ASSURING THE ROBUSTNESS AND RESILIENCY OF LEARNING-ENABLED AUTONOMOUS SYSTEMS

By

Michael Austin Langford

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science – Doctor of Philosophy

2022

ABSTRACT

ASSURING THE ROBUSTNESS AND RESILIENCY OF LEARNING-ENABLED AUTONOMOUS SYSTEMS

By

Michael Austin Langford

As Learning-Enabled Systems (LESs) have become more prevalent in safety-critical applications, addressing the assurance of LESs has become increasingly important. Because machine learning models in LESs are not explicitly programmed like traditional software, developers typically have less direct control over the inferences learned by LESs, relying instead on semantically valid and complete patterns to be extracted from the system's exposure to the environment. As such, the behavior of an LES is strongly dependent on the quality of its training experience. However, run-time environments are often noisy or not well-defined. Uncertainty in the behavior of an LES can arise when there is inadequate coverage of relevant training/test cases (e.g., corner cases). It is challenging to assure safety-critical LESs will perform as expected when exposed to run-time conditions that have never been experienced during training or validation. This doctoral research contributes automated methods to improve the *robustness* and *resilience* of an LES. For this work, a robust LES is less sensitive to noise in the environment, and a resilient LES is able to self-adapt to adverse run-time contexts in order to mitigate system failure. The proposed methods harness diversity-driven evolution-based methods, machine learning, and software assurance cases to train robust LESs, uncover robust system configurations, and foster resiliency through self-adaptation and predictive behavior modeling. This doctoral work demonstrates these capabilities by applying the proposed framework to deep learning and autonomous cyber-physical systems.

To Jinny Shows, who kept me sane throughout this process and fed when I was hungry.

ACKNOWLEDGEMENTS

This dissertation would not be possible without the support and assistance of others, whom I appreciate a great deal.

Many thanks go to my advisor, Dr. Betty H.C. Cheng, who has always been a welcoming presence from the first day I stumbled onto campus. I appreciate our many brainstorming sessions, the tangents we explored, and her reliably cheerful attitude, even during a global pandemic and complete social isolation. Even more, I appreciate the many red ink-filled pages of mark-ups she returned to me on a near weekly basis. While often a source of frustration, I always valued her constructive criticisms and know that she has helped me to become a better communicator of my own ideas. It is no doubt that this dissertation could not have been created without her guidance and support.

Recognition also goes to my many collaborators at Michigan State University. Dr. Philip K. McKinley has been equally as welcoming and supportive throughout all my studies. Thanks go to Glen Simon, Jared Clark, and Jonathon Fleck for their collaboration on AC-ROS and the EvoRally platform. Thanks also go to Kira Chan for his collaboration on MoDALAS. Finally, thanks go to Dr. Wolfgang Banzhaf and Dr. Sandeep Kulkarni for agreeing to be members of my doctoral committee. I want to thank Dr. Banzhaf specifically for sparking my interest in evolutionary computation and also his genuine enthusiasm to hear about my research. None of this work would have been possible without the input of others, and it has been a joy to spend these past years studying at Michigan State University.

Finally, I would like to thank my family and friends for their support with each life decision. I thank my parents for providing me the freedom and love to grow and learn on my own, to explore whatever passions interest me the most, rather than limiting myself to those that make me most comfortable. Likewise, I give love to all the friends I have met along the way and to those I have yet to meet, because life is no fun without friends to share both the best and worst moments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

KEY TO ABBREVIATIONS

AC Assurance Case

AI Artificial Intelligence

API Application Programming Interface

CNN Convolution Neural Network

DFD Data Flow Diagram

DNN Deep Neural Network

EA Evolutionary Algorithm

EEA Estimation-Exploration Algorithm

ES Evolutionary Search

DFD Data Flow Diagram

DNN Deep Neural Network

GAN Generative Adversarial Network

GSN Goal Structuring Notation

GUI Graphical User Interface

KAOS Keep All Objectives Satisfied

LEC Learning-Enabled Component

LES Learning-Enabled System

MAPE-K Monitor-Analyze-Plan-Execute over shared Knowledge

MSE Mean Squared Error

PID Proportional-Integral-Derivative

ReLU Rectified Linear Unit

ROS Robot Operating System

SAS Self-Adaptive System

SBSE Search-Based Software Engineering

SBT Search-Based Testing

SOA Service-Oriented Architecture

SUT System Under Test

XAI Explainable Artificial Intelligence

XML Extensible Markup Language

CHAPTER 1

INTRODUCTION

New methods are needed to address the assurance of safety-critical, autonomous Learning-Enabled Systems (LESs), particularly when faced with uncertainties presented by real-world environments. For this dissertation, an LES is defined as any system that has one or more Learning-Enabled Components (LECs), where an LEC is any functional component with behavior that can be refined and optimized based on information gathered through experience (i.e., *learning models*) [187]. Many state-of-the-art LESs are trained via supervised learning and rely on finite sets of training and validation examples [73]. For example, an autonomous vehicle may include a vision-based LEC for obstacle detection that is trained to identify common driving obstacles based on thousands of hours of video recordings [70, 205]. However, once deployed, autonomous systems will likely encounter conditions not addressed at design time, such as adverse weather conditions for a cyber-physical system. Stakeholders require some level of confidence that these systems will deliver safe behavior in all contexts, even those that have not been explicitly evaluated at design time.

For assurance purposes, LESs must be shown to enforce system-level requirements even when faced with real-world uncertainties. *Assurance cases* are often used by the software engineering community as a formal means to certify a system satisfies its requirements [167]. Each assurance case is a structured argument, comprising claims and supporting evidence. To build an assurance argument for an LES, this dissertation uses assurance cases constructed with *Goal Structuring Notation* (GSN), which depicts each assurance case as a hierarchically-organized graphical model [2]. This dissertation also uses *Keep All Objectives Satisfied* (KAOS) goal models to hierarchical decompose high-level system objectives into system-level requirements [124]. In order to certify a system satisfies its requirements, assurance cases and goal models must be developed to consider the role and impact of uncertainty. For LESs, however, additional steps must be taken to assure that a system's learned behavior will respond appropriately to conditions that deviate from training experience.

This dissertation presents a framework to improve the *robustness* and *resiliency* of an LES by evaluating and mitigating the impact of uncertainty. Robustness refers to the ability of an LES to adequately process contexts that deviate from previously seen data. Resiliency refers to the ability of an LES to adapt and mitigate the impact of adverse operating conditions. This dissertation investigates how automatic techniques can evaluate the impact of uncertain conditions on an LES for assurance purposes. Furthermore, this dissertation contributes newly discovered automated methods to address the robustness and resiliency of an LES. To improve LES robustness when subject to environmental uncertainties, both design-time and run-time methods have been developed to identify gaps in training coverage and harden the LES with diverse synthetic training examples. Finally, to improve LES resiliency, self-adaptive methods have been developed to monitor and control an LES at run time to mitigate potential requirements violations due to inadequately trained learning models in uncertain environmental conditions.

1.1 Problem Description

The purpose of software assurance is to provide a level of confidence to stakeholders that a software system conforms to established requirements [167]. In contrast to traditional software programs, where an explicit algorithm exists to determine which actions will be performed, machine learning models in an LEC are typically trained and evaluated end-to-end in a black box manner with internal logic that is difficult to interpret by humans. Training effective machine learning models is also highly dependent on the quality of training data. If all run-time scenarios are not covered during training, then the behavior of an LEC may unexpectedly violate system requirements for corner cases. Software assurance for an LES must be able to show that even when training coverage is insufficient, mitigating actions are taken to ensure the system delivers acceptable behavior (i.e., fail-safe actions). Using assurance case models that consider the impact of uncertain hazardous conditions to an LES, this dissertation focuses on the robustness and resiliency of an LES to uncertainty, which are described in the context of LECs as follows.

Robustness. Robustness for an LES refers to the ability of its LECs to adequately process data that deviate from training experience. Learning models are optimized to perform well on a given set of scenarios during a training phase. It is assumed that these trained models will *generalize* to new data drawn from a similar distribution as provided training data. However, if training data fails to fully cover all real-world conditions, the behavior of the learning model can be unpredictable (i.e., corner cases may exist for which there has been no training or testing). Combined with issues of overfitting, the behavior of learning models for any data that deviates from seen training examples is uncertain. This dissertation explores techniques to improve the robustness of an LES to targeted forms of operational uncertainty, supported by established GSN assurance cases and KAOS goal models.

Resiliency. For an LES, resiliency refers to the ability of the system to adapt and compensate for inadequately trained LECs, in order to maintain satisfaction of system requirements at run time. When an LEC implements a fixed supervised learning model that has been trained on insufficient data, steps must be taken to determine under which contexts the learning model has been unprepared and actions must be taken to mitigate any requirement violations resulting from the inadequately trained learning model. This dissertation explores self-adaptative methods to enable an LES to assess its operating environment, determine when its LECs are insufficient for the current context, and perform alternative actions to mitigate failures that would result from the use of inadequate learning models.

1.2 Thesis Statement

This doctoral research explores assurance-guided methods to assess and mitigate the impact of uncertainty on systems with machine learning components. In particular, techniques are considered for assessing and improving the robustness and resiliency of an LES based on the analysis of system-level requirements with respect to assurance cases at both design time and run time.

Thesis Statement. *The combined application of machine learning and diversity-driven search-based software engineering techniques can be used to assess and improve the robustness and*

resiliency of an LES when faced with uncertain operating conditions.

1.3 Research Contributions

The objective of this dissertation is to address key challenges surrounding the robustness and resiliency of an LES when faced with uncertain operating conditions. The research contributions of this doctoral work are as follows.

1. This doctoral work introduces an automated method to assess the behavior of an LES in response to uncertain operating conditions. This method observes the statistical performance and structural behavior of an LES under a range of simulated operating contexts to create an archive of operating contexts that lead to unique system behaviors. This method then generates an archive of diverse contexts (i.e., mutually unique with respect to system behavior) to characterize the response of an LES to uncertain operating conditions [128].

2. This doctoral work introduces an automated method to augment the training and validation of learning models for operating conditions that are inadequately covered by training at design time [126, 130].

3. This doctoral work introduces an automated method to detect when learning models have encountered operating conditions for which they have been inadequately trained [127].

4. This doctoral work introduces a run-time framework to monitor and control the robustness and resiliency of learning models in response to run-time operating conditions, in order to mitigate failure resulting from the use of inadequately trained learning models [129].

This doctoral work has validated the above techniques on real-world autonomous vehicle datasets and on real-world autonomous platforms, such as a 1:5-scale autonomous vehicle and an autonomous rover with an embedded artificial intelligence (AI) processor for real-time computer vision and deep learning [34, 59, 70, 205]. This work has benefited from collaborators in the aerospace industry, the United States Air Force Research Laboratory (AFRL), and the automotive industry.

1.4 Organization of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides background information on key enabling technologies and foundational concepts for this work. Chapter 3 describes an approach to improve the robustness of an LES through the evolution of diverse synthetically augmented training/validation data. Chapter 4 describes an approach to improve the robustness of an onboard controller for a cyber-physical system. Chapter 5 describes methods to predict how LECs behave under uncertain run-time scenarios. Chapter 6 describes an approach to ensure autonomous systems maintain acceptable run-time behavior with respect to established assurance cases. Chapter 7 describes an approach to dynamically adapt the behavior of autonomous systems at run time with respect to high-level system requirements models. Chapter 8 describes an approach to monitor and control LECs at run time to prevent their use under conditions for which they are inadequately trained. Finally, Chapter 9 presents conclusions, summarizes research contributions, and overviews future investigations for this doctoral work.

CHAPTER 2

BACKGROUND

This chapter provides background information on enabling technologies and foundational concepts used to support this doctoral work. Section 2.1 describes Deep Neural Networks (DNNs) and challenges associated with their use in LESs. Section 2.2 describes evolutionary computation techniques that are relevant for this doctoral work. Section 2.3 describes fundamental concepts for autonomous systems with self-adaptive capabilities. Section 2.4 describes the various DNN datasets and implementation used to validate this doctoral work. Finally, Section 2.5 describes the autonomous platforms used to demonstrate this doctoral work.

2.1 Deep Neural Networks

This section describes fundamental concepts for DNNs and current challenges facing the use of DNNs in software systems.

2.1.1 Elements of a Deep Neural Network

As a form of machine learning, DNNs are *universal approximators* [89], capable of approximating any complex function with sufficient training. In the case of autonomous vehicles, they may be used to process data from onboard sensors [99] for tasks such as lane-keeping [191] and collision-avoidance [204]. Represented with multi-layered architectures (as shown in Figure 2.1), DNNs comprise multiple intermediate *hidden layers* that connect a set of input *features* (x_i) to target output *labels* (y_i). Input features can include any observable property, such as pixels for camera images. Output labels can describe any piece of information that may be inferred from the input, such as a safe steering angle or brake pedal pressure for a vehicle. Each hidden layer, comprising a number of units called *neurons*, represents a single linear transformation of the layer's input followed by a non-linear *activation* function that acts as a mechanism to filter or amplify information derived from the layer's input. DNNs are trained to approximate target functions by adjusting values of the

weight matrix ($\mathbf{W}_i$) and bias vector ($\mathbf{b}_i$) associated with each layer. Typically, weights and biases are adjusted to minimize an *objective loss* function that measures the amount of error between a DNN's actual output and the expected (i.e., trained) output. Once the objective loss is minimized, the training phase is terminated, and the DNN is verified by evaluating it against separate test data.

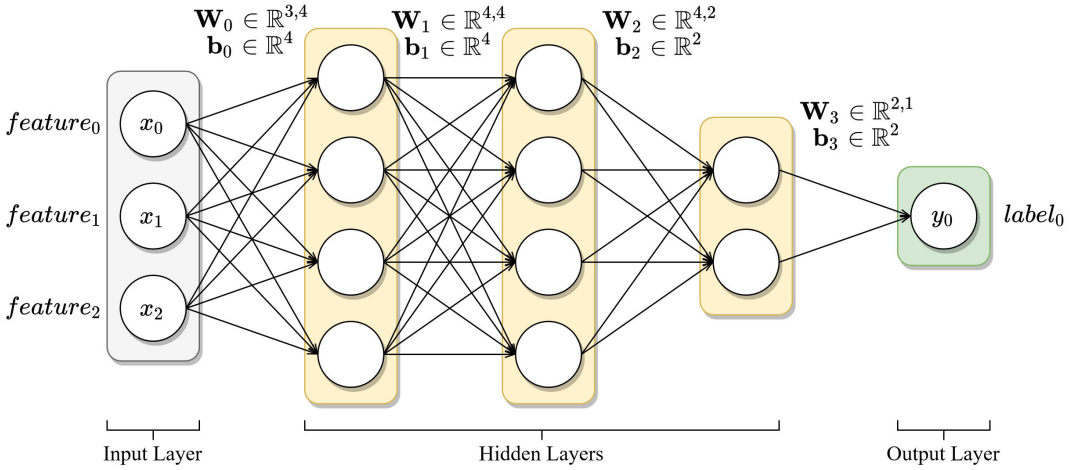


Figure 2.1: High-level illustration of a simple DNN. In practice, the network topology may be more complex. Hidden layers enable the network to model a target function that maps input features (x_i) to output labels (y_i). Neurons are represented by circles and links represent the flow of data from input to output. The output of each neuron is determined by applying an activation function to a linear transformation of input values by weight matrix ($\mathbf{W}_i$) and bias vector ($\mathbf{b}_i$).

By linking layers together, DNNs are capable of learning increasingly abstract relationships between features in the source input [73]. However, when weights are adjusted for the network as a whole with an objective defined only in terms of the network's final output, as with typical training methods, developers have no direct control over how any specific hidden layer infers information in isolation. As such, intermediate representations of information inferred by the DNN (i.e., the decision logic) can be difficult to interpret. Even when the final output of a DNN may appear to be correct for all evaluated test data, the exact reasoning for how a DNN determines its output is often a "mystery" [112].

2.1.2 Challenges for Deep Neural Networks

With DNNs, an increasing number of hidden layers (i.e., the *depth*) has been associated with an ability to derive and learn from increasingly abstract features from the source input [116]. AlexNet [120], an image recognition application, popularized the use of DNNs for vision tasks by showing significant improvements over existing computer vision techniques. Research even suggests that DNNs can compete with the human visual cortex [108]. However, this intuition has been challenged by the discovery of "adversarial examples" that exploit a DNN into making incorrect predictions in unexpected ways [54, 75, 122].

Adversarial examples have been shown to force a DNN to make *false negative* predictions by adding small amounts of noise (typically imperceptible to a human) to images [209] and *false positive* predictions for images constructed entirely from noise [160]. Furthermore, studies have shown that when both training and test data contain the same surface statistical regularities, it is possible for a DNN to learn and make predictions based on superficial details of an image's composition rather than the high-level semantics of the image's contents [101]. In such cases, a DNN can show a high degree of accuracy on test data while not successfully generalizing to data outside of the training or test datasets.

This doctoral work is complementary to the field of Explainable Artificial Intelligence (XAI), where XAI aims to address the accessibility of machine learning systems for human understanding [10]. Active research in XAI has developed methods to quantify the interpretability of hidden layers [12], visualize intermediate representations [229], and verify software implementations of an LES [26, 50, 193]. One emphasis of XAI research is to develop methods that lead to more robust and reliable machine learning systems [80]. In the context of XAI, robustness refers to the ability of a DNN to reduce its sensitivity to malicious noise in its input. Chapter 3 proposes a method for improving the robustness of a DNN by identifying gaps in training data that make the DNN perform poorly in the presence of a given environmental condition and "filling" in the gaps with synthetic training data. This doctoral work supports the goal of XAI to help humans understand, trust, and effectively manage DNNs in noisy environments.

2.2 Evolutionary Computation

This section describes how evolutionary computation has been harnessed for search-based software engineering (SBSE) and evolutionary robotics; for a more comprehensive review, see the survey by Harman *et al.* [82]. Additional information is provided on *novelty search*, a diversity-driven form of evolution.

2.2.1 Search-Based Software Engineering with Evolution

Test case generation for software verification and validation has been a long-standing challenge, including considerable research in automated Search-Based Testing (SBT) techniques [82]. These search-based methods explore a space of candidate test cases to find a subset that optimizes a given objective (e.g., software coverage, program faults, etc.). [148]. Common parameter optimization methods include *random search* and *grid search* [17]. Random search methods generate candidate solutions with random parameter values until a desirable solution is found. Grid search methods exhaustively search every parameter value to determine the optimal solution. While random search can typically find solutions more efficiently than grid search, a high amount of variation in results is possible, where the resulting solution may be far from optimal. Grid search guarantees an optimal solution will be found, but it is too computationally expensive to implement for high-dimensional solutions spaces or when the evaluation time for each candidate solution is non-trivial. More advanced techniques introduce heuristics to help guide the search towards relevant solutions more reliably than random generation and more efficiently than grid search [82].

Testing tools such as EvoSuite [60] and SAPIENZ [147] use Evolutionary Algorithms (EAs) [52]. These tools require software engineers to specify a test case representation as a *genotype* for the EA, and each encoded instance of a test case is referred to as a *genome*. The EA includes an iterative process of test case generation and selection to evolve better quality test cases. Genomes are manipulated through variation operations, such as *mutation* and *recombination*, and are selected for preservation between generations by a *fitness* heuristic. Fitness is typically a metric based on

observable properties of the System Under Test (SUT), such as code coverage, specified as the *phenotype*. Thus, EAs explore the space of test cases by their genomes and compare the quality of each test case the fitness of its associated *phenome*. Through this process, SBTs can use evolution to discover test cases more pertinent to the SUT with respect to the chosen fitness metric. For solution spaces with large regions of suboptimal solutions, the use of a fitness heuristic allows the search process to avoid much of the unnecessary evaluation of an exhaustive search (i.e., the evaluation of undesirable candidates). However, since EAs do rely on stochastic elements, there can be some variation in resulting solutions. The amount of variation is often determined by the quality of the chosen fitness metric, the corresponding *fitness landscape* [180], and on how well the fitness heuristic is able to converge onto an optimal solution.

2.2.2 Evolutionary Robotics

The field of *evolutionary robotics* [58] harnesses the open-ended search capabilities of EAs by using genomes to encode specifications of the robot's control system or even aspects of its morphology (external structure), allowing for evolution to be leveraged in the process of designing more robust and resilient robots. Individuals in a population are evaluated with respect to one or more tasks, with the best performing individuals selected to pass their genes to the next generation. Simulation is typically used to evaluate individuals, greatly reducing the time to evolve solutions while avoiding possible damage to physical robots. From an engineering perspective, a major advantage of evolutionary search is the possible discovery of solutions (as well as potential problems) that the engineer might not otherwise have considered.

2.2.3 Novelty Search

Novelty search is an evolution-based method to explore large solution spaces for interesting candidate solutions in the absence of an explicit fitness objective [133, 134]. Greedy algorithms that focus on maximizing or minimizing an objective function have been found to be efficient and effective for simple solution spaces, but they are prone to discovering suboptimal solutions when the solution

space contains many local optima or the global optima covers a very small region with no smooth gradient (i.e., non-convex spaces). In contrast, novelty search ignores any specific objective other than diversifying each candidate solution. Diversity is encouraged by comparing each candidate with its neighbors and favoring candidates that are least similar to their nearest neighbors. Thus, through an iterative process, novelty search can discover a collection of candidate solutions that cover a wide region of the solution space while preventing all candidates from being attracted to the same local optima. When implemented as an EA, a population of individual candidates can be evolved based on how diverse they are in relation to an archive of the most diverse candidates discovered.

2.3 Self-Adaptive Systems

This section describes basic concepts for Self-Adaptive Systems (SASs) and methods for run-time monitoring to support run-time adaptations for autonomic computing.

2.3.1 Self-Adaptation at Run Time

Research in autonomic computing has led to systems that can manage and re-configure (i.e., *self-adapt*) to maintain high-level objectives in complex and dynamic run-time environments [105]. The MAPE-K (Monitor-Analyze-Plan-Execute over shared Knowledge) control loop (see Figure 2.2) was introduced by Kephart and Chess [105] as a framework to manage adaptations for SASs. First, the *Monitor* step gathers information about managed system components and the run-time environment. Second, the *Analyze* step infers a context for the current system states and selects an appropriate response (i.e., system adaptation). Third, the *Plan* step identifies an appropriate procedure to perform the selected adaptation. Finally, the *Execute* step realizes the adaptation by re-configuring managed components. Each step has access to a shared *Knowledge* database that includes information such as resource availability, performance constraints, and functional objectives. Thus, an *autonomic manager* enables continuous monitoring and re-configuration of a SAS at run time.

This dissertation uses the terminology described by Cheng *et al.* [37] to describe adaptations for an SAS. An adaptation *tactic* is a procedure that comprises a *condition*, *action*, and *effect*, where a condition describes when an adaptation is applicable, an action (or sequence of actions) describes what operations to perform on managed components, and an effect describes the intended result of the adaptation. For each cycle of the autonomic manager's control loop, a tactic is chosen based on the determined status of managed components. Once a tactic has been selected, the control loop is paused while the actions associated with the tactic are executed from beginning to end. Finally, the effectiveness of the tactic is determined by comparing its intended effect to the resulting status of managed components. Success or failure of a tactic can inform future iterations of the control loop.

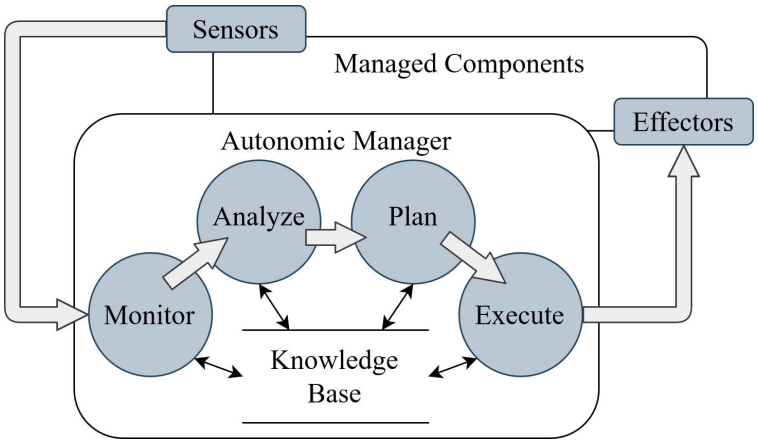


Figure 2.2: Illustration of a generic MAPE-K control loop for an SAS. An *autonomic manager* monitors, analyzes, plans, and executes changes to a collection *managed components* at run-time.

