeller, K. Höfig, A. Viehl, O. Bringmann, and W. Rosenstiel, "Verification of component fault trees using error effect simulations," pp. 212–226, 2017.

[27] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "An empirical study of the impact of single and multiple bit-flip errors in programs," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2020.

[28] X. Meng, Q. Tan, Z. Shao, N. Zhang, J. Xu, and H. Zhang, "Optimization methods for the fault injection tool seinjector," in *2018 International Conference on Information and Computer Technologies (ICICT)*, 2018, pp. 31–35.

[29] S. Jha, S. Banerjee, T. Tsai, S. K. Hari, M. B. Sullivan, Z. T. Kalbarczyk, S. W. Keckler, and R. K. Iyer, "Ml-based fault injection for autonomous vehicles: A case for Bayesian fault injection," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2019, pp. 112–124.

[30] M. Moradi, B. Oakes, and J. Denil, "Machine learning-assisted fault injection," in *39th International Conference on Computer Safety, reliability and Security (SAFECOMP), Position Paper, Lisbon, Portugal*, 2020.

[31] "Adaptive cruise control with sensor fusion," https://nl.mathworks.com/help/driving/ug/adaptive-cruise-control-with-sensor-fusion.html, accessed: 2021-06-01.

[32] G. Coley, A. Wesley, N. Reed, and I. Parry, "Driver reaction times to familiar, but unexpected events," *TRL Published Project Report*, 2009.

[33] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[34] A. Gulli and S. Pal, *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[35] T. Danka and P. Horvath, "modal: A modular active learning framework for python," *arXiv preprint arXiv:1805.00979*, 2018.

[36] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, no. 1. Citeseer, 2013, p. 3.

[37] J. Han and C. Moraga, "The influence of the sigmoid function parameters on the speed of backpropagation learning," in *International workshop on artificial neural networks*. Springer, 1995, pp. 195–201.

[38] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[39] P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, "A tutorial on the cross-entropy method," *Annals of operations research*, vol. 134, no. 1, pp. 19–67, 2005.

[40] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.