By modeling the behavior of an SAS as finite state automata, Zhang and Cheng [233] categorize adaptations into three general categories: *one-point adaptation*, *guided adaptation*, and *overlap adaptation*. These categories focus on the behavior of the managed element before an adaptation (i.e., source behavior) and after an adaptation (i.e., target behavior) Since adaptations may not be safe to perform in all states of a managed element, focus is also given to those states for which adaptation can be safely executed (i.e., *quiescent states*). For one-point adaptation, the managed element is able to transition from the source behavior to the target behavior in a single state transition. For guided adaptation, a request is made before executing the adaptation, and the

managed element is forced into a restricted mode, where the source behavior is constrained such that a quiescent state can be found for executing the adaptation. Finally, for overlap adaptation, the managed element begins exhibited target behavior before the source behavior ceases, such that for a period it exhibits both behaviors. Particularly for safety-critical applications, it is important to consider the appropriate category of adaptation to avoid an accidental transition into a hazardous system state.

2.3.2 Run-Time Monitoring with Utility Functions

When monitoring run-time systems for self-adaptation [105], *utility functions* have been used to simplify system behavior assessments [36, 215]. Utility functions map system attributes (i.e., the system state) into real scalar values to express a degree of goal (i.e., requirements) satisfaction [45]. Explicitly, a utility function takes the following form.

$$u = f(\mathrm{v}) \tag{2.1}$$

The *utility value* is a real scalar value ($u \in [0, 1]$), and the *system state* vector ($\mathrm{v} = [s_0, ..., s_n]$) reflects specific attributes (s_i) of a system and its environment. Utility functions can be elicited manually by domain experts and application users (i.e., *subjective utility*), or derived automatically from software artifacts (i.e., *objective utility*) [45]. Whether derived manually or automatically, the intent of a utility function is to rank desirable system states higher than undesirable system states. Thus, utility functions enable a quantifiable comparison of low-level system states in terms of high-level task-oriented objectives.

2.4 Validation Datasets and Implementations for Deep Neural Networks

A variety of datasets have been selected to demonstrate and validate DNNs using the methods proposed by this doctoral work. This section describes each respective dataset and DNN application in detail.

2.4.1 CIFAR-10 Dataset for Image Recognition

Portions of this doctoral work have been applied to image recognition DNNs trained for the CIFAR-10 benchmark, a dataset widely cited in deep learning research [119]. For image recognition, the goal is to classify the contents of each CIFAR-10 image into one of ten evenly distributed categories: *airplane*, *automobile*, *bird*, *cat*, *deer*, *dog*, *frog*, *horse*, *ship*, or *truck*. The benchmark includes two datasets: a set of 50,000 labeled training images and a set of 10,000 labeled test images. Performance for a CIFAR-10 DNN is typically measured by its *accuracy* when given a set of validation images, where accuracy is the percentage of correct predictions over the total number of given images. State-of-the-art DNNs have been reported to achieve accuracy above 91% on default CIFAR-10 test data by use of highly-tuned architectures, optimization methods, and data augmentation [68, 232].

All CIFAR-10 image recognition DNNs constructed for this doctoral work use *PyTorch* [174] machine learning libraries to implement a ResNet-20 [87] architecture. The ResNet architecture (illustrated in Figure 2.3) comprises a series of residual blocks with decreasing resolutions and increasing numbers of filters. Each residual block can be bypassed when processing a given input via a "shortcut connection." Individual blocks include convolution layers to transform incoming images, batch normalization operations to reduce covariate shift [95], and rectified linear unit (ReLU) [157] activations to filter out or amplify relevant features. Source images begin with a resolution of 32×32, pixels, and intermediate representations are gradually reduced in size down to an 8×8 resolution. The network then produces a feature vector by performing a *global average pooling* operation to compute the average value for each feature image. A *softmax*[1] operation is applied to the final layer to predict a corresponding classification category. For example, in Figure 2.3, the *deer* classification category has been assigned by the DNN with the highest probability for the given input image.

The baseline CIFAR-10 DNN for this doctoral study has been trained on default CIFAR-10

[1] *Softmax* is a function used to estimate a classification probability distribution [73, p. 179].

training images to minimize the categorical cross entropy[2] of its predictions against the expected training output. The training procedure implements an Adaptive Moment Estimation (Adam) gradient descent method [109] with a learning rate that decays from 10^{-3} to 10^{-7} over 200 epochs.[3] These settings were chosen by empirical analysis and cross-validation on a subset of the training data. Standard data augmentation methods such as image shifting and image flipping have also been used to add variation to the training data. Trained under these conditions, the baseline CIFAR-10 DNN can achieve an accuracy of 91% on default CIFAR-10 test data.

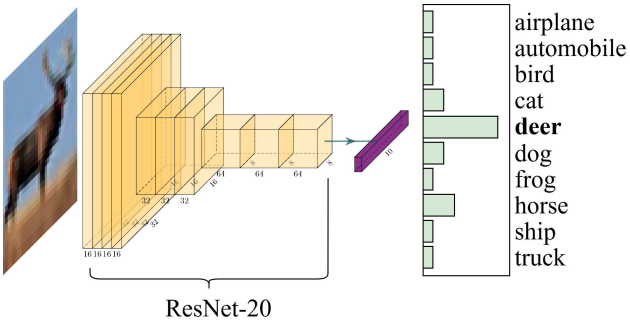


Figure 2.3: High-level illustration of a CIFAR-10 DNN with a ResNet-20 architecture. Images are provided as input. The network is composed of a series of residual blocks with "shortcut connections." Output is a single classification category determined from a probability distribution.

2.4.2 KITTI Vision Benchmark Suite for Autonomous Driving Object Detection

Portions of this doctoral work have also been applied to DNNs trained for image-based object detection. The KITTI Vision Benchmark Suite [70], produced by Karlsruhe Institute of Technology and Toyota Technological Institute at Chicago, provides a real-world benchmark for data-driven autonomous driving tasks. KITTI data has been collected from an autonomous car equipped with a laser scanner, a stereo camera, and a global positioning system. However, for two-dimensional object detection, only static images taken from the vehicle's camera system are referenced in this dissertation. Separate training and validation images have been selected and prepared from the benchmark dataset [166]. In contrast to the CIFAR-10 image recognition problem, each image in the

[2] *Cross entropy* measures the relative difference between two probability distributions [73, p. 73].
[3] An *epoch* refers to a single training iteration [73, p. 239].

KITTI dataset can contain multiple objects of interest. Object detection consists of two sub-tasks: classifying and locating each object in the image with a bounding box. Possible classification categories for each object include *car*, *van*, *truck*, *pedestrian*, *sitting-person*, *cyclist*, *bus*, and *miscellaneous*. Bounding boxes for each object are defined as the smallest rectangle in the source image to contain all pixels related to the object of interest. Thus, for each given input image, an object detector will output a corresponding set of classifications and bounding boxes for each relevant object detected in the image. For demonstration, Figure 2.4 provides an example image input (Figure 2.4a) and the corresponding output (Figure 2.4b) produced by an object detector.

All KITTI object detection DNNs constructed for this doctoral work use the RetinaNet [139] architecture for two-dimensional object detection with the PyTorch machine learning libraries. The implemented RetinaNet architecture (illustrated in Figure 2.5) uses a ResNet-50 "backbone" to extract a pyramid of features from the source image. These features are then fed into a set of classification sub-networks to classify each detected object and a set of regression sub-networks to compute bounding boxes for each detected object.

The baseline KITTI DNN referenced in this dissertation has been trained by an Adam gradient descent method to minimize a focal loss[4] for given training images. Training began with an initial learning rate set to 10^{-4} and ended after 25 epochs, when the focal loss converged to a minimum. These settings were chosen by empirical analysis and cross-validation with a subset of the training data. In contrast to the CIFAR-10 image recognition application, where an image is either correctly classified or misclassified, object detection can result in *false positives* (i.e., detecting false objects) and *false negatives* (i.e., failing to detect an object), and therefore, the performance of an object detector is measured by its *precision*, *recall*, and *F-scores*.[5] For this doctoral study, the baseline KITTI DNN has been trained on 6,373 training images to achieve a precision of 77%, a recall of 88%, and an F-score of 82% on 1,108 validation images.

[4] *Focal loss* is an adaptive form of cross entropy to account for class imbalance [139].

[5] *Precision* is the ratio of correctly detected objects to all detected objects (i.e., the ratio of *true positives* to both *true positives* and *false positives*). *Recall* is the ratio of correctly detected objects to all real objects (i.e., the ratio of *true positives* to both *true positives* and *false negatives*). An *F-score* is the harmonic mean of both precision and recall [202].

(a) Example Input Image (b) Example Output From Object Detector

Figure 2.4: Examples of an image input and output from a KITTI object detector DNN. Images are taken from a vehicle's onboard camera (a). Output is a set of labeled bounding boxes corresponding to each detected object (b).

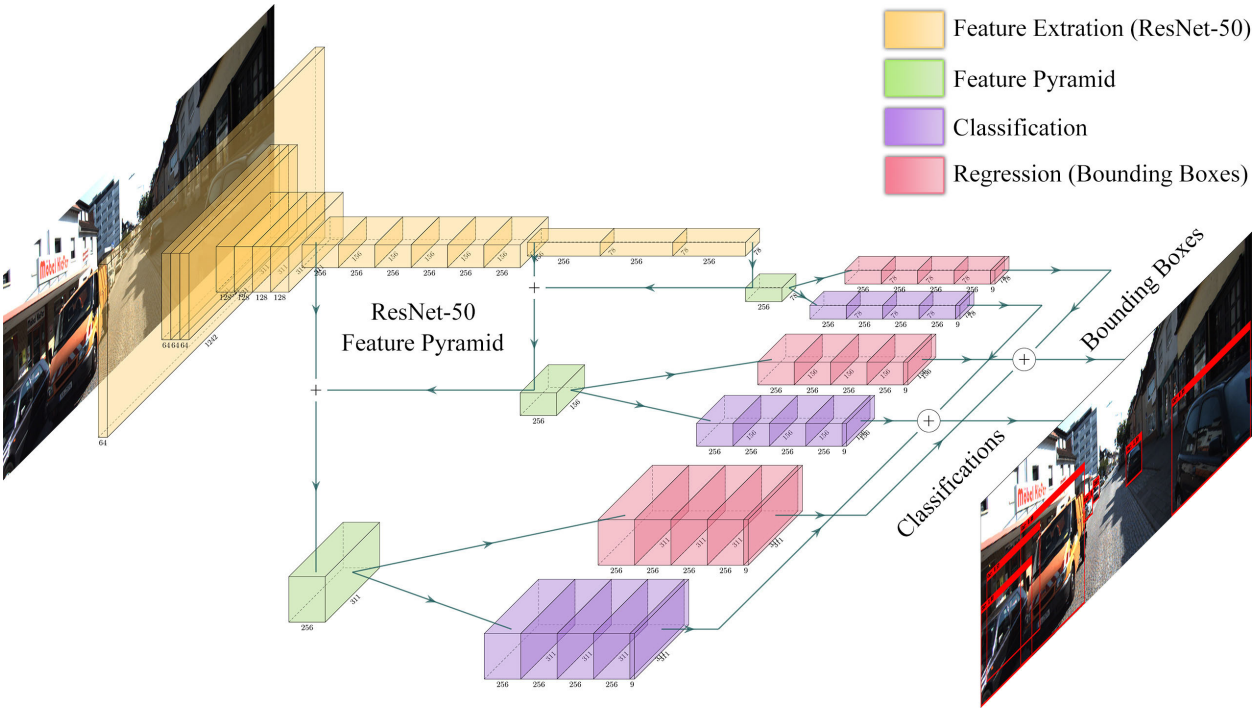
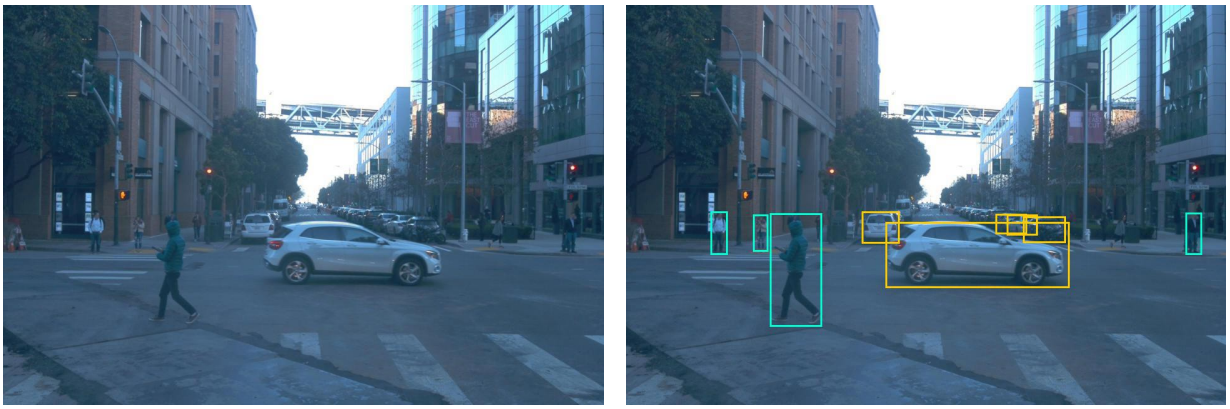


Figure 2.5: High-level illustration of a RetinaNet DNN with a ResNet-50 backbone architecture. Images are provided as input. The network is composed of a feature extraction sub-network (yellow) with classification and regression sub-networks (blue and purple, respectively) at each level of a feature pyramid (green). Output is a collection of classification categories and bounding boxes for each respective object detected in the image.

2.4.3 Waymo Open Dataset for Autonomous Driving Object Detection

In addition to training object detection DNNs on KITTI benchmark data, portions of this doctoral work have also trained object detection DNNs on the Waymo Open Dataset [205] comprising real-world autonomous driving data. Separate training and validation datasets have been selected

and prepared from the forward-facing camera images [166] in the Waymo dataset. Possible classification categories for each object in the Waymo dataset include *vehicle*, *pedestrian*, *sign*, *cyclist*, and *unknown*. For demonstration, Figure 2.6 provides an example image input (Figure 2.6a) and the corresponding output (Figure 2.6b) produced by an object detector on an image from the Waymo dataset.



(a) Example Input Image (b) Example Output From Object Detector

Figure 2.6: Examples of an image input and output from a Waymo object detector DNN. Images are taken from a vehicle's onboard camera (a). Output is a set of labeled bounding boxes corresponding to each detected object (b).

All Waymo object detection DNNs constructed for this doctoral work use the same RetinaNet architecture illustrated in Figure 2.5 for two-dimensional object detection. The baseline Waymo DNN referenced in this dissertation has been trained by an Adam gradient descent method to minimize a focal loss for given training images. Training began with an initial learning rate set to 10^{-4} and ended after 25 epochs, when the focal loss converged to a minimum. These settings were chosen by empirical analysis and cross-validation with a subset of the training data. For this doctoral study, the baseline Waymo DNN has been trained on 158,081 training images to achieve a precision of 88%, a recall of 74%, and an F-score of 80% on 39,987 validation images.

2.5 Demonstration Platforms

Two separate autonomous platforms have been used to demonstrate and validate the methods proposed by this doctoral work. This section describes the underlying control software and imple-

mentations of each autonomous system.

2.5.1 Robot Control Software

In order to manage heterogeneous hardware and enable software reuse in robotic applications, many developers implement the control logic of sensors and actuators as components of a *robot middleware* [53]. The Robot Operating System (ROS) [176] is an open-source robot middleware that has been widely adopted by both academia and industry [114]. The fundamental elements of a ROS-based system are *nodes*, *topics*, and *services*. ROS enables the controlling algorithms for a single application to be divided into multiple independent processes (i.e., ROS *nodes*). ROS nodes can publish/subscribe data unidirectionally through message buses (i.e., ROS *topics*) and also handle bidirectional request/reply interactions (i.e., ROS *services*). As a *peer-to-peer network* of nodes, a ROS-based system can be implemented over multiple processing units with a common registry service to facilitate communication between nodes (illustrated in Figure 2.7).

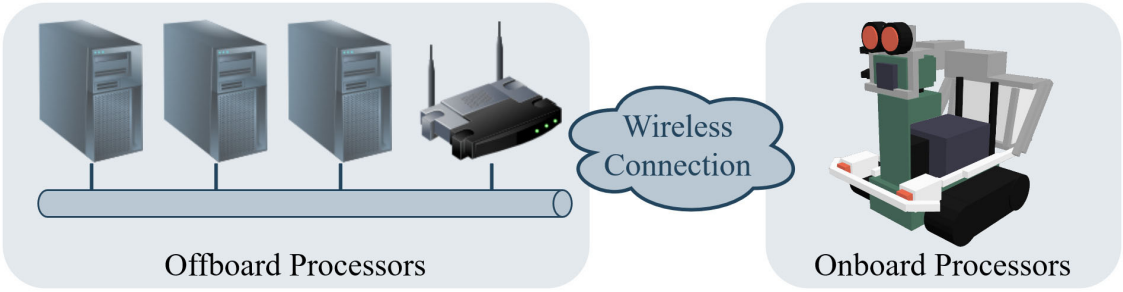


Figure 2.7: Typical ROS configuration [176]. Software for a ROS-based system executes ROS nodes over multiple onboard and offboard processors that communicate over a wireless bridge via ROS topics and services.

2.5.2 The EvoRally Autonomous Vehicle

Portions of this doctoral work have been validated on the EvoRally autonomous vehicle platform. EvoRally is based on the AutoRally [71, 223] open-source platform designed and developed by the Georgia Institute of Technology (Georgia Tech) as a test bed for various autonomous vehicle sensing and control methods. EvoRally is modeled to represent a 1:5 scale remote control truck

and has a top speed of 27 m/s (60 mph). Figure 2.8 provides images of the EvoRally platform in both its physical and simulated form. Software for controlling EvoRally is modular and highly customizable, implemented using the *Kinetic* [184] distribution of ROS packages. The rover can be controlled either autonomously or manually by a remote operator. An advantage for using EvoRally for research purposes is that it is smaller and less expensive than a full-sized autonomous vehicle, yet it uses much of the same control software and mechanical structures.

(a) Physical platform (b) Simulated platform

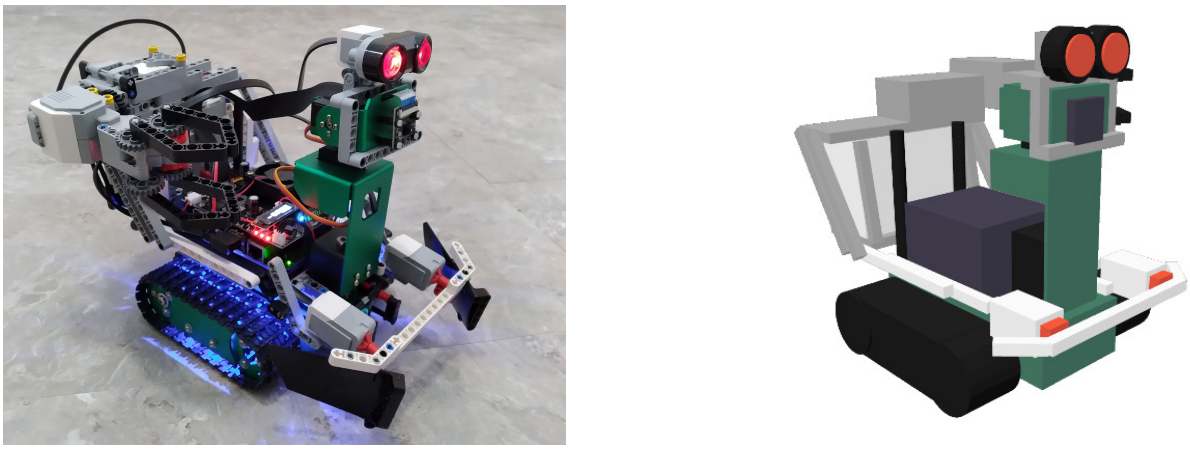
Figure 2.8: The EvoRally autonomous driving platform. The physical platform is shown on left (a), and a simulated version is shown on right (b).

Simulation of EvoRally is managed by the *Gazebo* simulator [113], chosen for its support of complex environments and sensors modeled after many commercially available devices. An accurate model of the EvoRally platform has been constructed within Gazebo, closely matching all components and physical characteristics of the real platform. With the capabilities offered by Gazebo and an accurate model of the vehicle, the reality gap between what is observed in simulation and the behavior of a physical system is expected to be minimal.

2.5.3 Deep Learning-Driven Autonomous Rover

Portions of this doctoral work have also been validated on a robotic rover assembled with a suite of sensors and actuators to enable autonomous behavior. Photographed in Figure 2.9, the dimensions of the rover are approximately $30.5 \times 20.5 \times 22.0$ centimeters. The rover includes an *NVIDIA Jetson Nano* processor to support efficient onboard deep learning computations [59] for computer

vision. Sensors include a forward-facing camera, an ultrasonic range finder, and a touch-sensitive bumper. Control software for the rover is implemented using the *Melodic* [185] distribution of ROS packages. The rover can be controlled either autonomously or manually by a remote operator. When operating autonomously, the rover uses both an ultrasonic range finder and a vision-based object detector to detect obstacles in the environment.



(a) Physical platform

(b) Simulated platform

Figure 2.9: For demonstration, an autonomous rover has been assembled to explore deep learning on embedded systems. Sensors include a camera and an ultrasonic range finder. The physical platform is shown on left (a), and a simulated version is shown on right (b).

In autonomous mode, the rover relies on computer vision to identify the types of obstacles present in its environment. The rover's vision-based object detector is implemented as a RetinaNet DNN, using PyTorch deep learning libraries. The object detector has been trained to detect objects from two-dimensional images taken from the rover's forward-facing camera. For each object detection, both a category label and bounding box are given to identify the type of object and what region of the image it covers.

To train and validate the object detector, a set of $2,500$ labeled images were collected with images of both humans and deer scattered throughout a controlled test environment, where $2,000$ of the images were reserved for training and 500 were reserved for validation purposes only. The object detector DNN was trained until the training error converged to a minimum (after 25 epochs). When evaluated against the reserved validation images, it was found to correctly detect images of

humans and deer with a precision of 98.8%, a recall of 94.8%, and an F-score of 96.8%.

CHAPTER 3

LEARNING ROBUSTNESS THROUGH DIVERSIFIED TRAINING

Data-driven LESs are limited by the quality of available training data, particularly when trained offline. For systems that must operate in real-world environments, the space of possible conditions that can occur is vast and difficult to comprehensively predict at design time. Environmental uncertainty arises when run-time conditions diverge from design-time training conditions. To address this problem, automated methods can generate synthetic data to fill in gaps for training and test data coverage. This chapter proposes an evolution-based technique to assist developers with uncovering limitations in existing data when previously unseen environmental phenomena are introduced [126, 130]. This technique explores unique contexts for a given environmental condition, with an emphasis on diversity. Synthetic data generated by this technique may be used for two purposes: (1) to assess the robustness of a system to uncertain environmental factors and (2) to improve the system's robustness. This technique is demonstrated to outperform random and greedy methods for multiple adverse environmental conditions applied to image-processing DNNs.

The remainder of this chapter is organized as follows. Section 3.1 overviews the motivation and objectives of this chapter. Section 3.2 describes a methodology for assessing and improving the robustness of a DNN. Section 3.3 presents results from an empirical evaluation of DNNs trained with benchmark data. Section 3.4 reviews related work in automated testing and DNNs. Finally, Section 3.5 provides a concluding summary for this chapter.

3.1 Overview

For cyber-physical LESs in real-world environments, unpredictable behavior can occur when design-time training conditions deviate from run-time conditions [44, 212]. When different forms of adversity are introduced into the environment, such as inclement weather or a malicious security attack, the outcome can result in costly damage to hardware, human injury, or even fatalities (e.g., recent accidents with autonomous vehicles [162, 163, 164]). This chapter focuses on systems that

can refine or optimize functional behavior based on information gathered through experience (e.g., obstacle detection for a navigation system) in contrast to the use of machine learning to manage self-adaptations [121]. Model-driven LESs use domain knowledge and a semantic model of the learned task [227]. In contrast, the behavior of a *data-driven LES* (e.g., a system comprising a DNN) is determined entirely from patterns inferred from training data without a well-defined semantic model [16]. Trustworthiness for these systems must be established at the design stage, but accounting for every possible example of a run-time condition is challenging. For example, consider an autonomous vehicle with a DNN that has been trained for object detection but has only been exposed to images taken in clear weather. Due to rainy conditions at run time, a situation may arise where the lens of the camera is partially obstructed by a falling raindrop on the camera lens. Examples illustrated in Figure 7.16 show how raindrops with changing size and position can adversely impact image recognition in potentially subtle and unexpected ways. Absent of any training or test data that include this particular raindrop condition, it is difficult for developers to assess the impact occluding raindrops will have on the object detector or autonomous vehicle. This chapter introduces an automated technique to explore the impact of such uncertain environmental conditions on an LES and to uncover specific contexts that produce a wide range of system behavior. Results from this approach can be leveraged to assess and improve a system's *robustness* to the environment, where robustness refers to an LES's ability to deliver consistent and acceptable behavior in the presence of a noisy environment [230].

Despite a widespread movement to incorporate *deep learning* [73] into software applications [85], proving the correctness of software driven by deep learning remains challenging [81]. For the scope of this chapter, an LES is any system with one or more learning components that have been trained offline via supervised deep learning. Since deep learning methods allow a system to solve tasks by example without a full description of the problem space, it is attractive for use in problem domains that are dynamic, poorly defined, or otherwise too complex to solve by classic algorithms. However, introducing deep learning into safety-critical systems is troublesome for software assurance purposes [213]. Classical algorithms contain well-defined, logic-based rules that

can be verified by formal methods, and traditional software systems developed through programming languages have a broad collection of tools/techniques for testing and quality assurance. In contrast, deep learning systems such as DNNs are difficult to interpret and have been shown to have limited robustness when training examples inadequately cover the problem domain. Collecting "good" data to train and test DNNs is non-trivial [25, 86, 197] and expensive to produce, in terms of both time and human effort. Furthermore, manual data selection is susceptible to cognitive biases [27]. Computer-assisted techniques can alleviate these issues by augmenting data to expand coverage of the problem space.

i. Correct ii. Correct iii. Incorrect i. Correct ii. Correct iii. Incorrect

(a) Impact of raindrops on automobile detection (b) Impact of raindrops on deer detection

Figure 3.1: Examples of how occluding raindrops with changing size and position can impact an image recognition DNN trained with 91% test accuracy for CIFAR-10 [119] images. Examples images are provided for an automobile (a) and a deer (b). Unaltered images (i.) are shown with resulting DNN classifications labeled on the bottom. Synthetic raindrops can be introduced with either no impact on classification (ii.) or negative impact (iii.). The impact any given raindrop will have on the resulting classification is not known *a priori*.

This chapter describes an evolution-based technique to construct synthetic variants of existing datasets by introducing contexts of environmental conditions that are not present in default training/validation data and therefore have an uncertain impact on the system. Furthermore, in order to assess and improve system robustness, this technique seeks out contexts that result in the most unique and extreme system behavior, to uncover sets of training/test examples that cover a wide range of system behavior. For example, an environmental condition may describe where and how a raindrop appears on an image from an autonomous vehicle's onboard camera, and the proposed technique can generate multiple raindrop variations that impact the performance of the autonomous vehicle in mutually unique (i.e., *diverse*) ways. This technique implements an evolution-based

search method that simulates environmental conditions on sensor data via parameterized *transformation functions* to find specific parameter values (i.e., *contexts*) that result in increasingly different behavior for the LES over each generation. Synthetic data is then produced by transforming existing sensor data to reflect the evolved contexts. This technique enables a developer to automatically assess a machine learning system under a wide range of contexts for an environmental condition not covered by default. Furthermore, it provides a means to train machine learning systems to be more robust to a wider range of otherwise unexposed environmental contexts.

This chapter introduces ENKI,[1] a black box automation tool that supports the proposed technique [126, 130]. ENKI can be used to discover diverse operating contexts for any general SUT, based on the SUT's observable behavior. This chapter addresses two key research questions. First, can we use ENKI to identify gaps in training that cause an LES to perform poorly? Second, can ENKI improve the robustness of an LES to the effects of an uncertain environmental condition? To answer these questions, this chapter focuses on DNNs designed for image recognition and object detection, key capabilities needed for several autonomous driving features (e.g., obstacle avoidance, lane management, and adaptive cruise control). Experiments have been conducted on DNNs under a variety of environmental conditions to demonstrate how ENKI can be used to construct useful synthetic test and training data.

Results from this study show that by using ENKI with established benchmark datasets for image recognition (CIFAR-10 [119]) and object detection (KITTI [70]), relevant examples of environmental conditions that negatively impact the performance of DNNs (e.g., decreased lighting, haze, and the presence of rain) can be automatically identified. Furthermore, this chapter demonstrates that synthetic training data generated by ENKI can be used to improve the robustness of these DNNs to the introduced environmental conditions.

[1] Enki is an ancient Sumerian deity associated with knowledge, mischief, and creation [118].

3.2 Assessing and Improving the Robustness of Learning Models

This section provides an overview of how ENKI can be used to assess and improve the robustness of a DNN when exposed to uncertain environmental conditions. Section 3.2.1 clarifies how an environmental condition is defined for this study, Section 3.2.2 describes the steps involved with assessing a DNN's performance under a specific uncertain environmental condition, and Section 3.2.3 describes the steps involved with improving the robustness of a DNN.

3.2.1 Definition of an Environmental Condition

This approach is intended for use when the given training data for a DNN contains no real-world examples of an environmental phenomenon of interest. It is assumed that the phenomenon can be simulated using examples of real-world data and a simulation environment. For illustrative purposes, this section considers a specific environmental condition where a raindrop may partially occlude the view of an autonomous system's camera sensor. However, any environmental phenomena that the simulation environment is capable of replicating may be used in place of the raindrop condition.

The raindrop occlusion condition is considered a "known unknown" for DNNs that have only been trained with clear-weather images. It is an environmental condition that has *known* appearance (i.e., its impact on sensor input can be described) but also has *unknown* effects on the DNN's behavior. For this work, such environmental conditions can be described by a parameterized *transformation function* (T),

$$\mathbf{x}' = T(\mathbf{g}, \mathbf{x}) \tag{3.1}$$

where $\mathbf{g}$ is a set of parameter values that describe a specific appearance of the condition (i.e., an operational *context*), $\mathbf{x}$ is an example of sensor data absent of the condition, and $\mathbf{x}'$ is an example of how the sensor data would appear in the presence of the given context for the condition. For raindrop occlusion, parameters in $\mathbf{g}$ may describe a raindrop's appearance (e.g., position, size, etc.), and each instance of $\mathbf{g}$ corresponds to an alternative context of a raindrop (e.g., a large raindrop in

the center of view versus a small one in the upper-right corner). Given a real-world camera image absent of any raindrop occlusion ($\mathbf{x}$), transformation function T is capable of generating an example of how the image would appear ($\mathbf{x}'$) with a raindrop on the camera's lens.

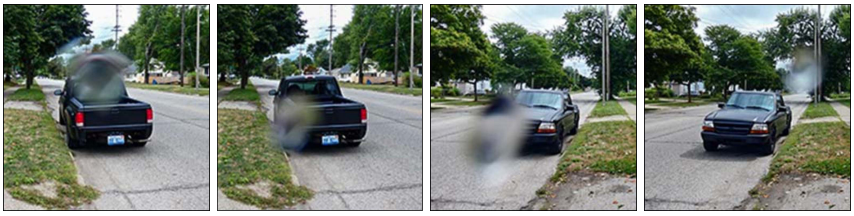
In absence of real-world examples of raindrops occluding the camera's view, synthetic examples are generated via a ray tracing[2] method. Modeling the visual appearance of rain to a high degree of detail has been studied extensively [67]. However, these models are often complex and computationally expensive when applied to datasets as large as those normally required for training or evaluating a DNN. Raindrops falling onto a camera lens can either settle as stationary pools of water or even produce streaks, depending on factors such as the angle of gravity, lens curvature, and wind. For simplicity, this chapter assumes raindrops are hemispherical and stationary on the lens (i.e., circular). These simulated raindrops can be described by the following properties: position, radius, and blur. For comparison, Figure 3.2 shows example images taken with a clean lens (Figure 3.2a), images of occluding real-world water droplets from the same view (Figure 3.2b), and synthetic droplets inserted into the scene (Figure 3.2c). Since the appearance of real-world raindrops include rays of light coming from sources outside of the camera's view, these simulated raindrops may not completely replicate all details of the real-world phenomenon. For cases where more realistic raindrop phenomena are needed, an alternative method for raindrop simulation may be used; however, for this study, it is assumed that the "reality gap" [96] is sufficiently small to ignore.

3.2.2 Assessing System Behavior When Exposed to an Uncertain Condition

To assess a DNN under a new, previously-unseen environmental condition, synthetic examples are generated using contexts (e.g., alternative raindrop variations) evolved by ENKI to produce a wide range of behavior for the DNN. Synthetic test data is generated by transforming default test data with the contexts evolved by ENKI. A data flow diagram (DFD) for this stepwise process is depicted in Figure 3.3, where circles represent computational steps, boxes represent external entities, arrows

[2]Ray tracing is a method of image generation that simulates photorealistic effects by tracing the path rays of light project in a scene from a specific viewpoint (the image plane) back to the originating light sources [91].

(a) No Droplets



(b) Real Droplets

(c) Simulated Droplets

Figure 3.2: Examples of real-world and simulated water droplets. Images in (a) are not obscured by water droplets on the lens. Images in (b) have had real water droplets placed on the lens. Images in (c) show simulated droplets superimposed over the original images.

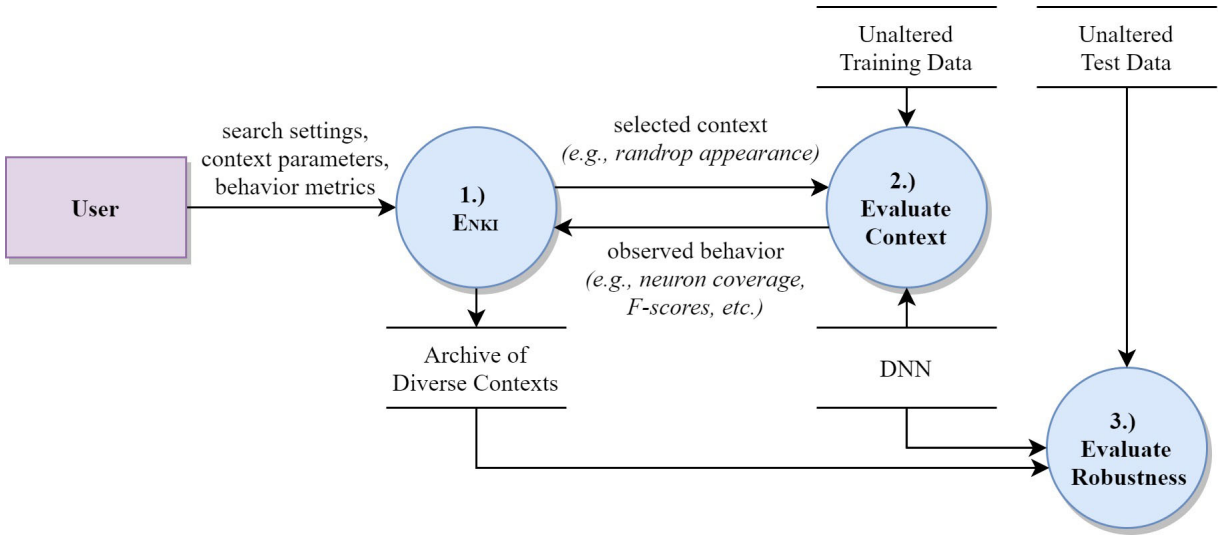


Figure 3.3: DFD for using ENKI to assess a DNN with an uncertain environmental condition. Circles represent computational steps, boxes represent external entities, parallel lines mark persistent data stores, and connecting arrows show the flow of data between steps.

represent data flow, and data stores are represented within parallel lines. Each step is described in turn as follows.

Step 1. Enki

Enki searches a space of possible contexts for the environmental condition of interest (e.g., every possible variation of a raindrop), assesses the behavior of the DNN under each selected context, and then archives the most diverse contexts based on the DNN's observed behavior. The intent is to apply this technique for situations where the full impact of an environmental condition is not known for the DNN, and it is not possible to exhaustively assess every possible context. An evolution-based search procedure enables Enki to evaluate a population of multiple contexts for the DNN in parallel and guide the population towards the most diverse selection of contexts thus far observed. Enki requires a specification for the environmental condition and a specification for the system behavior to be observed.

Step 1.1. Evolutionary Search

Each individual operational context is defined by a set of independent parameter values (**g** in Equation (3.1)) that describe characteristics of the operational environment. The user must specify a name for each parameter and permissible values. Each individual context generated by Enki is a sample taken from these specified value ranges. Parameter values may be real, integer, or categorical. Sampled values are then encoded by Enki into a numeric vector form, designated as a *genome*, with normalized values ranging between 0 and 1, based on the user-specified value ranges. For the raindrop occlusion condition, contexts can be specified by x, y, $radius$, and $blur$ parameters, where a specific context corresponds to explicit values selected for these parameters. For example, the context $\{x = 0.5, y = 0.5, radius = 0.1, blur = 0.02\}$ would correspond to a raindrop in the center of an image, radius covering 10% of the image, and with a small amount of blur.

Enki evaluates a DNN's behavior when exposed to each generated context (see Step 2 for details about behavior metrics). With each generated context, a user-defined evaluation procedure

is executed to monitor the behavior of the DNN according to the given behavior metrics. The results are then encoded into a numeric vector form, designated as a *phenome*, with normalized values ranging between 0 and 1.

ENKI uses an EA (Algorithm 3.1), to explore the space of possible operational contexts and select those that lead to the most diversity. Diversity is determined by comparing the phenomes (i.e., system behavior) associated with each context within a *population* (see Step 1.3 for details about how diversity is computed). The initial population is created from random-generated contexts. With each generation of evolution, the population is said to become more diverse when the phenome of each context becomes less similar to its nearest neighbors. A very diverse population of contexts will each produce mutually unique system behavior (regardless of how similar the operating environment may appear between contexts). For each generation, the population is also compared to a separate archive that is maintained to track the least similar contexts discovered.

Standard evolutionary operations are used to evolve the population's genomes [52], such as *selection*, *recombination*, and *mutation*, over multiple generations to guide the population towards contexts that improve the phenome diversity of the archive. Round-robin *tournament selection* chooses which *parent* genomes is recombined, favoring genomes that result in more diverse phenomes (see Step 1.3). Random *single-point crossover* produces *offspring* genomes by merging the beginning of one parent's genome with the end of the other's at a random selected point within the genomes. Finally, *creep mutation* is implemented to slightly increase or decrease the value of each element in an offspring genome according to a *mutation rate* probability. The amount of change for each mutated element is randomly drawn from a uniform distribution bounded by a *mutation shift* variable.

Step 1.2. Commit to Archive

For each generation, the current population is evaluated and compared to an archived collection of contexts. This archive preserves the least similar contexts across generations (with respect to how the DNN's behavior is affected), and each context in the current population is ranked according to

how it compares to those in the archive. Contexts that exceed a *novelty threshold* are added to the archive. When the archive is full, it is truncated by discarding the archived contexts that contribute the least to its overall diversity.

Step 1.3. Rank Individuals

Enki ranks each individual context according to how similar its phenome (i.e., system behavior) is to its nearest neighbors in the archive, favoring those that are less similar. Similarity is determined by the Euclidean distance between phenomes as follows,

$$\text{novelty}(\mathbf{p}, \mathbf{P}, k) = \text{mean}\Big(\text{min-k}\big(\|\mathbf{p} - \mathbf{p}_i\|^2 \; \forall \mathbf{p}_i \in \mathbf{P} : \mathbf{p}_i \neq \mathbf{p}, k\big)\Big) \tag{3.2}$$

where the *novelty score* is computed as the average distance between a phenome ($\mathbf{p}$) with its k nearest neighbors in the set of all archived phenomes ($\mathbf{P}$). Contexts with phenomes that are more distant from the closest matching phenomes in the archive (i.e., less similar) are given priority. By focusing on *phenome diversity* over *genome diversity*, Enki is able to uncover and archive cases where even slight changes to the operational context produce significantly different system behavior (e.g., adversarial examples for DNNs).

Figure 3.4 plots how novelty scores for archived contexts typically evolve over each generation. The blue curve shows the mean novelty score for all archived contexts. The green shaded area shows the range between the minimum and maximum archived novelty score. An increasing mean novelty score indicates that archived contexts have become more distant from each other, and narrowing range of novelty scores indicates contexts have become more uniformly spaced.

Step 2. Evaluate Context

To assess the effect a specific context has on the DNN, a subset of the original, unaltered training data is selected for use as a basis for evaluation. Using the transformation function (Equation (3.1)) for the environmental condition of interest, synthetic examples ($\mathbf{x}'$) are generated by applying the parameter values associated with a given context ($\mathbf{g}$) to each sampled training example ($\mathbf{x}$). The behavior of the DNN is then observed when processing the entire set of synthetic examples. This

procedure is described by Algorithm 3.2. Three behavior metrics are considered for this study: the DNN's classification *F-scores*, neuron activation *coverage*, and neuron activation *pattern*. Alternative metrics may also be considered for future investigation, such as the Kullback-Leibler divergence [203] or the Receiver Operating Characteristic (ROC) curve [203].

Classification accuracy is defined as the ratio of correct classifications to the total number of classifications made. However, when considered alone, accuracy can be a misleading metric, especially when classification categories are not equally balanced. Therefore, *F-scores* are considered for each classification category, defined as the harmonic mean of the *precision* and *recall* for each respective category [192].

A DNN's neuron activation coverage is computed by monitoring the cumulative output of each neuron in each activation layer of the DNN and then computing the ratio of activated neurons to the entire set of neurons. Thus, the neuron coverage indicates the fraction of the network exercised by the entire set of given synthetic images. To determine when a neuron is activated, an *activation threshold* must be defined, and any neuron output with an absolute value that exceeds the threshold is considered activated. DNNs for this study primarily use the rectified linear unit (ReLU) [157] activation function (commonly used for image processing DNNs [120]), and neurons are considered activated with any output value greater than zero.

Neuron activation patterns are computed in a manner similar to the neuron coverage. A vector is constructed with elements having a one-to-one correspondence to each neuron in each activation layer of the DNN. Elements corresponding to activated neurons are assigned a value of 1, whereas all others are assigned a value of 0. These patterns are observed as a means for measuring which specific portions of the DNN are exercised by the given synthetic images, analogous to the execution paths for a traditional software program. Including the activation pattern as a metric for ENKI to diversify will encourage each archived context to activate mutually unique sections of the DNN under test. Figure 3.5 illustrates two examples of possible activation patterns for the same simple neural network. Activated neurons (depicted as green) contribute to the final output, while non-activated neurons have zero contribution.

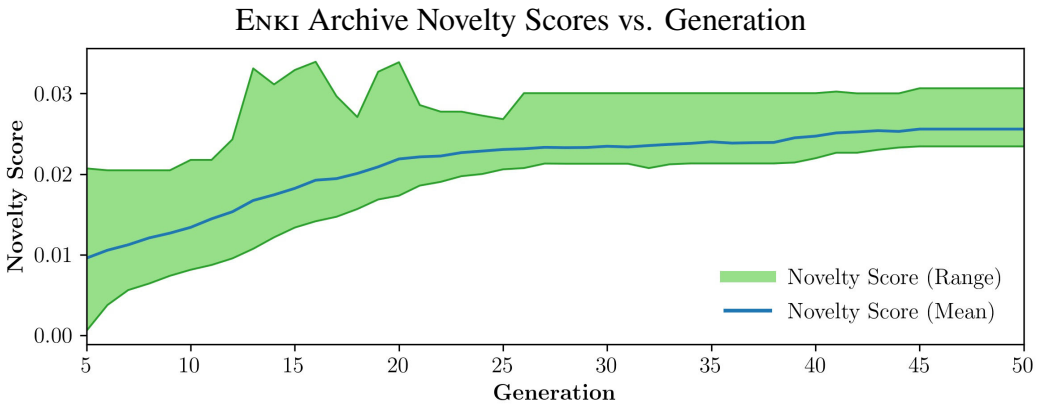
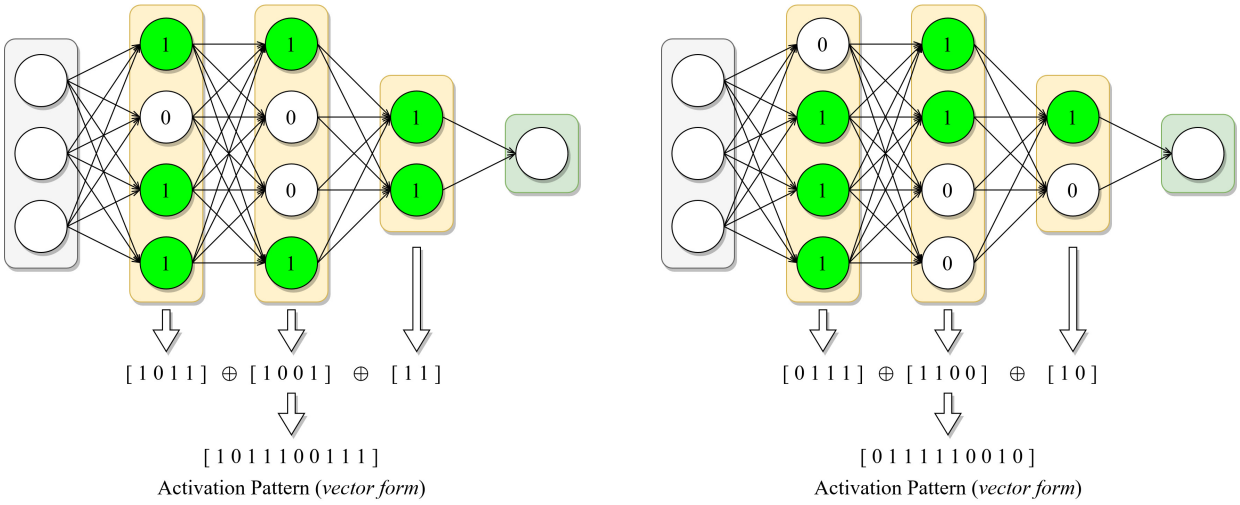


Figure 3.4: Plot of novelty scores for ENKI-generated raindrop contexts over 50 generations. The blue curve shows the mean novelty score of all contexts archived. The green-shaded area shows the bounds for the highest and lowest novelty score in the archive.



(a) Example 1

(b) Example 2

Figure 3.5: Example activation patterns for the same DNN. Activated neurons (marked green), are assigned a value of 1, and inactive neurons are assigned a value of 0. Finally, all designated values are concatenated into a vector form.

Algorithm 3.1 ENKI

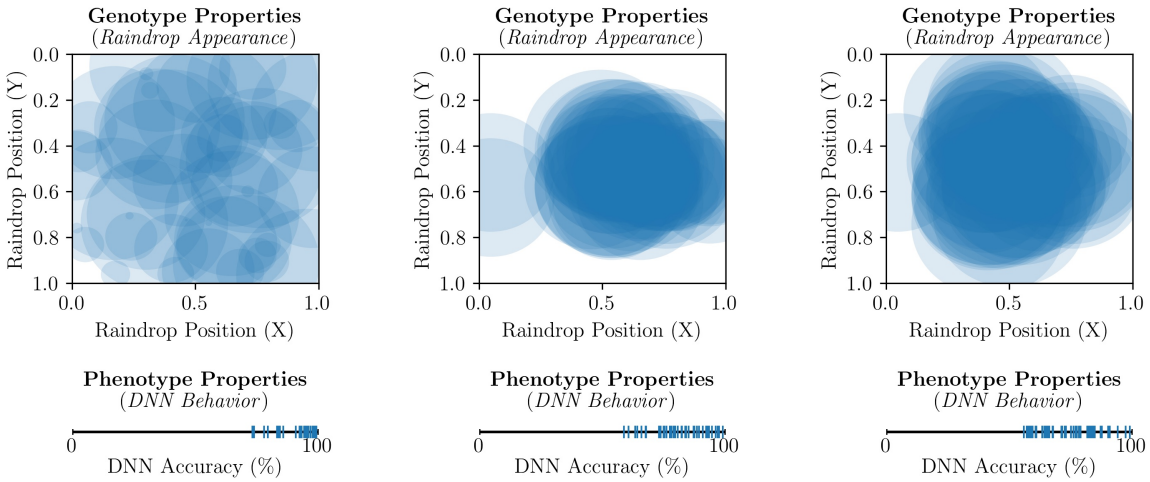
```
function EVOLUTIONARY-SEARCH(n_generations, eval_func)
    archive ← ∅
    pop ← RANDOM-POPULATION()              ▷ Initialize population with random genomes.
    for 0 to n_generations do              ▷ Evolve over given number of generations.
        pop ← SELECTION(pop)               ▷ Select genomes via tournament selection.
        pop ← RECOMBINATION(pop)           ▷ Recombine genomes via single-point crossover.
        pop ← MUTATION(pop)                ▷ Mutate genomes via creep mutation.
        pop ← EVALUATION(pop, eval_func)   ▷ Evaluate phenomes with given procedure.
        archive, pop ← COMMIT-TO-ARCHIVE(archive, pop)
    return archive
function COMMIT-TO-ARCHIVE(archive, pop)
    scores ← RANK-INDIVIDUALS(archive, pop)     ▷ Compute novelty scores via Eq. (3.2).
    pop ← pop : scores > novelty_threshold      ▷ Filter out any below minimum threshold.
    archive ← TRUNCATE(archive ∪ pop)           ▷ Combine and truncate to desired size.
    return archive, pop
```

Algorithm 3.2 Evaluate Context

```
function EVALUATE-CONTEXT(context)
    train_subset ← SAMPLE-TRAINING-DATA()              ▷ Sample unaltered training data.
    synth_subset ← TRANSFORM(context, train_subset)    ▷ Transform w/ the given context.
    behavior ← EVALUATE-DNN-BEHAVIOR(synth_subset)     ▷ Evaluate synthetic data.
    return behavior
```

Since ENKI's evolved contexts are encouraged to exhibit mutually unique behavior in the DNN (i.e., phenome diversity), each context exercises different aspects of the DNN regardless of how similar operational characteristics may be between test cases (i.e., genome similarity). For example, Figure 3.6 illustrates both genotypic and phenotypic properties of each raindrop context evolved by ENKI for the raindrop occlusion condition. Plots on top depicted archived contexts as circles with a position and radius in proportion to each occluding raindrop. On bottom, the DNN accuracy is shown for processing images under each context. ENKI focuses on evolving an archive of raindrops that have a diverse impact on the DNN rather than having diverse appearances. After 30 generations, ENKI found that large raindrops towards the center the of the image produced the most varied behavior for the DNN. Furthermore, since ENKI's objective is diversity (rather than optimizing test cases to favor any specific behavior metric), the resulting suite of test cases are not expected to be strictly *failure* cases. Contexts generated by ENKI are expected to cover a wide range

for each behavior metric, and therefore, the resulting contexts enable an assessment of the DNN that ignores any bias towards a specific level of performance or any bias towards the likelihood of any specific context to occur.



(a) Generation 1 (b) Generation 15 (c) Generation 30

Figure 3.6: Plots of Enki-archived raindrops, evolved over 30 generations. On top, raindrops are plotted as overlapping circles at the same relative image position and with the same relative raindrop radius. On bottom, accuracy is shown for when the DNN is exposed to each raindrop. Enki starts with a random assortment of raindrops and evolves raindrops that produce diverse effects on the DNN, rather than evolving raindrops with diverse appearances. After 30 generations, Enki found that larger raindrops towards the center created the widest distribution of DNN behavior.

Step 3. Evaluate Robustness

To evaluate the robustness of a DNN to a specific environmental condition, each Enki-generated context is applied to the default test data. A robustness measure (ψ) is then computed for each test input with the following formula [230],

$$\psi(x) = \frac{1}{\max\limits_{\delta \in set} D_{KL}\left(P(x), P(x+\delta)\right)} \tag{3.3}$$

where D_{KL} is the Kullback-Leibler divergence[3] between the DNN's predictions on default test data, $P(x)$, and its predictions on synthetically altered data, $P(x+\delta)$. A DNN's robustness for any given input (x) is the inverse of the maximum D_{KL} found for the entire set of generated contexts.

[3] *Kullback-Leibler divergence* is a measure of the *relative entropy* between two probability distributions [203].

Thus, a more robust DNN implies that its output has been disturbed less when exposed to the noise introduced by the given contexts. To determine the robustness of a set of test inputs ($X = [x_0 ... x_n]$), the mean robustness is taken over the whole set. The entire evaluation procedure is shown in Algorithm 3.3.

Algorithm 3.3 Evaluate Robustness

```
1:  function EVALUATE-DNN-ROBUSTNESS(dnn, test_data, contexts)
2:      dataset_robustness ← ∅
3:      for x in test_data do                          ▷ Iterate through each test example.
4:          d_max ← 0
5:          y ← EVALUATE-DNN(dnn, x)                   ▷ Evaluate the unaltered example.
6:          for context in contexts do                 ▷ Iterate through each given context.
7:              x′ ← TRANSFORM(context, x)             ▷ Transform x into synthetic example x′.
8:              y′ ← EVALUATE(dnn, x′)                 ▷ Evaluate the synthetic example.
9:              d ← KL-DIVERGENCE(y, y′)               ▷ Compute divergence for synthetic example.
10:             d_max ← MAX(d, d_max)                  ▷ Track which context created the most divergence.
11:         robustness ← 1/d_max
12:         dataset_robustness ← dataset_robustness ∪ robustness
13:     return MEAN(dataset_robustness)
```

3.2.3 Improving System Robustness for Exposure to an Uncertain Condition

Aside from using the diverse contexts from Enki to generate test data for the DNN, additional steps can be taken to create a more robust DNN by introducing synthetic training examples, created in a similar manner. Data flow for these additional steps are depicted in Figure 3.7, and descriptions for each step follow.

Step 4. Train Deep Neural Network With Synthetic Data

A new DNN is trained by mixing synthetic training data with the default training dataset during the training phase (Algorithm 3.4). With each epoch, a fraction (ρ) of the training images are chosen to be transformed to match a randomly selected context from Enki's archive. After multiple training iterations (i.e., epochs), the DNN is exposed to each training image in its original form and also a mixture of different synthetic variants (based on Enki's archived contexts). The new DNN is

structurally identical to the original DNN, with the only difference being that the weights chosen by this training procedure is optimized to better handle ENKI's archived contexts, as opposed to the weights chosen for the original DNN, where only the unaltered training data is considered (i.e., the targeted uncertain environmental conditions are not included).

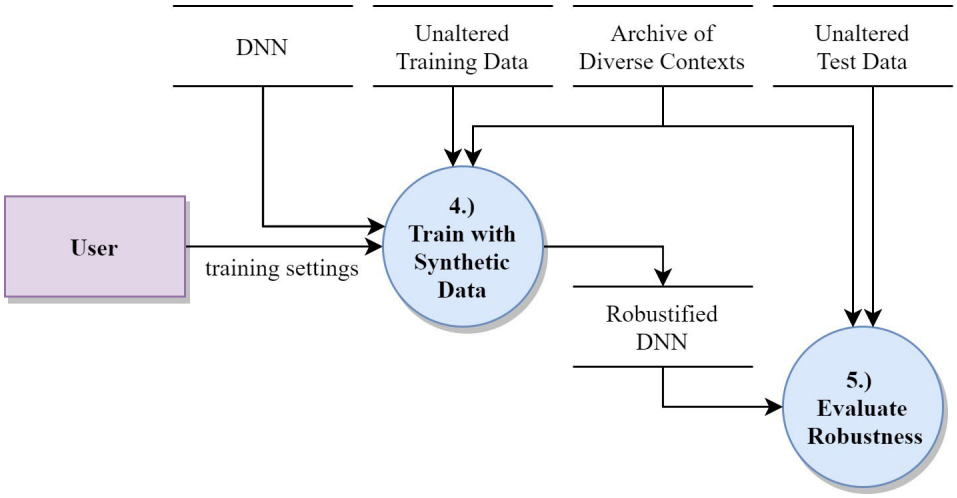


Figure 3.7: DFD for using ENKI-generated contexts to retrain and evaluate a more robust DNN with synthetic data.

Algorithm 3.4 Train DNN with Synthetic Data

```
1:  function TRAIN-WITH-SYNTH-DATA(dnn, train_data, contexts, n_epochs)
2:      dnn ← INIT-WEIGHTS(dnn)                    ▷ Initialize weight values for the given DNN.
3:      for 0 to n_epochs do
4:          synth_data ← MIX-DATA(train_data, contexts)   ▷ Mix synthetic & unaltered data.
5:          dnn ← FIT-WEIGHTS(dnn, synth_data)     ▷ Fit weights to data (gradient descent).
6:      return dnn
7:  function MIX-DATA(train_data, contexts)
8:      mixed_data ← ∅
9:      for x in train_data do
10:         if RANDOM() > ρ then                   ▷ Transform according to transformation rate ρ.
11:             context ← SELECT-RANDOM(contexts)  ▷ Select random context.
12:             x ← TRANSFORM(context, x)          ▷ Transform data with selected context.
13:     mixed_data ← mixed_data ∪ x                ▷ Add current example to set of mixed data.
14:     return mixed_data
```

Additional factors may be considered when retraining DNNs with synthetic data in practice. To help reduce overfitting to the contexts archived by ENKI, some fuzzing could be introduced when data is transformed (Algorithm 3.4, Line 20). For this study, fuzzing has not been introduced,

in order to fairly compare the quality of the exact contexts selected by Enki. Furthermore, this study assumes $\rho = 0.5$ to give equal balance to unaltered and synthetic training examples. Alternative values of ρ will bias the training to either emphasize or de-emphasize the presence of the environmental condition of interest. Finally, additional constraints should be considered when applying a specific context to a specific training input. For example, when applying raindrops to an image, an additional step should be taken to ensure that the photographed object of interest is not completely obscured by the introduced raindrop. In cases where it is completely obscured, the resulting synthetic image should not be included in the training dataset.

Step 5. Evaluate Robustness

Since the new DNN is structurally identical to the original DNN (i.e., trained only on default data), it can be evaluated with the same procedure as Step 3 (Algorithm 3.3), where synthetic test data is generated to reflect the contexts archived by Enki. By using the same evaluation procedure, the performance results of the new DNN can be directly compared to the original DNN.

3.3 Empirical Validation

This section presents results from experiments conducted with Enki on DNNs trained with the CIFAR-10 and KITTI datasets. For these experiments, Enki has been configured with the settings in Table 3.1. For comparison, experiments have also been conducted using DeepTest [211] and a random (i.e., Monte Carlo) generation method. Results from these experiments answer the following research questions with accompanying *null hypotheses* (H_0) and *alternative hypotheses* (H_1):

RQ1.) Can we use Enki to identify gaps in training that cause an LES to perform poorly?

H_0: Testing with Enki-generated data does not significantly decrease DNN performance.

H_1: Testing with Enki-generated data significantly decreases DNN performance.

RQ2.) Can we use ENKI to improve the robustness of an LES to the effects of uncertain environmental conditions?

H_0: Retraining with ENKI-generated data does not significantly improve DNN robustness.

H_1: Retraining with ENKI-generated data significantly improves DNN robustness.

Synthetic test datasets resulting from each method are labeled T with a corresponding subscript to indicate the source method for generation (e.g., T_{DEEP}, T_{ENKI}, and T_{RAND} to represent DeepTest-generated, ENKI-generated, and random-generated test data, respectively). Similarly, DNNs are labeled with a subscript to indicate a source for synthetic training data (e.g., DNN_{ENKI} is trained with ENKI-generated data). Default (i.e., unaltered) test datasets are referred to as T_{BASE}, and DNNs trained only on unaltered training data are referred to as DNN_{BASE}. When referring to a specific application, a prefix will be attached (e.g., CIFAR10-DNN_{BASE} and CIFAR10-T_{BASE}). For brevity, the application prefix is omitted when the discussion applies to both applications.

Table 3.1: ENKI Configuration Settings

Setting	Value
num_generations	50 generations
archive_size	50 individuals
population_size	10 individuals
tournament_size	3 comparisons
mutation_rate	14%
mutation_shift	20%
num_nearest_neighbors	3 individuals

In addition to raindrop occlusion (see Section 3.2), additional environmental conditions have been included in these experiments, such as variable brightness and contrast. Transformation functions for these conditions have been implemented using standard image enhancement functions included in the Pillow [143] module for Python. For variable brightness, pixel values for each input image have been altered by a uniform $brightness$ factor, where $brightness = 0$ results in a black image, and $brightness = 1$ results in the original pixel intensity. For variable contrast, pixel values for each input image have been altered by a $contrast$ factor, where $contrast = 0$ results in a gray

image, and $contrast = 1$ results in the original image contrast. Experiments have been conducted with these transformations implemented in isolation as well as in *combination* with the raindrop condition. Parameters with value ranges for all environmental conditions are provided in Table 3.2 when applied to both the CIFAR-10 and KITTI applications.

Table 3.2: Environmental Parameter Ranges

	CIFAR-10	KITTI
Parameter	Value Range	Value Range
$brightness$	0 to 1	0 to 1
$contrast$	0 to 1	0 to 1
$raindrop\_x$	0 to 31 pixels	0 to 1241 pixels
$raindrop\_y$	0 to 31 pixels	0 to 374 pixels
$raindrop\_radius$	0 to 9 pixels	0 to 123 pixels
$raindrop\_blur$	1 to 2 pixels	9 to 18 pixels

3.3.1 Deep Neural Network Applications

For validation, this chapter's approach has been applied to two benchmark applications (described in Chapter 2). Both applications implement image-processing DNNs to learn from labeled training data and make predictions for new data. The first application implements a DNN for *image recognition*, where each image input is a photograph of a single object, and the task is to classify the type of object photographed. The second application implements a DNN for *object detection*, where each image input is a photograph of zero or more objects, and the task is to both locate and classify each object photographed.

The focus of this study is how to assess and improve any pre-trained DNN to conditions *not* covered by existing training or validation data. A baseline CIFAR10-DNN$_{\mathrm{BASE}}$ has been trained on default training images to achieve a 91% accuracy on default test images. Similarly, a baseline KITTI-DNN$_{\mathrm{BASE}}$ has been trained on default training images to achieve a 88% recall on default test images. Any specific specializations in architecture or hyper-parameter selections that might incrementally improve performance of the described DNNs on default test data is considered

tangential to this study. The objective of this study is to assess and improve each DNN's performance when new environmental phenomena are introduced into each respective dataset.

3.3.2 Evaluation of System Behavior Under Uncertain Environmental Conditions

To assess the performance of CIFAR10-DNN$_{\mathbf{BASE}}$ and KITTI-DNN$_{\mathbf{BASE}}$ under each environmental condition, ENKI creates an archive of diverse contexts for each respective condition. The T$_{\mathbf{ENKI}}$ synthetic test dataset was created by applying ENKI-generated contexts to the default test data. Synthetic test datasets T$_{\mathbf{RAND}}$ and T$_{\mathbf{DEEP}}$ were also created for comparison. T$_{\mathbf{RAND}}$ was created by applying random-generated contexts to the default test data, and T$_{\mathbf{DEEP}}$ was created by using the DeepTest method [211].

3.3.2.1 System Performance on Synthetic Test Data

To address the first research question (***RQ1***), the quality of synthetic test data produced by ENKI (T$_{\mathbf{ENKI}}$) is compared to alternative test datasets (T$_{\mathbf{BASE}}$, T$_{\mathbf{DEEP}}$, and T$_{\mathbf{RAND}}$) under each environmental condition introduced to both CIFAR-10 and KITTI datasets. The null hypothesis (H_0) is that there is no difference between the mean test performance of each DNN$_{\mathbf{BASE}}$ for T$_{\mathbf{ENKI}}$ when compared to T$_{\mathbf{DEEP}}$ and when compared to T$_{\mathbf{RAND}}$ (i.e., H_0: $\mu_{\text{diff}} = 0$). The alternative hypothesis (H_1) is that the mean test performance of each DNN$_{\mathbf{BASE}}$ will be less for T$_{\mathbf{ENKI}}$ when compared to T$_{\mathbf{DEEP}}$ and T$_{\mathbf{RAND}}$ (i.e., H_1: $\mu_{\text{diff}} > 0$). To validate the statistical significance of these results and test each hypothesis, paired samples t-tests [214] have been conducted to reject H_0 and accept H_1 when comparing T$_{\mathbf{ENKI}}$ versus T$_{\mathbf{DEEP}}$ and T$_{\mathbf{ENKI}}$ versus T$_{\mathbf{RAND}}$.

Plots in Figure 3.8 and Figure 3.9 show the respective performance of CIFAR10-DNN$_{\mathbf{BASE}}$ and KITTI-DNN$_{\mathbf{BASE}}$ when evaluated with both unaltered and synthetic test data from each method. Results shown are the mean over multiple trials to account for variation (a 95% confidence interval is shown by whiskers to illustrate variation). Figure 3.8 shows the mean accuracy of CIFAR10-DNN$_{\mathbf{BASE}}$ for test datasets generated for each environmental condition in isolation (e.g., *Brightness*, *Contrast*, and *Raindrop*) as well as in combination. For all trials, CIFAR10-DNN$_{\mathbf{BASE}}$

had the lowest mean accuracy for CIFAR10-T_{ENKI}. Similarly, Figure 3.9 shows the mean recall of KITTI-DNN_{BASE} for each test dataset and each environmental condition. KITTI-DNN_{BASE} had the lowest mean recall for KITTI-T_{ENKI}.

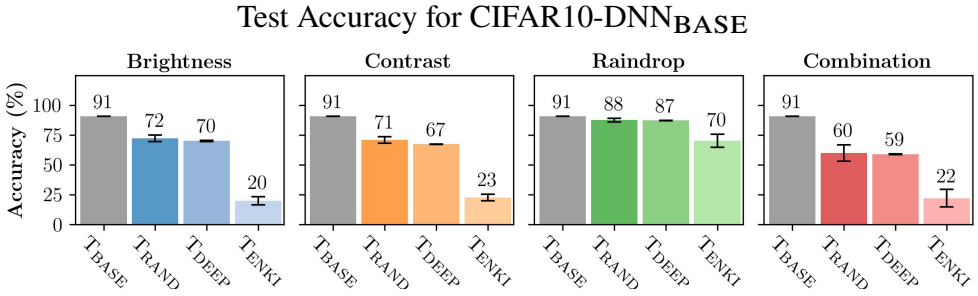


Figure 3.8: Test accuracy of CIFAR10-DNN_{BASE} for each respective test dataset and each environmental condition. Displayed values are the mean over multiple trial runs, with 95% confidence intervals shown by whiskers.

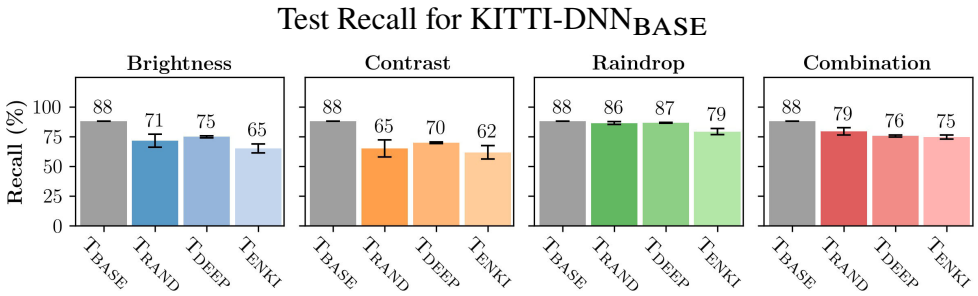


Figure 3.9: Test recall of KITTI-DNN_{BASE} for each respective test dataset and each environmental condition. Displayed values are the mean over multiple trial runs, with 95% confidence intervals shown by whiskers.

Tables 3.3 and 3.4 provide t-test results and corresponding p-values for these experiments. In Table 3.3, the value μ_{diff} is the mean difference in CIFAR10-DNN_{BASE} accuracy for each pair of generated test datasets over multiple trials, and σ_{diff} is the corresponding standard deviation. In Table 3.4, μ_{diff} is the mean difference in KITTI-DNN_{BASE} recall, and σ_{diff} is the corresponding standard deviation. For both CIFAR-10 and KITTI experiments, there is strong evidence ($p \leq 0.01$) to reject H_0 and support H_1. Therefore, these results show that ENKI was able to generate test examples that cause both DNN's to perform more poorly than T_{BASE}, T_{DEEP}, and T_{RAND}, thus addressing ***RQ1***.

Table 3.3: Paired Samples t-Test For Significance of CIFAR10-T_{ENKI} Accuracy Comparisons

	CIFAR10-T_{ENKI} vs. T_{DEEP}				CIFAR10-T_{ENKI} vs. T_{RAND}			
	μ_{diff}	σ_{diff}	t	p	μ_{diff}	σ_{diff}	t	p
Brightness	50.34	1.86	85.59	**<0.01**	52.79	0.62	269.11	**<0.01**
Contrast	44.58	1.38	97.07	**<0.01**	48.38	1.12	136.56	**<0.01**
Raindrop	16.88	2.71	18.71	**<0.01**	17.85	2.12	26.61	**<0.01**
Combination	37.59	4.07	29.19	**<0.01**	37.93	1.49	80.38	**<0.01**

Table 3.4: Paired Samples t-Test For Significance of KITTI-T_{ENKI} Recall Comparisons

	KITTI-T_{ENKI} vs. T_{DEEP}				KITTI-T_{ENKI} vs. T_{RAND}			
	μ_{diff}	σ_{diff}	t	p	μ_{diff}	σ_{diff}	t	p
Brightness	9.71	1.60	14.91	**<0.01**	6.33	1.25	12.40	**<0.01**
Contrast	8.09	2.66	7.46	**<0.01**	3.12	1.19	6.40	**<0.01**
Raindrop	7.37	1.20	13.78	**<0.01**	7.16	0.74	21.50	**<0.01**
Combination	0.99	0.45	4.40	**0.01**	4.89	0.81	12.12	**<0.01**

3.3.2.2 Analysis of Resulting System Behavior

To illustrate the diversity of DNN behavior produced by ENKI-generated contexts, both Figures 3.10 and 3.11 compare random-generated and ENKI-generated contexts from each trial run. Results of the neuron coverage and accuracy of CIFAR10-DNN_{BASE} for random-generated and ENKI-generated contexts are shown in Figures 3.10a and 3.10b, respectively. Each point corresponds to an individual context. Bold (i.e., colored) points show contexts from a single trial run, while gray points show contexts from all trial runs. Regression lines are drawn as dashed lines, with a *Pearson's correlation coefficient*[4] (r) computed for each set of results and displayed in the bottom corner of each plot. The overall mean accuracy is marked by a vertical dotted line for each plot, and for each environmental condition. Similarly, Figures 3.11a and 3.11b plot the relationship between neuron coverage and recall of KITTI-DNN_{BASE} for random-generated and ENKI-generated contexts. Contexts generated by ENKI (Figure 3.10b and Figure 3.11b) are more evenly distributed along both axes of each plot, such that each context activates a different per-

[4]The *Pearson's correlation coefficient* is a measure of the linear association of two variables, ranging between -1 and 1 [110].

centage of neurons and provides a different level of difficulty for the DNN. In contrast, a majority of the random-generated contexts are focused on the upper-right corner of each plot, resulting in a higher percentage of neurons covered and fewer misclassifications. These results imply that a random selection of environmental properties is not as likely to produce challenging test examples for CIFAR10-DNN$_{\text{BASE}}$ or KITTI-DNN$_{\text{BASE}}$, in comparison to test examples from ENKI.

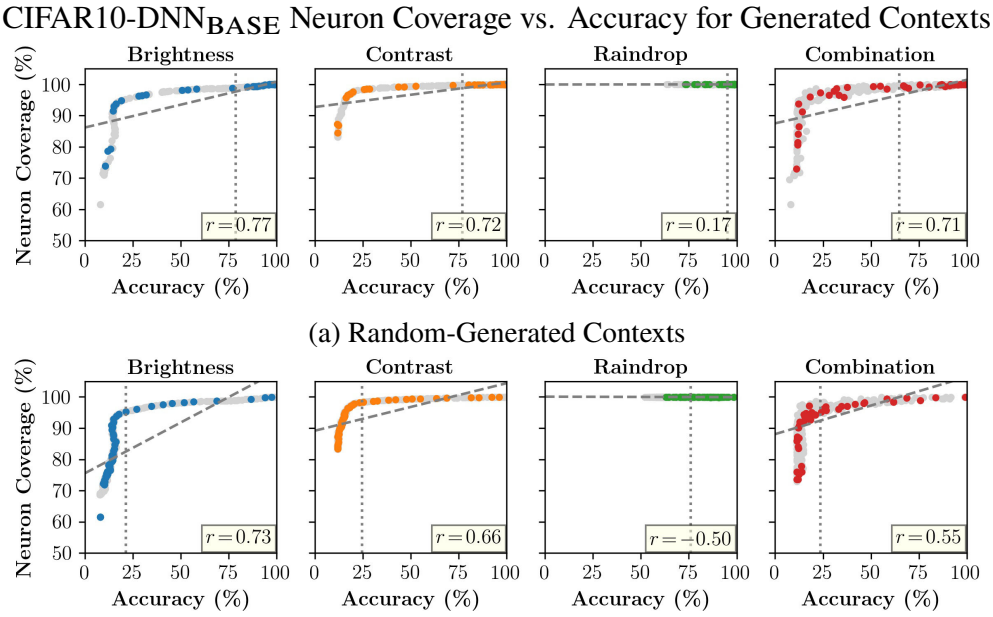
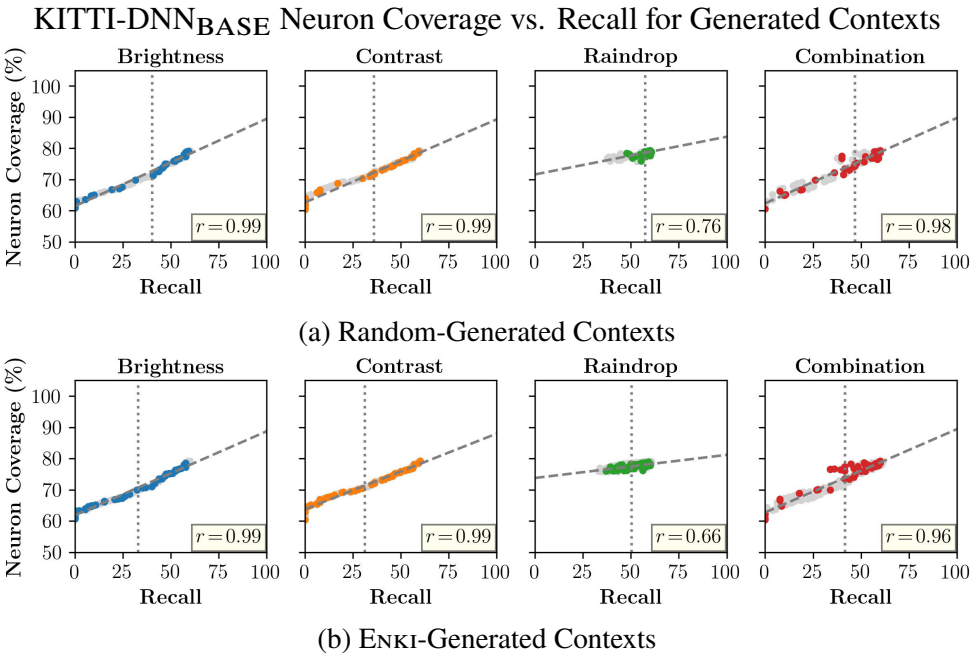


Figure 3.10: Random-generated (a) and ENKI-generated (b) contexts are plotted to show the relationship between the neuron coverage and accuracy of CIFAR10-DNN$_{\text{BASE}}$ under each respective environmental condition. Bold (i.e., colored) points show contexts resulting from a single trial run. Dashed lines are regression lines with a Pearson's correlation coefficient (r). Dotted vertical lines mark the mean accuracy over all contexts. Compared to random-generation, ENKI-generated contexts are more evenly distributed and result in more misclassifications overall (i.e., lower accuracy).

For the variable brightness and contrast conditions, there is a strong positive correlation ($r > 0.5$) between neuron coverage and accuracy, which implies that more challenging contexts are likely to activate fewer neurons in the DNN. An intuitive rationale for this result is that information is lost when decreasing brightness or contrast, and weaker input signals will result in fewer activated neurons. However, greedy techniques that select test examples by maximizing neuron coverage (e.g., DeepTest) assume adverse properties of the environment will correlate to an increased neuron coverage. Contexts created by ENKI are more evenly distributed along the y-axis (i.e.,

neuron coverage). Since no assumption is made about how neuron coverage relates to failing test examples, more challenging test examples can be found by ENKI. For the raindrop occlusion condition, no significant variation in the DNN's neuron coverage has been observed between contexts, most likely since raindrops only affect a small portion of the source image, and therefore, no strong correlation could be found between neuron coverage and accuracy. When combining all environmental conditions, there is a strong correlation between neuron coverage and accuracy. Finally, in comparison to random-generated contexts, ENKI-generated contexts resulted in the overall least accuracy for CIFAR10-DNN$_{\mathbf{BASE}}$ and overall least recall for KITTI-DNN$_{\mathbf{BASE}}$.



(a) Random-Generated Contexts

(b) ENKI-Generated Contexts

Figure 3.11: Random-generated (a) and ENKI-generated (b) contexts are plotted to show the relationship between the neuron coverage and accuracy of KITTI-DNN$_{\mathbf{BASE}}$ under each respective environmental condition. Bold (i.e., colored) points show contexts resulting from a single trial run. Dashed lines are regression lines with a Pearson's correlation coefficient (r). Dotted vertical lines mark the mean recall over all contexts. Compared to random-generation, ENKI-generated contexts are more evenly distributed and result in more misclassifications overall (i.e., lower recall).

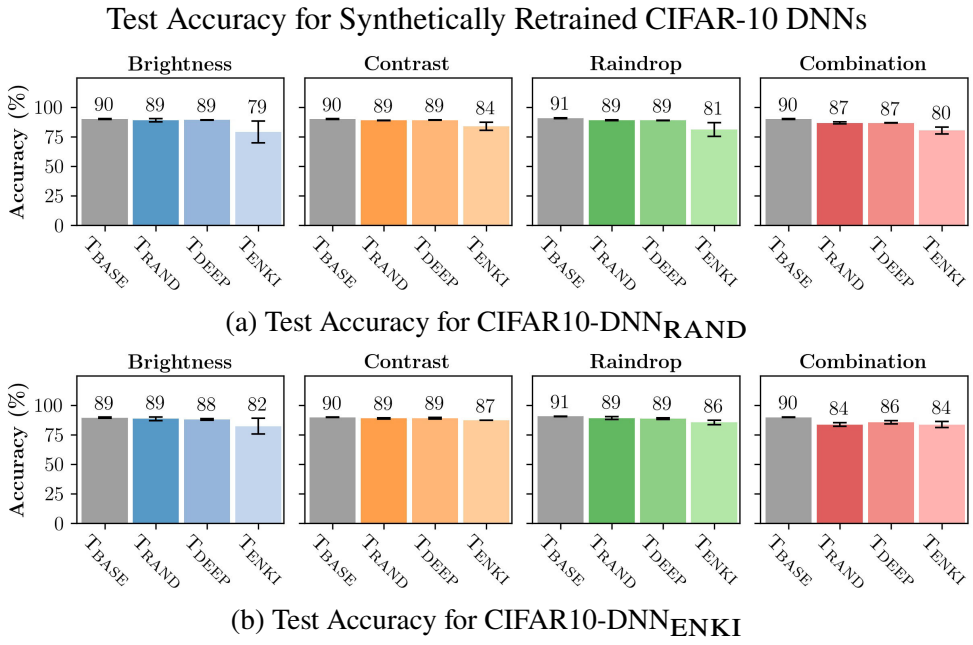
3.3.3 Evaluation of System Robustness

To address the second research question (***RQ2***), the CIFAR-10 and KITTI DNNs were retrained with a mixture of synthetic training data to improve each DNN's performance in the presence

of the introduced environmental conditions (see Section 3.2.3). DNN_{ENKI} was retrained with a mixture of unaltered training images and training images transformed with ENKI-generated contexts. For further comparison of ENKI-generation versus random-generation, experiments have been conducted to compare DNN_{ENKI} to a DNN retrained similarly but with a mixture of random-generated contexts (DNN_{RAND}). The hypothesis for ***RQ2*** is that DNN_{ENKI} will be more robust to each introduced environmental condition when compared to DNN_{BASE}. Therefore, the null hypothesis (H_0) is that there is no difference between the mean robustness of DNN_{ENKI} when compared to DNN_{BASE} (i.e., H_0: $\mu_{\text{diff}} = 0$). The alternative hypothesis (H_1) is that DNN_{ENKI} will be more robust than DNN_{BASE} for each respective environmental condition (i.e., H_1: $\mu_{\text{diff}} > 0$). To validate the statistical significance of these results and test each hypothesis, paired samples t-tests [214] have been conducted to reject H_0 and support H_1 when comparing T_{ENKI} versus T_{BASE}.

Plots in Figures 3.12 and 3.13 show the respective test performance of retrained DNNs (DNN_{ENKI} and DNN_{RAND}) when evaluated with both unaltered and synthetic test datasets. Figure 3.8 shows that both CIFAR10-DNN_{ENKI} and CIFAR10-DNN_{RAND} are more accurate for the introduced synthetic data when compared to CIFAR10-DNN_{BASE}. Similarly, Figure 3.9 shows that both KITTI-DNN_{ENKI} and KITTI-DNN_{RAND} have higher recall for the introduced synthetic data when compared to KITTI-DNN_{BASE}.

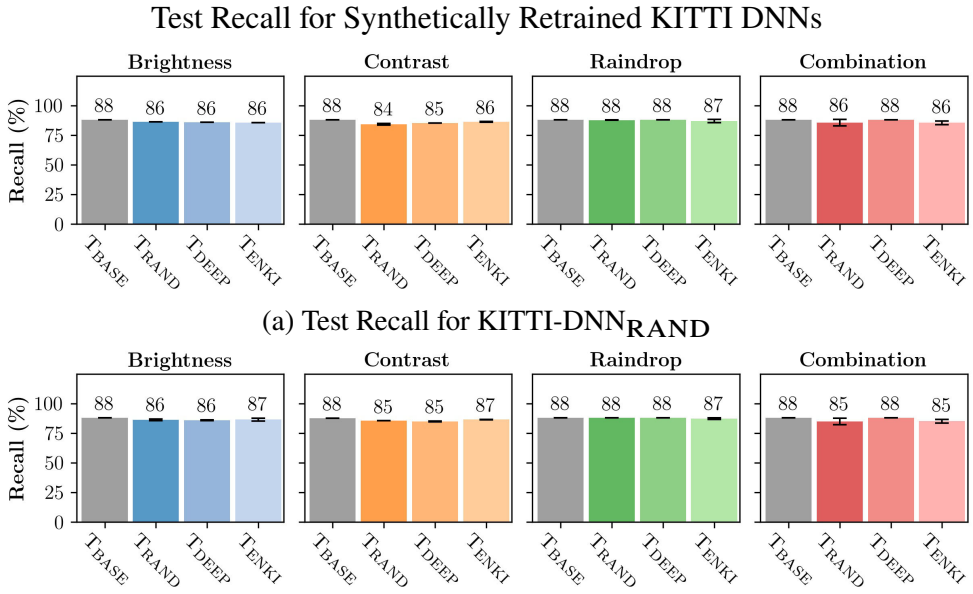
When comparing the robustness of each DNN to the introduced environmental conditions, robustness is evaluated according to Equation (3.3). This form of robustness compares a DNN's output when processing an unaltered image to its output when processing each synthetic variant. A higher robustness value indicates that the DNN's output has been disturbed less by the associated environmental condition. Plots in Figures 3.14 and 3.15 show the robustness of each retrained DNN when exposed to synthetic test datasets for each respective environmental condition. Results are shown as the mean over multiple trials to account for variation (a 95% confidence interval is shown by whiskers to illustrate variation). The mean robustness across all test datasets is shown by the horizontal gray line in each plot (also displayed as μ in the top corner). Both



(a) Test Accuracy for CIFAR10-DNN$_{\mathbf{RAND}}$

(b) Test Accuracy for CIFAR10-DNN$_{\mathbf{ENKI}}$

Figure 3.12: Test performance of CIFAR10-DNN$_{\mathbf{RAND}}$ (a) and CIFAR10-DNN$_{\mathbf{ENKI}}$ (b) for each respective test dataset and for each environmental condition. Displayed values are the mean accuracy over multiple trial runs, with 95% confidence intervals shown by whiskers. When compared to CIFAR10-DNN$_{\mathbf{BASE}}$ (Figure 3.8), synthetic test data accuracy has improved for both CIFAR10-DNN$_{\mathbf{RAND}}$ and CIFAR10-DNN$_{\mathbf{ENKI}}$.



(a) Test Recall for KITTI-DNN$_{\mathbf{RAND}}$

(b) Test Recall for KITTI-DNN$_{\mathbf{ENKI}}$

Figure 3.13: Test performance of KITTI-DNN$_{\mathbf{RAND}}$ (a) and KITTI-DNN$_{\mathbf{ENKI}}$ (b) for each respective test dataset and for each environmental condition. Displayed values are the mean recall over multiple trial runs, with 95% confidence intervals shown by whiskers. When compared to KITTI-DNN$_{\mathbf{BASE}}$ (Figure 3.9), synthetic test data recall has improved for both KITTI-DNN$_{\mathbf{RAND}}$ and KITTI-DNN$_{\mathbf{ENKI}}$.

CIFAR10-DNN$_{\mathbf{BASE}}$ (Figure 3.14) and KITTI-DNN$_{\mathbf{BASE}}$ (Figure 3.15) were observed to be the least robust to the introduced environmental conditions. Introducing random-generated synthetic training data improved DNN robustness, but on average, use of ENKI-generated synthetic training data lead to a more robust DNN.

Tables 3.5 and 3.6 provide t-test results and corresponding p-values for these experiments. The value μ_{diff} is the mean difference in robustness for each pair of DNNs, and σ_{diff} is the corresponding standard deviation. For both CIFAR-10 and KITTI experiments, when comparing DNN$_{\mathbf{ENKI}}$ to DNN$_{\mathbf{BASE}}$, there is evidence ($p \leq 0.05$) to reject H_0 and support H_1. These results show that retraining a DNN with ENKI-generated synthetic training data was able to produce a more robust DNN compared to DNN$_{\mathbf{BASE}}$, thus rejecting H_0 and supporting H_1 for ***RQ2***. Furthermore, these results show that retraining with ENKI-generated synthetic data is more likely to produce a more robust DNN in comparison than retraining with random-generated synthetic data.

Table 3.5: Paired Samples t-Test Results For CIFAR10-DNN$_{\mathbf{ENKI}}$ Robustness Comparisons

	CIFAR10-DNN$_{\mathbf{ENKI}}$ vs. DNN$_{\mathbf{BASE}}$				CIFAR10-DNN$_{\mathbf{ENKI}}$ vs. DNN$_{\mathbf{RAND}}$			
	μ_{diff}	σ_{diff}	t	p	μ_{diff}	σ_{diff}	t	p
Brightness	4.70	1.38	8.98	**<0.01**	0.96	0.37	6.88	**<0.01**
Contrast	4.99	1.20	10.20	**<0.01**	1.68	0.12	35.46	**<0.01**
Raindrop	2.93	0.30	23.91	**<0.01**	0.95	0.55	4.24	**<0.01**
Combination	2.67	0.95	6.88	**<0.01**	0.45	0.28	3.54	**0.01**

Table 3.6: Paired Samples t-Test Results For KITTI-DNN$_{\mathbf{ENKI}}$ Robustness Comparisons

	KITTI-DNN$_{\mathbf{ENKI}}$ vs. DNN$_{\mathbf{BASE}}$				KITTI-DNN$_{\mathbf{ENKI}}$ vs. DNN$_{\mathbf{RAND}}$			
	μ_{diff}	σ_{diff}	t	p	μ_{diff}	σ_{diff}	t	p
Brightness	1.42	0.16	15.33	**<0.01**	0.72	0.24	5.18	**0.02**
Contrast	2.59	1.06	5.45	**<0.01**	0.65	0.12	7.62	**0.04**
Raindrop	1.19	0.24	7.02	**0.05**	2.38	0.33	10.18	**0.03**
Combination	5.76	0.10	82.38	**<0.01**	1.51	0.36	5.88	**0.05**

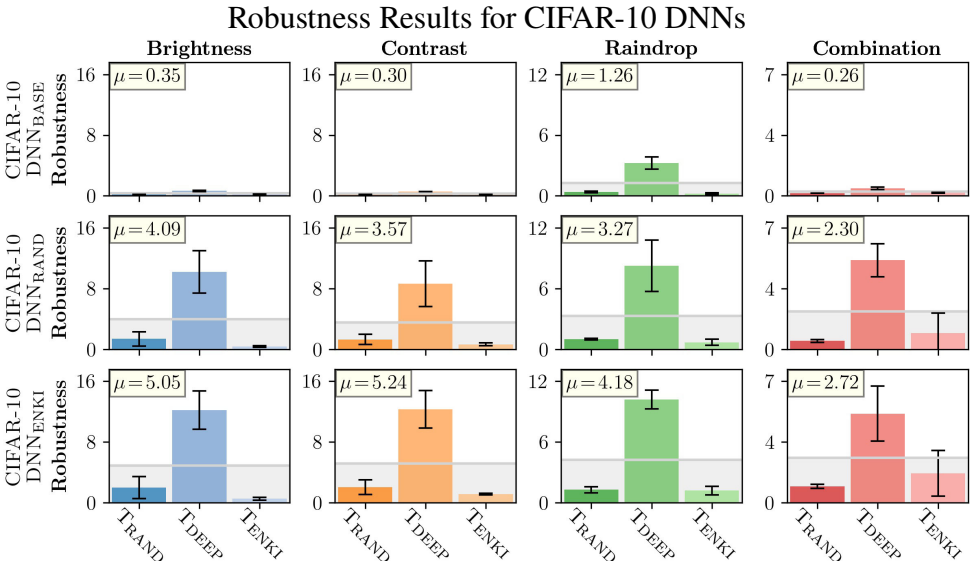


Figure 3.14: Box plots of the robustness of CIFAR10-DNNs (rows) when evaluated against each test dataset for each environmental condition (columns). Robustness (y-axis) was measured over multiple trials for each test dataset (x-axis). Each box shows the interquartile range for the DNN's measured robustness. Medians values are marked orange, and whiskers show the full range. For each plot, the mean (μ) robustness found across all test datasets for an environmental condition is also shown (top-left corner and gray line). On average, CIFAR10-DNN$_{\mathbf{ENKI}}$ was found to be the most robust to each introduced environmental condition.

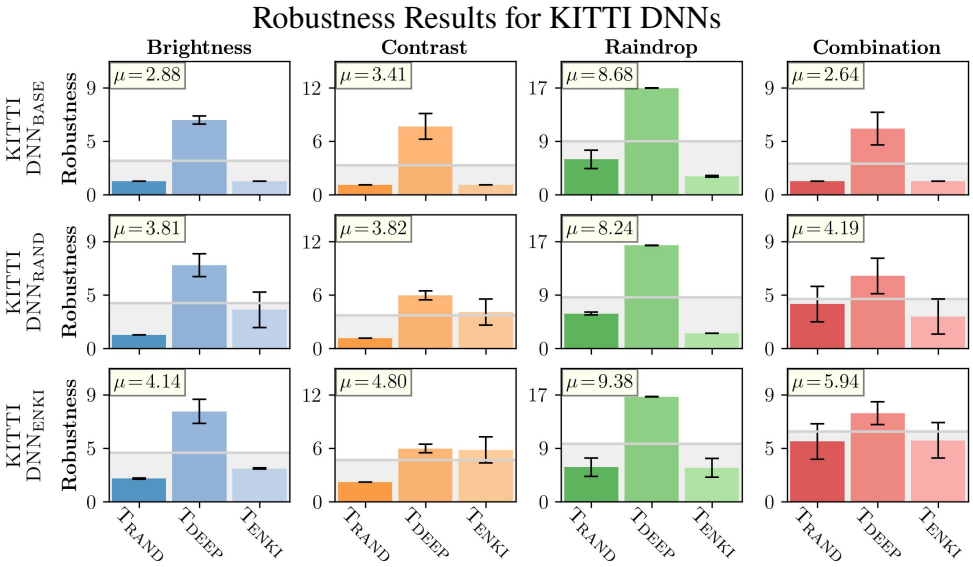


Figure 3.15: Box plots of the robustness of KITTI-DNNs (rows) when evaluated against each test dataset for each environmental condition (columns). Robustness (y-axis) was measured over multiple trials for each test dataset (x-axis). Each box shows the interquartile range for the DNN's measured robustness. Medians values are marked orange, and whiskers show the full range. For each plot, the mean (μ) robustness found across all test datasets for an environmental condition is also shown (top-left corner and gray line). On average, KITTI-DNN$_{\mathbf{ENKI}}$ was found to be the most robust to each introduced environmental condition.

3.3.4 Summary of Results

Both research questions (***RQ1*** and ***RQ2***) can be answered with consistent results from two different applications and their respective datasets (CIFAR-10 and KITTI). To answer ***RQ1***, these results show that ENKI generates test data that is more difficult for the evaluated DNNs, thus highlighting training gaps for the introduced environmental conditions. To answer ***RQ2***, these results show that retraining with ENKI-generated training data leads to a DNN that is more robust to the introduced environmental conditions.

Results from these experiments show that ENKI-generated contexts are more likely to correspond to more difficult test examples. When compared to random-generated and DeepTest test data, ENKI-generated test data caused CIFAR10-DNN_{BASE} and KITTI-DNN_{BASE} to result in more misclassifications. Furthermore, an analysis of the contexts archived by ENKI can assist a developer with identifying environmental properties that lead to DNN failure. As Figure 3.6 shows, ENKI can automatically discover that large raindrops focused on the center of the camera image produced the most varied performance in DNNs. This information could potentially be used by developers or automated processes to pre-emptively identify run-time contexts that lead to DNN failure.

ENKI can also enable a developer to improve the robustness of an LES to the effects of environmental conditions that are absent from default training data. Results from these experiments show that retraining with synthetic training data from ENKI-generated contexts is able to improve the overall accuracy of a DNN in the presence of a newly introduced environmental condition. Furthermore, these results show that DNN_{ENKI} is significantly more robust than DNN_{BASE} and comparable or better than DNN_{RAND} when faced with each respective environmental condition.

3.3.5 Threats to Validity

We consider two main threats to the validity of this study. First, we acknowledge the use of simulation and possible discrepancies with reality (i.e., the "reality gap") [96]. Second, we acknowledge possible variation in results due to stochastic components involved with each method. Each point is addressed in turn.

This chapter's approach depends on a simulation environment capable of emulating sources of uncertainty, and therefore, a reality gap may exist. The quality of synthetic data will need to be assessed to determine if it is sufficiently realistic before use in a safety-critical application. However, even if synthetic data is not used directly to test and retrain the learning component of an LES, the contexts selected by Enki can provide insight to help guide follow-on studies with real-world phenomena. For example, additional images can be taken with real water droplets placed with the same sizes and locations identified by Enki.

All DNN performance results presented are subject to variation, since both the training of DNNs and each data generation method include stochastic functions. To account for variation in these results, multiple independent trials were conducted for statistical analysis (at least ten trials each). Results displayed in Figures 3.8, 3.9, 3.12, 3.13, 3.14, and 3.15 are the mean over multiple trials with confidence intervals displayed to show the degree of variation observed. Finally, paired samples *t*-tests have been conducted to validate the statistical significance of comparisons between generated test datasets and re-trained DNNs.

3.4 Related Work

This section compares this chapter's work to related work in the areas of automated testing with novelty search and automated testing for DNNs.

3.4.1 Automated Testing with Novelty Search

This chapter's approach is inspired by Loki [178], a novelty search method to generate environmental conditions that lead to diverse system behavior as specified by high-level KAOS goal models [43]. Additional research has been conducted on the topic of using novelty search for automated testing to generate test data for object-oriented systems [22, 23]. However, such approaches use novelty search to diversify test cases by their testing parameters rather than by the resulting system behavior. By using the system's behavior as the basis for diversity, Enki is able to discover test cases that may have similar operating conditions but unexpectedly lead to very different system behavior.

3.4.2 Automated Testing for Deep Learning

Alternative methods have been proposed to address the problem of automatic test generation for DNNs. Related techniques, such as DeepXplore [171], DeepTest [211], DeepGauge [144], DeepHunter [226], DeepRoad [235], and DeepConcolic [207], generate test inputs by maximizing structural coverage metrics. DeepXplore and DeepTest measure neuron coverage over the entire DNN for a given test input. However, preliminary results show that test inputs created to maximize neuron coverage are not able to account for all possible behaviors that a DNN may exhibit and that neuron coverage alone can be insufficient for finding corner cases [195]. Finer variants of neuron coverage have also been considered with DeepGauge, where test cases are generated to maximize coverage over subsections or specific activation patterns within a DNN at the layer-level. DeepConcolic supports neuron coverage but also modified condition/decision condition (MC/DC) variants [208]. Further studies have shown that even techniques that attempt to maximize these more fine-grained structural coverage metrics can be misleading, since selected subsets of data with significantly different error rates can produce similar amounts of coverage [137]. Thus, more recent techniques such as DeepGini [56] have shifted focus on prioritizing test cases based on neuron coverage to more statistical performance metrics.

One key difference that differentiates this work from related work in testing DNNs is that this work encourages diversity in terms of how a DNN performs across all generated test cases. In contrast to the related work, Enki explicitly searches for operating contexts that lead to unique system responses. Instead of only discovering test cases that highlight weaknesses in a DNN, Enki can be used to discover a broad range of test cases that may uncover strengths, weaknesses, or otherwise unknown (latent) behavior. Furthermore, Enki's emphasis on diversity identifies more diverse weaknesses, thus enabling better coverage of vulnerabilities. Another key difference between these related techniques and Enki is in the form of each technique's output. The aforementioned techniques generate transformations that are linked to a specific source input, and therefore, no general trends can be inferred from the synthetic data produced. In contrast, Enki generates transformations based on DNN behavior over multiple inputs rather than one specific input, which

enables more generalized patterns to be inferred. By uncoupling problematic test cases from a single source input, ENKI helps developers identify general limitations of a DNN (e.g., what sizes of raindrops are more likely to cause objects to be misidentified).

Though there are different approaches in technique, all related methods for testing DNNs rely on a generative procedure to construct synthetic test data. Methods like DeepTest and DeepXplore depend on a manually crafted procedure to transform real data into a manipulated, synthetic form (e.g., an image processing procedure to add fog to an existing image). Using a rule-based algorithmic transformation procedure gives developers direct control over how synthetic data is generated, but it also relies on the assumption that a transformation procedure can be developed to accurately reproduce the desired phenomenon. Alternatively, methods like DeepRoad use Generative Adversarial Networks (GANs) [74] that perform Image-to-Image translation [236] to transform the appearance of any input data to match real-world examples of the desired phenomenon (e.g., transform a clear-weather image of a road into a snowy version). Image-to-Image translation does not require a formal understanding or model of the introduced environmental condition, but it does require an adequate collection of real-world examples (i.e., a sufficiently large collection of snowy images). Use of GANs in this manner gives developers less control over how the resulting synthetic data will appear, as it is entirely dependent on the example images given to the GAN and whatever features the GAN learns to focus on during its training phase. This chapter assumes a parameterized transformation procedure can be constructed to reproduce the environmental condition of interest (similar to DeepTest and DeepXplore).

3.5 Summary

This chapter has described a technique to both assess and improve the robustness of an LES to previously unseen environmental conditions. When applied to image-processing DNNs for two different vision tasks (image recognition and object detection), ENKI can introduce and discover unique contexts of an environmental condition that lead to more difficult test examples than random-generation or the DeepTest method. Likewise, this chapter has demonstrated how these contexts can

be used to augment training data to retrain a DNN to be more robust to the introduced environmental condition.

Two key benefits of the diversity-driven approach described by this chapter are that (1) multiple behavior criteria may be considered and (2) generated training/test cases are encouraged to elicit unique effects on the LES. Existing techniques that generate test cases by optimizing specific behavior metrics, such as neuron coverage, assume that the given behavior metric has some meaningful contribution to the failure of a DNN. Results from this study have shown that for the raindrop occlusion condition, neuron coverage has no correlation to failing test cases, and therefore, other behavior metrics should be considered. Since Enki can observe multiple criteria for diversification, it can uncover failing cases even when it is unknown how any specific metric will correlate to the failing cases. Furthermore, by generating cases that result in diverse behavior, Enki can create a wider spectrum of failing cases, uncovering corresponding operating contexts that lead to each form of failure. Existing greedy optimization methods may be used in conjunction to generate additional test cases that target each specific case uncovered by Enki.

Since developers are required to supply a simulation environment able to emulate the conditions of interest, Enki is not able to address conditions of uncertainty completely unknown to developers *a priori*. For example, Enki will not explore the effects of snowy weather without a simulation environment with a predefined function to emulate snow conditions. However, Enki can uncover unconsidered combinations of known conditions that result in a novel condition when combined. Additionally, as a tool for developers, the quality of data produced by this technique will only be as realistic as the simulation environment allows. In cases where synthetic data cannot be trusted for retraining purposes, contexts generated by Enki can still inform developers with which characteristics of the real-world phenomena should be targeted for follow-on study (e.g., raindrops of a certain size and in certain regions of the camera image).

As described, this technique is intended for use during design time to enable developers to assess and train a more robust LES to uncertain operational conditions. Chapter 4 extends the scope of this work to applications beyond DNNs, such as using EAs to evolve more robust configuration

settings for onboard controllers. Chapter 5 considers additional run-time processes to determine if the underlying learning components are robust enough to handle the present conditions. Chapter 6 describes how a self-adaptive LES may mitigate anticipated undesirable behavior and switch to alternative strategies depending on run-time assessments [39, 121, 181].

CHAPTER 4

ENABLING THE EVOLUTION OF ROBUST ROBOTIC SYSTEMS

This chapter investigates the integration of evolution-based optimization and novelty search in order to improve the robustness of autonomous systems. This chapter introduces the Evo-Enki [128] framework, which comprises Enki and Evo-ROS. Simon *et al.* [199] developed Evo-ROS to integrate EC with physics-based simulations of autonomous systems controlled by ROS [176]. Enki uses novelty search to discover operational scenarios[1] that lead to the most diverse behavior in the target system. Combining these techniques and tools into a single framework yields an automated approach to explore the operational landscape of the target system, identify regions of poor performance, and evolve system settings that better manage wide ranges of adversity. Experiments have been conducted with the throttle controller for EvoRally, a 1:5-scale autonomous vehicle for the study of autonomous driving systems. Preliminary results demonstrate the ability of the Evo-Enki framework to identify and characterize input speed signals that cause the existing controller to perform poorly. By identifying these problematic signals and autonomously switching among optimized controller modes, a control system can be developed to handle a wider range of conditions

The remainder of this chapter is organized as follows. Section 4.1 overviews the motivation and objectives of this chapter. Section 4.3 provides background information on simulation for cyber-physical systems. Section 4.3 describes the Evo-Enki framework. Section 4.4 presents an empirical evaluation of Evo-Enki applied to the EvoRally platform. Section 4.5 overviews related work in the topics covered by this chapter. Finally, Section 4.6 provides a concluding summary for this chapter.

[1] A scenario is defined as a specific set of parameter values that describe the environment and internal conditions of a system.

4.1 Overview

Autonomous cyber-physical systems are required for tasks where the burden of having a human operator is too high. With the absence of human supervision, effort must be made to mitigate sources of failure before the system is deployed or to include a capability for the system to adapt to unexpected conditions. This problem is particularly difficult due to the many sources of uncertainty in natural environments. Autonomous systems are described as being *robust* when they are capable of mitigating a wide range of adverse conditions. This chapter proposes an automated approach to assist software developers with making design decisions that result in more robust autonomous systems.

Developing autonomous systems to meet software requirements at run time is challenging, because subsystems must interact with uncertain conditions in both internal mechanisms and the external environment. Techniques are needed to identify key scenarios that will have the most significant impact on system performance at design time to help developers form strategies that can mitigate potential sources of failure. Existing techniques either optimize the system to perform on a manual selection of scenarios, randomly generate scenarios, or use some heuristic-driven approach to create scenarios. Manual selection often requires expert knowledge for the problem domain and can be subject to confirmation bias [27]. Random generation may not be useful for discovering "corner-case" scenarios that cover small regions of the operational landscape [38, 78]. Finally, iterative heuristic-driven approaches rely on objectives that may be difficult to define and can often lead to sub-optimal solutions when the operational landscape is not amenable to hill-climbing or gradient search [134].

This chapter presents the Evo-Enki framework, a two-phase evolution-based approach to improve the robustness of an autonomous system. Leveraging prior research with Evo-ROS [199], the first phase optimizes system settings for a given set of scenarios. The second phase uses Enki (introduced in Chapter 3) to identify sets of scenarios that produce both extreme and *diverse* (i.e., mutually unique) system behavior. Using these two phases in tandem, the proposed Evo-Enki framework can systematically improve autonomous systems with less reliance on *a priori* expertise

of the problem domain and less subject to bias that might mask harmful corner cases.

For an empirical evaluation, this chapter applies Evo-Enki to the EvoRally autonomous vehicle. Experiments have been conducted to improve the robustness of EvoRally's throttle controller to a wide (i.e., diverse) range of operating scenarios. Preliminary results demonstrate that new throttle controller settings derived from the Evo-Enki framework exhibit less overall error in the presence of an increasingly varied set of scenarios. Thus, Evo-Enki is demonstrated to improve the robustness of EvoRally to its operating environment.

4.2 Digital Twin-Based Simulation of a Cyber-Physical System

The term "digital twin" [76] has been adopted to refer to alternative, virtual versions of a physical platform under test. Virtual simulation can address many of the challenges associated with testing cyber-physical systems. Design changes on a physical platform or alterations to a physical test environment require a considerable amount of human effort and time. Furthermore, when testing faults occur, there is risk of damage to hardware and the external environment, which can be expensive to replace or repair. Digital twins enable engineers to determine modes of failure before a physical version of a system is ever produced, resulting in less expensive, more efficient, and less risky procedures for system evaluation.

System exploration with digital twins enables developers to identify unforeseen, emergent behavior during the design stage of a system [76]. Four categories of behavior have been identified within this context. Any desired, intentional behavior of the system is categorized as *Predicted Desirable* (PD). Problematic behaviors that are known *a priori* are categorized as *Predicted Undesirable* (PU). Unexpected but beneficial behaviors are categorized as *Unpredicted Desirable* (UD). Finally, any behavior that is unpredicted and leads to system failure is categorized as *Unpredicted Undesirable* (UU). The final category is of the most concern, because UU behaviors will have the most potential for catastrophic problems. The earlier UU behaviors are identified in a system's design, the less expensive they will be to address. Through the use of digital twins, exploratory techniques can uncover such unexpected emergent behaviors before a physical platform is deployed.

Both *partial knowledge* of the physical world and the *state explosion problem* [40] are challenges that should be considered when using digital twins for evaluation. Simulations must systematically model the environment and all external interfaces for the cyber-physical system (e.g., sensors and actuators). However, partial knowledge of the physical world can lead to potential "reality gaps [96]." Since it is not feasible to model all physical laws of the universe into a simulation environment, simulation results will be limited by the specific subset of physical phenomena implemented. Any relevant real-world phenomena missing from the simulation could threaten the validity of results derived from the evaluation of a digital twin. Furthermore, as features are added to the system interface and additional phenomena are simulated, the state explosion problem can become an issue [76]. To uncover unexpected emergent behavior, it is possible that thousands of parameters must be considered, leading to an explosion in the number of possible states to be evaluated. Though each state could be evaluated in parallel, the computational cost can be expensive. To address the state explosion problem, heuristic search procedures could be considered to navigate the state search space more efficiently. Concerning the reality gap, the use of digital twins should be considered as one approach of a larger framework for system evaluation that also includes follow-on real-world tests for validation.

4.3 Exploring Environmental Diversity with Novelty Search

The proposed Evo-Enki framework executes two steps to support the evolution of a more robust system configuration. Figure 4.1 illustrates the data flow between the two main computational steps for Evo-Enki, where data flow is indicated by labeled arrows, steps are depicted by circles, external entities are depicted by boxes, and data stores are depicted by double lines. Step 1 enables the evolution of a system configuration that works optimally across a range of given operational scenarios. Step 2 enables the evolution of a new set of operational scenarios that exercise the resulting configuration in *diverse* ways. The resulting diverse scenarios can then enable an even more robust system configuration to be evolved through future executions of Steps 1 and 2. Next, each step is described in turn.

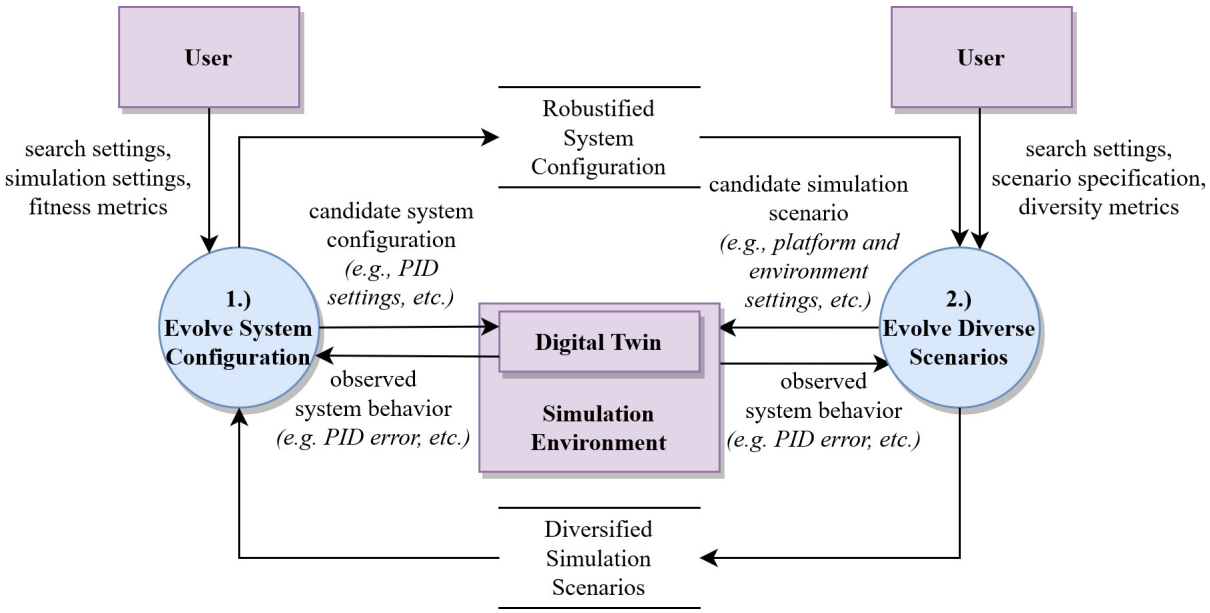


Figure 4.1: High-level DFD of the Evo-Enki framework. Circles represent computational steps, boxes represent external entities, parallel lines mark persistent data stores, and connecting arrows show the flow of data between steps.

4.3.1 Fitness-Driven Optimization of System Settings

The first step of the Evo-Enki framework (isolated in Figure 4.2) is responsible for evolving a system configuration that performs optimally over a given set of operating scenarios. A standard fitness-guided EA (Step 1.a) evolves a population of candidate configurations of the target platform (e.g., settings for a PID controller) towards an optimal configuration for a given set of scenarios. Candidate configurations are encoded into a vector form (i.e., a genome), and through standard operations of evolution, the genomes are recombined and mutated to uncover new candidates. Each candidate is evaluated by using Evo-ROS (Step 2.b) to spawn an instance of the simulation environment. Evo-ROS was developed by Simon *et al.* to enable an EA to use a ROS simulation environment when evaluating each candidate for evolution [199]. The simulation environment is initialized to run the given scenarios, and the simulated platform's state is monitored to determine the fitness of the given system configuration. The final output is the candidate configuration with the highest fitness score after multiple generations of evolution.

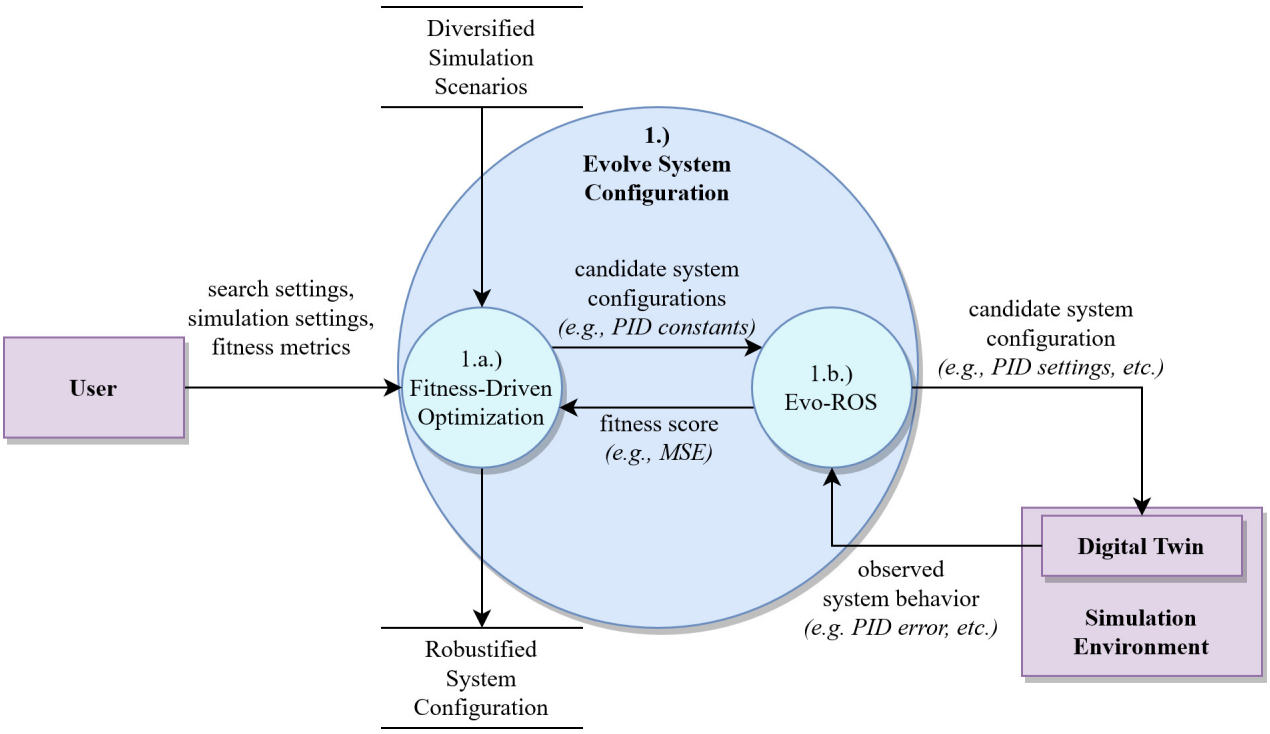


Figure 4.2: Data flow between the steps responsible for optimizing a robust system configuration for a diverse set of operating scenarios.

4.3.2 Diversity-Driven Evolution of Simulation Settings

The second step of the Evo-Enki framework (isolated in Figure 4.3) is responsible for evolving a new set of operating scenarios that exercise the given system configuration in mutually unique ways. Enki (Step 2.a) assesses the operation of a target software system to discover operating contexts that lead to unique, extreme, and possibly unexpected behavior. Enki uses an evolution-based novelty search algorithm to manage populations of individual scenarios. Each scenario is represented by a genome that encodes its operational characteristics and is associated with a phenome that encodes the target system's behavior when exposed to the scenario. Enki evolves this population for multiple generations, using standard operations of evolution (i.e., recombination, mutation, and selection). However, unlike traditional evolutionary search methods, Enki does not guide the population towards a single fitness objective. Instead, a novelty archive is maintained across generations that ranks all individual scenarios in the current population with a novelty score and only archives the scenarios that exhibit the most diverse behavior from the target system.

The novelty score for an individual scenario is determined by comparing its phenome to its nearest neighbors in the archive and averaging the distance. Thus, through evolution, ENKI guides the search for scenarios outward in the operational landscape in terms of the type of behavior they produce from the target system. A benefit of this approach versus traditional evolutionary search is that the output is a set of individual scenarios that produce unique results, which may be adverse or favorable for the target system across multiple metrics, instead of producing only scenarios that maximize a single objective metric.

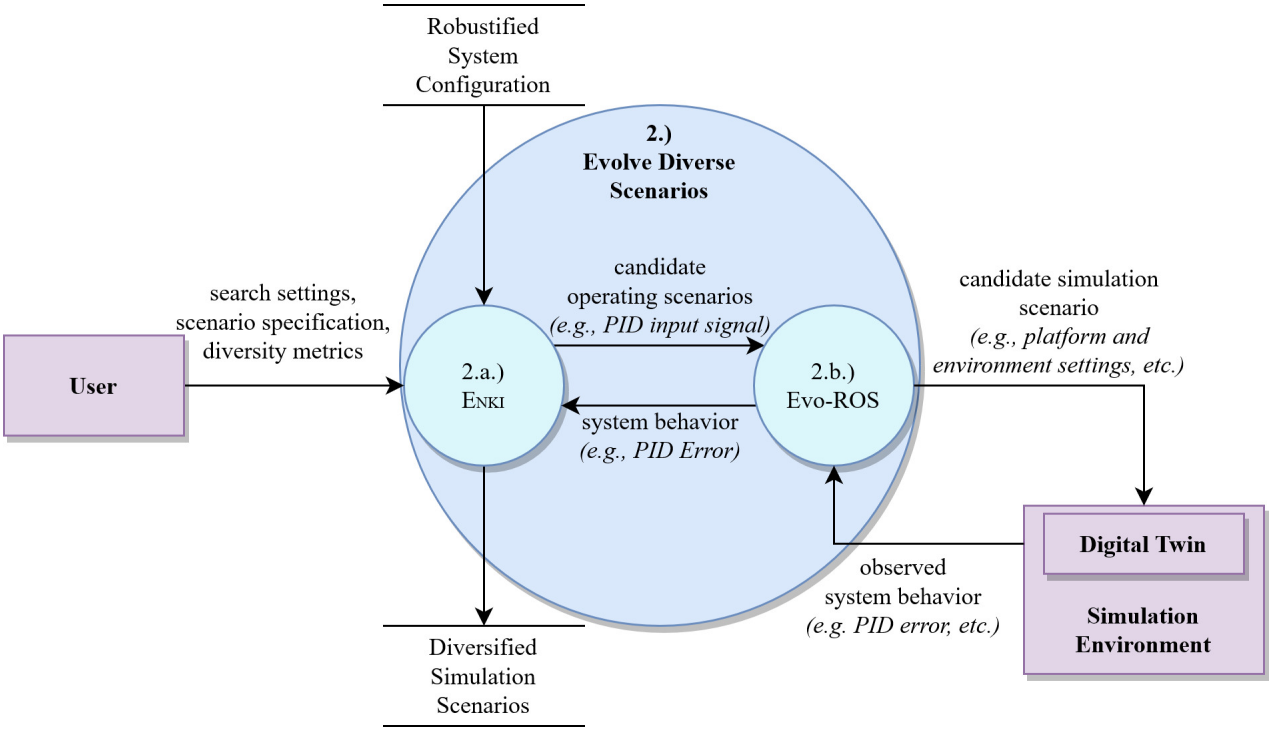
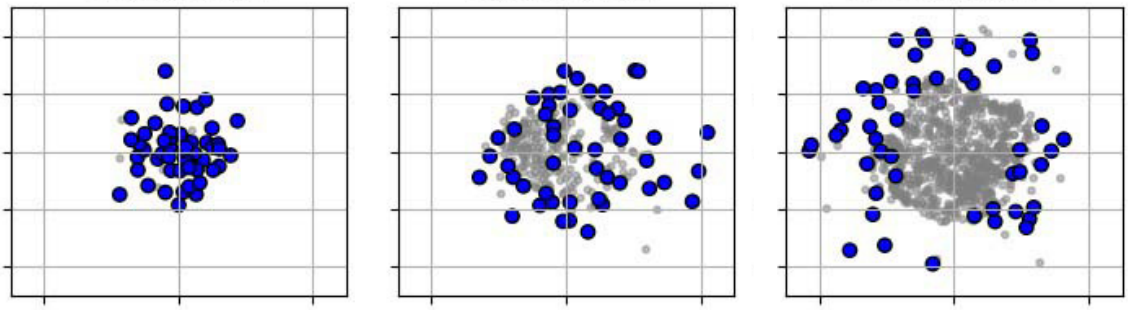


Figure 4.3: Data flow between the steps responsible for generating diverse operating scenarios for a given system configuration

Figure 4.4 illustrates the diversifying effect of ENKI's novelty search process on the archived collection of scenarios. Each point corresponds to an individual scenario. Blue points correspond to scenarios currently in the archive, and gray points correspond to those that have been evaluated but not archived. All points are projected into two dimensions with distances scaled to match the relative distance between the scenarios' phenomes. In early generations, where the scenarios are closer to a random selection, the archived scenarios produce very similar behavior from the target

system. As the search progresses, the archive "pushes" outward, demonstrating that each archived scenario is increasingly affecting the target system in different ways (i.e., becoming more diverse).

Visualization of Phenotype Distances in ENKI's Archive



(a) Generation 1 (b) Generation 5 (c) Generation 15

Figure 4.4: A visualization of phenotype distances between scenarios explored by ENKI. Blue points show archived scenarios. Gray points show all other evaluated scenarios. Archived scenarios increase in diversity over generations.

4.4 Empirical Validation

This section describes an experiment to evaluate the EVO-ENKI framework on the EVORALLY platform. The overall objective is to improve the ability of EVORALLY's throttle controller to handle a wider range of reference speed signals. EVORALLY uses a PID throttle controller that is calibrated by adjusting four tuning constants, K_P, K_I, K_D and I_{max}. This section designates default values for these tuning constants with the label C_0. For this experiment, EVO-ENKI first evolved new values for the PID tuning constants (C_1) based on a single reference speed signal (e.g., Step 1 Figure 4.1). EVO-ENKI then evolved a new collection of speed signals that result in a diverse range of behavior in the throttle controller (e.g., Step 2 in Figure 4.1). New PID constant values (C_2) were then evolved yet again but, in contrast to C_1, these were evolved to work optimally across all ENKI-generated scenarios. The resulting values for C_0, C_1, and C_2 are provided in Table 4.1. Through this experiment, we aim to answer the following research questions:

RQ1.) Can EVO-ENKI evolve tuning constants for the controller (C_1) that are more robust than the default constants (C_0)?

RQ2.) Can Enki discover more challenging test speed signals to identify weaknesses in the controller, when compared to a random-generation technique?

RQ3.) Can Evo-Enki leverage Enki-generated scenarios to evolve even more robust tuning constants for the controller (C_2), when compared to tuning constants evolved based on a single scenario (C_1)?

4.4.1 The EvoRally Throttle Controller

This section includes experiments conducted with a Proportional-Integral-Derivative (PID) throttle controller [13]. In general, controllers are responsible for monitoring and adjusting a system's state to ensure the system behaves correctly. More specifically, controllers monitor *process variables* and take corrective actions on *control variables* to ensure the system's output remains within expected limits. PID controllers are given a *reference signal* as the target value for a process variable, and an *error signal* is computed as the difference between the target value and the actual value from the process variable. In the case of a PID throttle controller, the process variable is the actual speed of the vehicle, the reference signal is the target speed, and the error is the difference between the target speed and actual speed. PID controllers attempt to minimize error by adjusting the control variable via three control terms (P, I, and D). In order for the controller to respond correctly, certain *tuning constants* associated with each of these terms (K_P, K_I, and K_D, respectively) must be tuned for the application. Improper values for these constants can lead to problems such as oscillation and overshoot.

Table 4.1: PID Controller Settings.

	K_P	K_I	K_D	I_{max}
C_0 (Default)	0.200	0.000	0.001	0.150
C_1 (Evolved)	0.203	0.045	0.092	0.511
C_2 (Evolved with Enki)	0.679	0.658	0.772	0.445

4.4.2 Evolving a New Controller

For this experiment, we used Step 1 of Evo-Enki to evolve a controller configuration (C_1) that improves upon the default configuration (C_0). Improvement is measured by the reduction in controller error when exposing EvoRally to a single reference signal (see Figures 4.5a and 4.5b). The configuration settings for the fitness-driven EA (Step 1.a) are provided in Table 4.2. Upon completion, the fitness-driven EA reduced the mean-squared-error (MSE) between the actual speed of the vehicle and reference speed for the controller from 0.528 for C_0 to 0.018 for C_1. However, further assessment is required to show that C_1 also exhibits improvement over a wider range of reference signals. Figure 4.5 compares the performance of C_0 and C_1 on three other reference signals. Notably, C_1 was not optimized for these signals. In Figures 4.5c through 4.5f, C_1 tracks acceleration for the reference signal better than C_0, but deceleration remains a challenge without the use of brakes. When braking is allowed, C_1 is able to track both the acceleration and the deceleration better than C_0 (Figures 4.5g and 4.5h, respectively).

4.4.3 Assessing the Controllers

To further assess the effectiveness of C_0, test reference signals were generated via Step 2 of Evo-Enki. Enki generated $1,250$ test reference signals (S_{ENKI}), and for comparison, an additional $1,250$ signals were created via a random-generation method (S_{RAND}). All signals were generated to cover a period of 60 seconds with speeds capped at a maximum of 10 meters-per-second (m/s). The MSE was computed for the performance of C_0 on each generated reference signal.

A probability distribution of MSE for signals generated from each method is displayed in Figure 4.6. Regions are shaded by quartile ranges, with green being the bottom quartile, blue being the middle two quartiles, and red being the top quartile. Signals from S_{RAND} showed an average MSE of 1.047 (Figure 4.6b) compared to an average MSE of 2.430 from signals in S_{ENKI} (Figure 4.6a). Compared to random-generation, signals from Enki are more likely to result in a lower MSE, and therefore, Enki is more likely to find reference signals that cause the system to perform poorly. The random-generation method creates reference signals by uniformly selecting values for each signal.

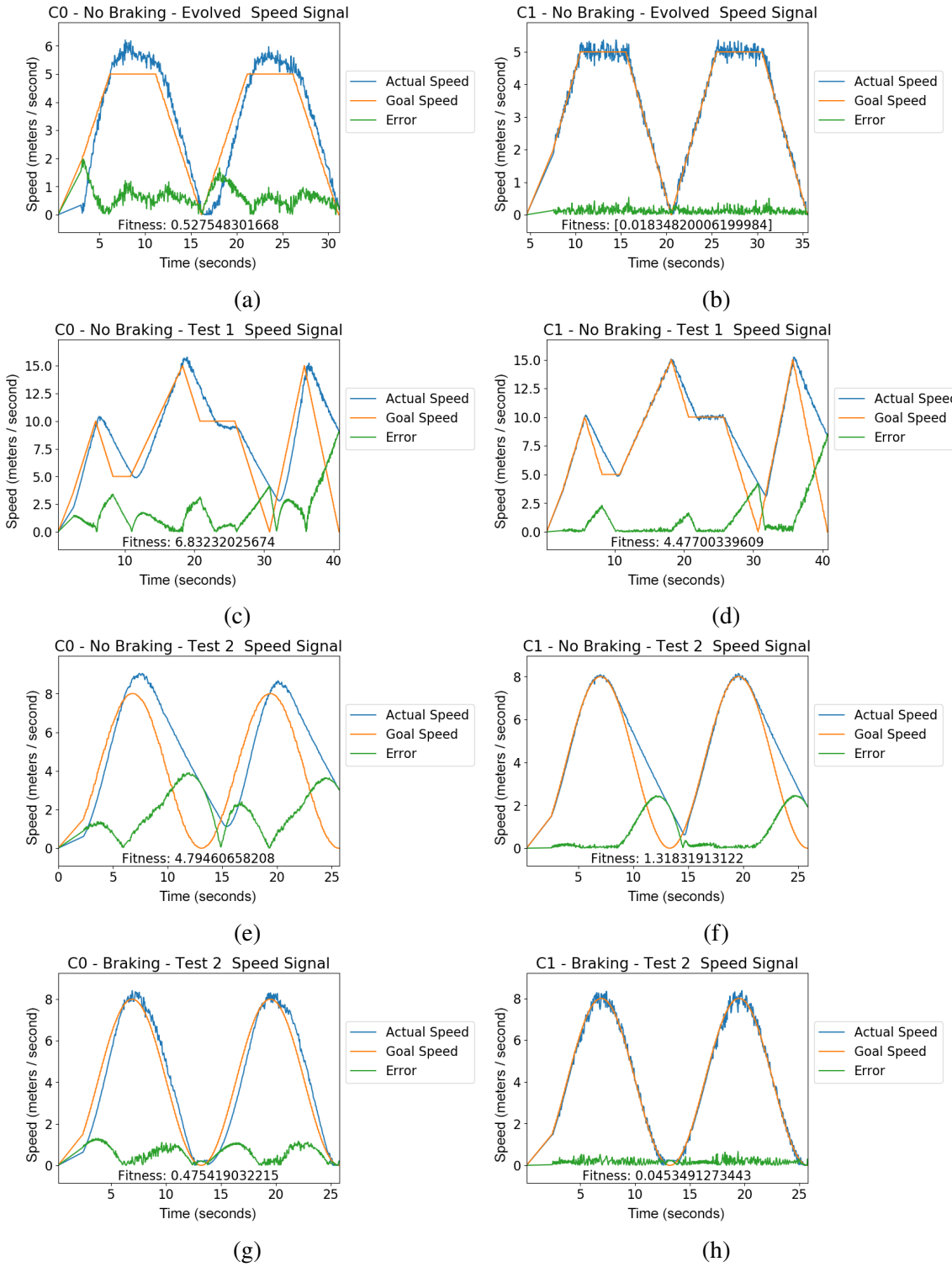


Figure 4.5: Comparison of the default C_0 PID controller (left) settings with the evolved C_1 PID controller (right) settings over a variety of different speed signals. Subplots (a) and (b) show the speed signal against which the C_1 was evolved. Subplots (c), (d), (e), and (f) show performance of C_0 and C_1 against other random test signals, and subplots (g) and (h) show performance of C_0 and C_1 on a test signal while braking is allowed.

When error-inducing reference signals occupy a small region of the domain of all possible reference signals, a uniform selection method is not likely to uncover such challenging signals. Instead, a random-generation method will more likely result in reference signals that exhibit similar degrees of error. In contrast, ENKI seeks out reference signals that produce unique differences in error, and therefore, ENKI can produce reference signals with a more uniform distribution of MSE, which may then lead to sets of more diverse and challenging reference signals. Because the high-level objective is to evolve a more robust controller, the purpose of ENKI is to discover less common and more adverse reference signals.

Table 4.2: Fitness EA Configuration

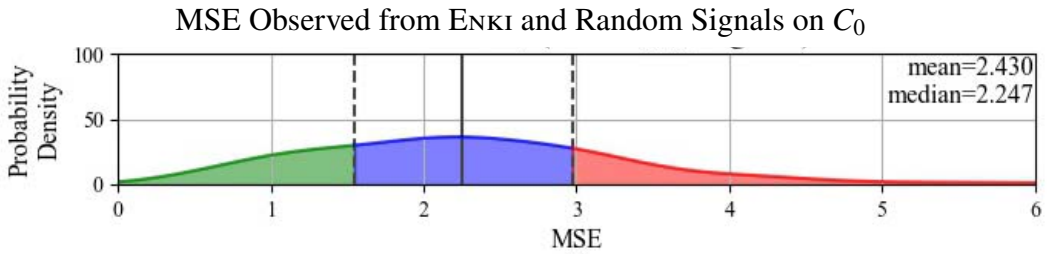
Parameter	Setting
Generations	25
Population Size	25
Tournament Selection	
- Size:	2
Two-Point Crossover	
- Rate:	50%
Gaussian Mutation	
- Sigma:	1.0
- Rate:	20%
Objective	*Minimize MSE*
Run Time	~12 *hours*

Table 4.3: ENKI EA Configuration

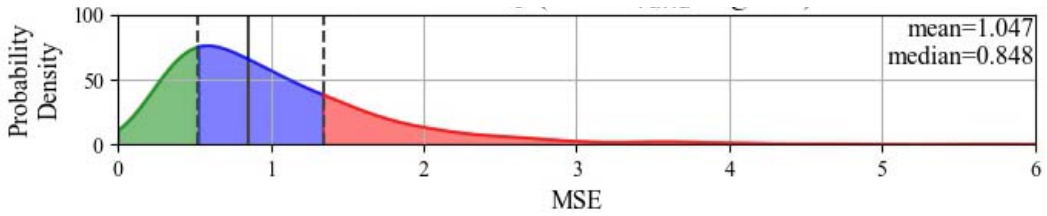
Parameter	Setting
Generations	50
Population Size	50
Tournament Selection	
- Size:	3
Single-Point Crossover	
- Rate:	100%
Creep Mutation	
- Range:	±20%
- Rate:	25%
Objective	*Diversify Error*
Run Time	~5 *hours*

To compare the effectiveness of C_1 to C_0 on a wider range of possible reference signals, we used ENKI to generate reference signals based on the performance of C_0 (i.e., the error produced by using C_0 for each generated reference signal). Again, all reference signals were generated for 60 seconds and capped at 10 m/s. The error was defined as the absolute difference between the actual speed of the vehicle and the reference signal. After running ENKI with the settings provided in Table 4.3, we generated $2,489$ signals (S_0), and Figure 4.7 shows a comparison of the results from each controller. We found that the average MSE for C_0 was 2.672 (Figure 4.7a). When exposing C_1 to the same test reference signals, we found that the average MSE for C_1 was 2.250 (Figure 4.7b). Each subplot shows a distribution of MSE observed by ENKI for each controller. Since the whole distribution can be seen to skew right for C_1 when compared to C_0, C_1 has been

observed to produce less error in general for the given reference signals. Therefore, assessment with ENKI further supports the claim that EVO-ENKI did successfully evolve more robust controller settings for C_1 than C_0.



(a) Distribution of MSE for C_0 for signals in S_{ENKI}

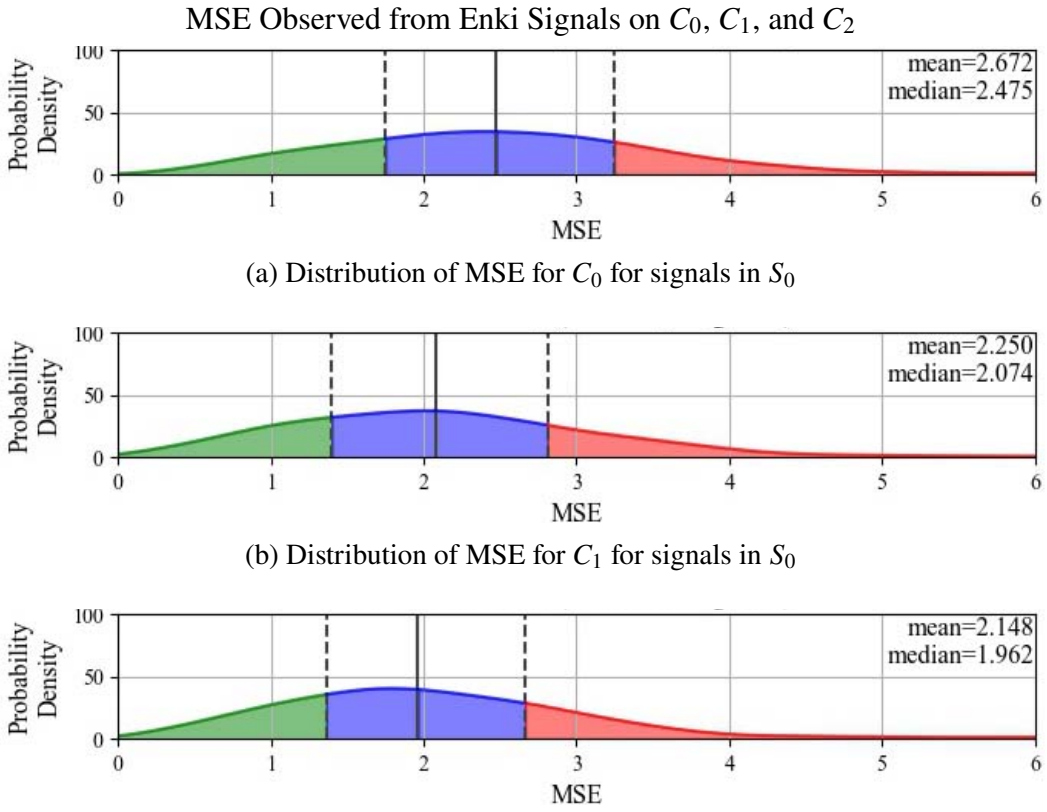


(b) Distribution of MSE for C_0 for signals in S_{RAND}

Figure 4.6: MSE observed from C_0 for signals from S_{ENKI} and S_{RAND}. Regions are shaded by quartile ranges, with green being the bottom quartile, blue being the middle two quartiles, and red being the top quartile.

4.4.4 Further Enhancing the Controller

To explore whether use of ENKI-generated scenarios as the basis of evolution can further improve the robustness of the controller, a second set of controller settings (C_2) was evolved. C_2 was created in a process similar to C_1 (see Section 4.4.2). However, for C_2, instead of only evolving the controller against the single reference signal shown in Figure 4.5a and Figure 4.5b, the controller was also evolved against the top 5 most unique reference signals generated by ENKI. After deriving the values for C_2, we evaluated the new settings against the signals in S_0, and the results are displayed in Figure 4.7c. We found that C_2 further reduced the average MSE to 2.148, with the overall MSE distribution skewed slightly more right. The reduced error exhibited by C_2 shows that by assessing the controller with ENKI, we can find scenarios to further harden the controller to a wider range of reference signals.



(a) Distribution of MSE for C_0 for signals in S_0

(b) Distribution of MSE for C_1 for signals in S_0

(c) Distribution of MSE for C_2 for signals in S_0

Figure 4.7: Error observed on controller settings C_0, C_1, and C_2 when tested against signals from S_0. Regions are shaded by quartile ranges, with green being the bottom quartile, blue being the middle two quartiles, and red being the top quartile.

4.4.5 Threats to Validity

Since this chapter's framework relies on simulation, it is assumed that the simulator is capable of accurately matching reality. Any deviation in the simulator from the physical vehicle and its environment will impact the effectiveness of this approach on a real platform. Future work includes validating the results with the physical EvoRally platform. Additionally, this experiment has only considered reference signals with a maximum speed of 10 m/s, and therefore, the comparative performance of the controllers may not be valid when the speed is allowed to exceed 10 m/s. Finally, the intent of this chapter has been to assess the potential value of the Evo-Enki framework as a proof-of-concept, but since evolution-based techniques contain stochastic elements, multiple trials will be required to determine the statistical relevance of these results.

4.5 Related Work

The work described in this chapter extends research conducted by Clark *et al.* [39] on discovering execution mode boundaries for adaptive controllers on robotic fish. Their work describes a mode discovery algorithm that evolves the controller over a fixed set of scenarios and then incrementally adds new scenarios to the set for each round of evolution. There work considers two different adaptive random techniques for scenario generation, where scenarios are randomly generated from a base scenario and selected based on specific criteria. In contrast to their work, the Evo-Enki framework uses novelty search to generate new scenarios without any need for a pre-defined "base" scenario.

4.6 Summary

This chapter has demonstrated that evolution-based techniques can enable the discovery of more robust configurations for an autonomous system with limited input by the user. As demonstrated with EvoRally's throttle controller, Evo-Enki can evolve a more robust system configuration by first evolving a diverse set of operating scenarios and then optimizing a new system configuration to minimize error across all diverse scenarios.

The ultimate goal is to apply this work to a physical system that can independently and effectively detect changing adverse environmental conditions, such as sharp inclines or slippery terrain, and safely transition into a better suited system configuration or mode to navigate efficiently through the adverse environment. Additional investigations address these issues. Chapter 5 introduces machine learning techniques that additionally learn to infer execution mode boundaries for a given system configuration. Chapter 6 proposes how to construct an adaptive system that can switch between predetermined system configurations when assurance cases are no longer satisfied at run time. Chapter 7 proposes how to adapt systems to maintain system requirements at run time. Finally, Chapter 8 proposes how to construct a service-oriented framework for managing the trustworthiness of LECs in an autonomous system at run time, when faced with environmental uncertainty.

CHAPTER 5

UNCOVERING INADEQUATE LEARNING MODELS AT RUN TIME

Since deep learning systems do not generalize well when training data is incomplete and missing coverage of corner cases, it is difficult to ensure the robustness of safety-critical self-adaptive systems with deep learning components. Nonetheless, stakeholders require a reasonable level of confidence that a safety-critical system will behave as expected in all contexts. However, uncertainty in the behavior of safety-critical LESs arises when run-time contexts deviate from training and validation data. To this end, this chapter proposes an approach to develop a more robust safety-critical LES by predicting its learned behavior when exposed to uncertainty and thereby enabling mitigating countermeasures for predicted failures [127]. By combining evolutionary computation with machine learning, an automated method is introduced to assess and predict the behavior of an LES when faced with previously unseen environmental conditions. By experimenting with DNNs under a variety of adverse environmental changes, the proposed method is compared to a Monte Carlo (i.e., random sampling) method. Results indicate that when Monte Carlo sampling fails to capture uncommon system behavior, the proposed method is better at training behavior models with fewer training examples required.

The remainder of this chapter is organized as follows. Section 5.1 overviews the motivation and objectives of this chapter. Section 5.2 describes the proposed method for building a more resilient LES. Section 5.3 presents results from an empirical evaluation. Section 5.4 reviews related work. Finally, Section 5.5 provides a concluding summary for this chapter.

5.1 Overview

As artificial intelligent software systems migrate from the purview of research and development into commercial applications with widespread use, trust in the ability for such systems to operate as intended and safely has become paramount. When deployed in the real world, autonomous systems will likely encounter adverse conditions not considered during their design, such as zero-

day security attacks or system degradation from inclement weather. This problem is exacerbated when the system being tested is also self-adaptive or relies on a learning component. As system behavior adapts, new test cases may need to be considered and existing test cases may become obsolete [62, 63]. Furthermore, when a run-time context diverges from training experience for a LES, it can be difficult to establish confidence in the system's behavior. This chapter proposes an automated approach to synergistically combine evolutionary computation with machine learning to model and predict system behavior for such uncertain contexts (i.e., *known unknowns*). This approach can be used by autonomous systems to predict and mitigate failure of a learning component at run time by deploying fail-safes for situations in which the component has not been adequately trained. By leveraging such context-aware self-assessment, use of the proposed techniques can help instill confidence that an autonomous system will recognize and use learning components in contexts for which it can handle.

It is difficult to determine the limitations of safety-critical systems that rely on DNNs [73], especially for situations that deviate from training experience [20]. In real-world environments, adverse corner cases (e.g., the effect of a scratched camera lens or a lens flare caused by the setting sun) are frequently missing when establishing training and validation examples for an LES. Even for known adverse environmental factors included in data, confirmation and selection bias [27] can negatively influence learned behavior. GANs [74] have been used to synthetically augment datasets to fill in gaps (e.g., image-to-image translation [236]), however the quality of such GANs is highly contingent on existing examples of the adverse phenomena. When examples of the adverse phenomena are lacking, other solution strategies are necessary to increase training/validation coverage for safety-critical LESs.

This chapter introduces the ENLIL[1] framework to produce a *behavior oracle*, which is used to predict how an LES will respond to operating conditions not covered by existing datasets [127]. ENLIL combines evolutionary computation with deep learning to support the automatic creation of a wide range of test cases for an LES, to assess how robust an LES is to specific forms of

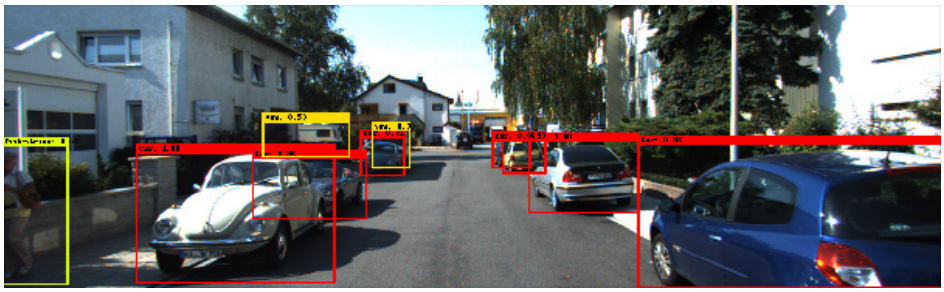
[1]Enlil is an ancient Sumerian deity that possesses the "Tablet of Destinies [118]."

sensory noise, and to model system behavior under uncertainty. ENLIL synthesizes examples of new operating contexts with newly-introduced sensory phenomena by transforming existing data. ENLIL then generates diverse synthetic examples that correspond to predefined behavior categories, which are then used to train a predictive behavior model via machine learning [9]. The resulting behavior model can then be used to predict how the target LES will respond to sensory data with the newly-introduced phenomena. Machine learning plays two roles in this work. First, the LES uses machine learning to accomplish its system objectives (e.g., detect obstacles). Second, ENLIL uses machine learning to create a behavior oracle for predicting the behavior of an LES when facing unexpected conditions. Ultimately, this work aims to enable the construction of a more robust safety-critical LES through automated methods that assess and predict how the LES will behave under previously untested conditions. ENLIL's behavior oracles can enable a safety-critical system to preemptively determine when learning components are inadequate for a given run-time context, thereby prompting transitions/adaptations to mitigate faults (i.e., fail-safes).

Experiments have been conducted to evaluate use of the ENLIL framework on an object detector for an autonomous vehicle under inclement environmental conditions (e.g., poor lighting, rain, and fog). Specifically, ENLIL is used to predict the object detector's performance in the presence of the newly-introduced environmental conditions. Figure 5.1 provides an illustrative use case. Figure 5.1b shows how a trained object detector can correctly identify all vehicles in a clear image (outlined in boxes). Figure 5.1c shows that when a synthetic raindrop is introduced onto the camera lens, the object detector fails to detect the occluded vehicles. Finally, Figure 5.1d shows how a behavior model trained by ENLIL can perceive the raindrop and predict that it will disrupt object detection (marked with red shading). This chapter assumes that these environmental conditions have not been sufficiently covered by default training/test data, and therefore, it is unknown how they will impact the object detector's performance *a priori*. For validation, example training cases created by ENLIL have been compared to those generated by a baseline Monte Carlo random sampling method, and the accuracy of ENLIL's predictive models have been evaluated against models trained only on Monte Carlo training cases.

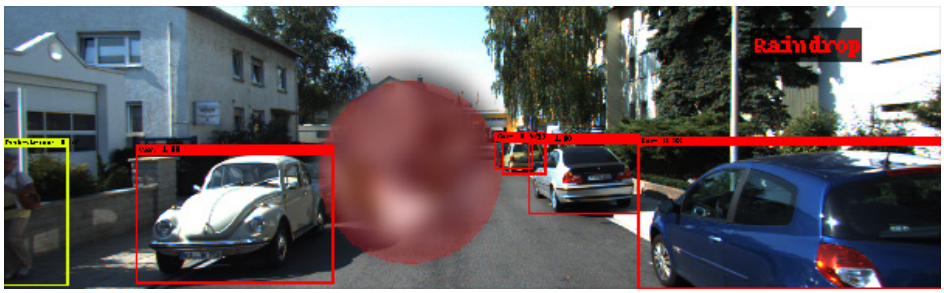
Demonstration of ENLIL Prediction

(a) Unaltered View of Street



(b) Automobiles Detected

(c) View Obscured by Raindrop



(d) Detection of Raindrop and Prediction of Failure

Figure 5.1: Example of an ENLIL behavior model use-case. An object detector is given a clear image (a) and identifies all automobiles (b). A raindrop is introduced into the scene, which causes the system to fail to detect occluded automobiles (c). The ENLIL model is able to correctly detect the interfering raindrop (d) and predicts that the system will fail.

Results show that a significant percentage of system failures can be predicted preemptively with the ENLIL framework, which can enable developers to better implement mitigation strategies. Furthermore, results demonstrate that ENLIL generates balanced sets of training cases with respect to the types of system behavior induced. As such, ENLIL can train comparably accurate or better predictive models with fewer training examples for the considered object detector.

5.2 Predictive Behavior Modeling for System Resilience

This section describes the ENLIL framework, an automated method to create a more robust LES through predictive behavior modeling. First, ENLIL automatically assesses LES behavior under synthetic environmental phenomena to generate a collection of operating contexts that lead to specific *behavior categories* (i.e., types of resulting behavior). Next, ENLIL uses the generated contexts to train a predictive behavior model of the LES (i.e., a behavior oracle [9]) that can be used either internally or by an external system (e.g., an autonomic manager [105]) to determine how the LES will behave when faced with any given context of the environmental phenomena.

Figure 5.2 provides a DFD for the ENLIL framework, where circles represent computational steps, boxes represent external entities, arrows represent data flow, and data stores are represented within parallel lines. ENLIL has two primary objectives: to generate a diverse variety of operating contexts for the LES (Step 1 and Step 2) and to infer a behavior model of the LES based on the generated contexts (Step 3).

For example, Figure 5.3 demonstrates how rainfall can impact an object detector for an autonomous vehicle to the point of failure. The object detector can successfully identify an automobile and a cyclist in the absence of rainfall (Figure 5.3a). Light synthetic rainfall is introduced with no impact on detection (Figure 5.3b). However, more intense rainfall can result in a failure to detect the cyclist (Figure 5.3c). When default training data does not include any such examples of rainfall, the ENLIL framework generates synthetic examples and trains a behavior model to predict the object detector's performance under rainfall (i.e., if it is likely to miss an object).

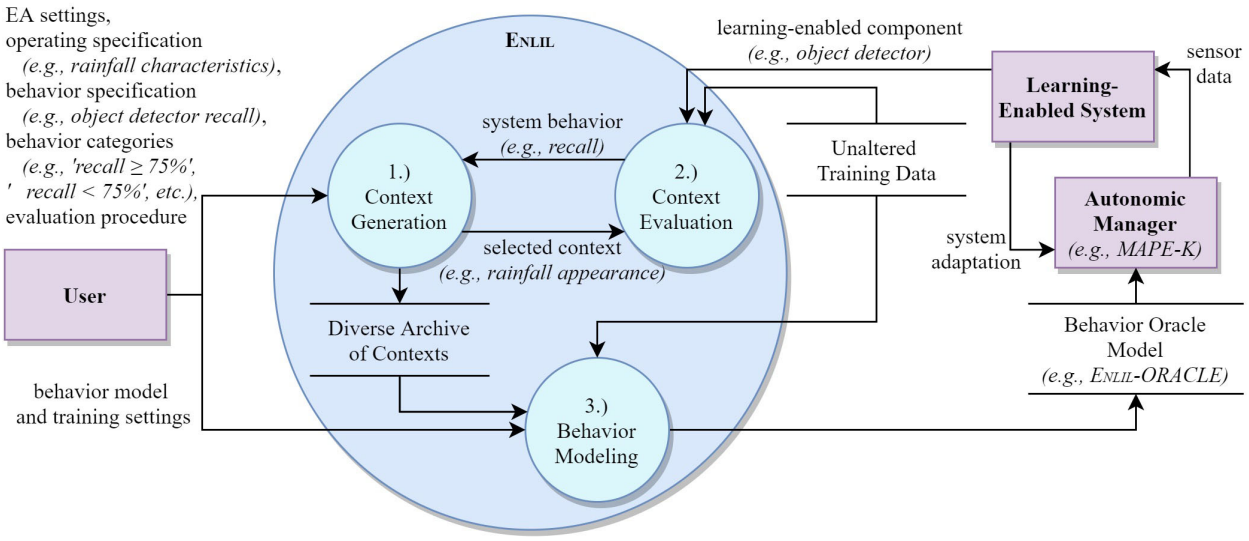


Figure 5.2: High-level DFD of the ENLIL framework. Computational steps are shown as circles. Boxes represent external entities. Directed lines indicate data flow. Persistent data stores are marked within parallel lines.

Step 1. Context Generation.

To assess the behavior of an LES under a previously unseen environmental condition, data must be collected to determine how the introduced condition impacts the behavior of the LES. To accomplish this task, the ENLIL framework requires an *operating specification*, *behavior specification*, *behavior categories*, and *evaluation procedure* from the user.

Operating specification The operating specification defines each configurable parameter of the LES's operating environment (e.g., rainfall angle, intensity, etc.). A label and permissible value ranges must be specified for each parameter, where values can be numeric or categorical. Each candidate operating context generated by ENLIL is sampled from the values defined in this specification. For the example rainfall phenomena (Figure 5.3), an operating specification (OP) could be defined as follows:

$$\text{OP} = \begin{bmatrix} rainfall\_angle & \in & [\frac{1}{4}\pi, \frac{3}{4}\pi] \\ rainfall\_density & \in & [0, 1] \\ rainfall\_intensity & \in & [0, 1] \\ rainfall\_seed & \in & [0, 100] \end{bmatrix}$$

The Effect of Rainfall on an Object Detection DNN

(a) Unaltered Image from KITTI dataset [70]

(b) Light Rainfall (Simulated)

(c) Medium Rainfall (Simulated)

(d) Heavy Rainfall (Simulated)

Figure 5.3: Demonstrative examples for how rainfall can affect an image-based DNN for object detection. Detected objects have been marked with overlaying bounding boxes, as shown in (a). As "rainfall" increases in intensity, as seen in (b) and (c), the effectiveness of the DNN diminishes to a point where it fails to detect the cyclist, as seen in (d).

Behavior specification The behavior specification defines a set of observable behavior metrics for the LES. For an object detector, this may include properties such as *precision* and *recall* for a set of validation images, specified as follows.

$$\text{BV} = \begin{bmatrix} precision & \in & [0,1] \\ recall & \in & [0,1] \end{bmatrix}$$

In addition to these component-level properties, system-level properties may also be considered when *utility functions* [36] exist to evaluate functional objectives at run time.

Behavior categories The Enlil framework assumes that a set of behavior categories has been established by the user, which are defined in relation to the behavior metrics in the behavior specification. For example, behavior categories could be specified as "expected object misdetection" (*Cat* 1), "expected pedestrian misdetection" (*Cat* 2), and "expected successful detection" (*Cat* 3), where each category is based on thresholds (θ) for the recall (r) according to the following user-specified mapping.

$$\text{BV-CAT}(r) = \begin{cases} \textit{Cat } 1, & \text{if } r_{total} < \theta_{total} \\ \textit{Cat } 2, & \text{if } r_{total} \geq \theta_{total} \wedge r_{ped} < \theta_{ped} \\ \textit{Cat } 3, & \text{if } r_{total} \geq \theta_{total} \wedge r_{ped} \geq \theta_{ped} \end{cases}$$

Evaluation procedure The evaluation procedure specifies how Enlil must instantiate the simulation environment and monitor the LES's behavior for each given operating context (see Step 2).

To efficiently search the space of all possible operating contexts and uncover a wide range of examples for each specified behavior category, the Enlil framework uses an EA search procedure. The EA generates candidate operating contexts, assesses the behavior of an LES under each candidate context, and archives the most diverse contexts found to match the given behavior category. The result is an archive of operating contexts with widely varying appearance but similar resulting LES behavior to match each respective behavior category. The intent is to apply this technique for situations where the full impact of an operating condition is not known for the LES,

and it is not possible to exhaustively assess every explicit context. When considering how rainfall might impact a camera-based object detector for an autonomous vehicle, it is not practical to evaluate every potential appearance of rainfall. Given a parameterized simulation procedure to emulate rainfall, ENLIL evolves a population of multiple rainfall contexts in parallel and encourages each generation to exhibit increasingly different appearances of rainfall (e.g., *angle* and *intensity*) that result in the same behavior category for the LES (e.g., "expected pedestrian misdetection").

ENLIL's EA for context generation is described in Algorithm 5.1. Each candidate context is represented by a *genome* and *phenome*. Genomes encode the operating characteristics of each context (e.g., rainfall angle, intensity, etc.), where the number of specified operating parameters will determine the length of each genome. Phenomes encode both the operating characteristics and resulting system behavior for each context (e.g., angle and intensity values for rainfall and resulting object detector accuracy). An initial *population* of operating contexts is generated with random operating characteristics. Operating characteristics for each context in the population are manipulated through the recombination and mutation of selected genomes. After modifying genomes, a population's phenomes are determined by evaluating the behavior of the LES for each individual context. To encourage diversity, the population is compared to an archive that tracks phenomes that are mutually unique and fit the given behavior category.

ENLIL ranks each context in the current population based on how its phenomes compare to the archived contexts. Context diversity is determined by the Euclidean distance between an individual's phenome and the phenomes of its nearest neighbors as follows,

$$\text{nov}(\mathbf{p}, \mathbf{P}, k) = \text{mean}\Big(\text{min-k}(\|\mathbf{p} - \mathbf{p}_i\|^2 \; \forall \mathbf{p}_i \in \mathbf{P} : \mathbf{p}_i \neq \mathbf{p})\Big)$$

where the *novelty score* (*nov*) is computed as the average distance between a phenome ($\mathbf{p}$) with its k nearest neighbors in the set of all archived phenomes ($\mathbf{P}$). Contexts with phenomes that are more distant from the closest matching phenomes in the archive (i.e., less similar) are given priority. Since ENLIL aims to generate contexts with diverse appearance but similar resulting system behavior, an additional step is included to filter out any candidate contexts from the population that do not match the specified behavior category (Line 9 in Algorithm 5.1). After ranking each context in a

population, any context that results in behavior other than the target behavior category is excluded from the population, such that it will not contribute to future generations. Thus, as ENLIL evolves the population over consecutive generations, individual contexts within the population will be favored for survival and reproduction when they exhibit operating characteristics that are different from those found in the archive yet produce similar behavior in the LES (e.g., "expected pedestrian misdetection"). In essence, ENLIL attempts to generate the broadest set of example contexts that yield the same resulting system behavior for a given behavior category.

Algorithm 5.1 ENLIL: Context Generation (Step 1)

```
1:  function EVOLUTIONARY-SEARCH(n_generations, behavior_pattern, eval_func)
2:      archive ← ∅
3:      pop ← RANDOM-POPULATION()
4:      for 0 to n_generations do
5:          pop ← SELECTION(pop)
6:          pop ← RECOMBINATION(pop)
7:          pop ← MUTATION(pop)
8:          pop ← EVALUATION(pop, eval_func)
9:          pop ← FILTER(pop, behavior_pattern)
10:         archive, pop ← COMMIT-TO-ARCHIVE(archive, pop)
11:     return archive
12: function COMMIT-TO-ARCHIVE(archive, pop)
13:     scores ← RANK-INDIVIDUALS(archive, pop)
14:     pop ← pop : scores > novelty_threshold
15:     archive ← TRUNCATE(archive ∪ pop)
16:     return archive, pop
```

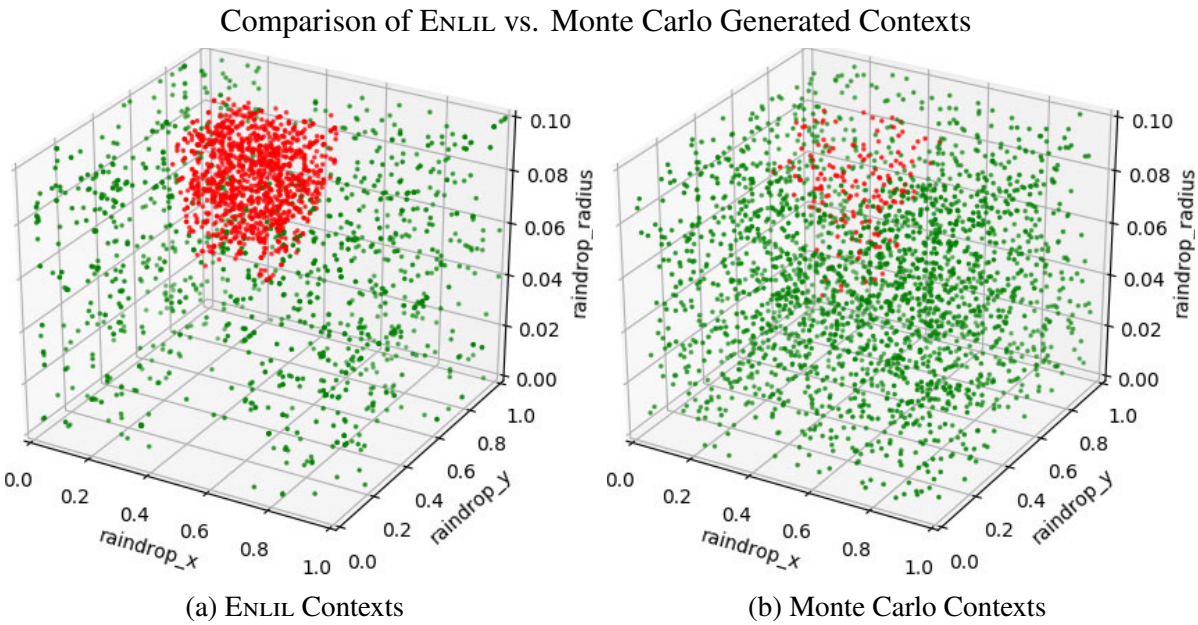
Step 2. Context Evaluation.

For each given context, a subset of the default training data is transformed to simulate the environment described by the context. Then the transformed data is processed by the DNN to determine the resulting *confusion matrix*,[2] which can be used to derive the DNN's precision and recall behavior metrics.

The scatter plot in Figure 5.4 shows how ENLIL's archived contexts compare to Monte Carlo (i.e., random-generated) contexts. For this example, ENLIL introduced a variety of raindrops onto

[2] A *confusion matrix* tallies the number of correct and incorrect classifications for each type of object [202].

the lens of a camera for an LES. Operating characteristics were defined by parameters such as the raindrop's image position and radius. Behavior categories were defined to include a category for raindrops that have no significant impact on the LES's ability to detect objects (green) and a category for raindrops that cause the LES to consistently fail to detect objects (red). ENLIL was able to generate an equal number of contexts for each behavior category, where Monte Carlo generation produced a severely imbalanced set of contexts, favoring raindrops that had no significant impact on the LES. Since ENLIL uses these generated contexts as training cases for predictive behavior models, a balanced distribution is preferred to reduce bias towards a specific behavior category. To reach a comparable number of failures cases, the Monte Carlo method would need to generate significantly more contexts (e.g., over 10X, for the examples shown in Figure 5.4). This form of *between-class imbalance* is a known problem for learning algorithms [86]. ENLIL addresses this problem using novelty search and parallel evolution to produce archives of equal sizes for each behavior category.



(a) ENLIL Contexts (b) Monte Carlo Contexts

Figure 5.4: Scatter plot comparison of ENLIL-generated (a) versus Monte Carlo-generated (b) raindrop contexts (where a raindrop occludes the view). Points correspond to raindrops with varying size/position. Colors show whether a raindrop resulted in failure for the LES (red) or had no significant impact (green). ENLIL contexts show distinct clusters of equal quantity. Monte Carlo contexts contain few problem-causing raindrops.

Step 3. Behavior Modeling.

To predict an LES's behavior under a condition of uncertainty (e.g., the presence of rainfall), the ENLIL framework constructs and trains a separate meta-level DNN from synthetically-generated data to act as a *behavior oracle* for the LES. Data for training the behavior oracle is derived from a combination of unaltered training data for the LES and the contexts archived by ENLIL in Step 1. Each entry in the ENLIL's training dataset consists of the following values: synthetic LES sensor data (x), the context parameter values (t_0) used to generate the synthetic data, and a label (t_1) for the observed behavior category for the LES when exposed to the synthetic data. ENLIL trains the behavior oracle DNN (ENLIL-ORACLE) by learning to model the relationship between x and the target variables t_0 and t_1. Once trained, the resulting ENLIL-ORACLE DNN may be used as a function to preemptively determine how the LES will react to new examples of sensor data and thus enable adaptation of the LES at run time to mitigate failure.

A high-level topology for the ENLIL-ORACLE DNN is illustrated in Figure 5.5. ENLIL-ORACLE takes sensor input data for the LES (x) and outputs an estimation of the corresponding operating context parameters (y_0) and a prediction for the LES's behavior category (y_1). ENLIL-ORACLE contains four sub-networks: an *interference isolation* sub-network, a *feature extraction* sub-network, an operating *context regression* sub-network, and a *behavior classification* sub-network.

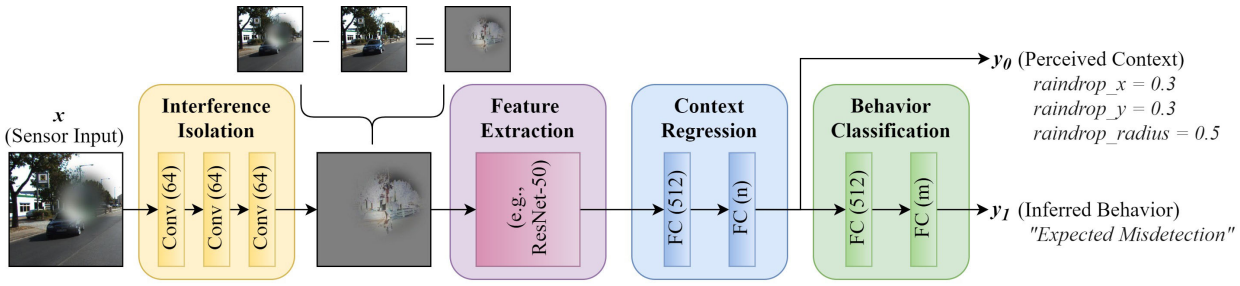


Figure 5.5: High-level illustration of the ENLIL-ORACLE DNN architecture. The architecture comprises four sub-networks. First, the suspected adverse noise is separated from the sensor input by the *interference isolation* sub-network. Second, a set of "latent" features are distilled from the isolated noise via a *feature extraction* sub-network. Next, a *context regression* sub-network converts these latent features into a "perceived" context for the current environment. Finally, a *behavior classification* sub-network predicts a behavior category for the LES.

The interference isolation and feature extraction sub-networks are responsible for reducing sensor input (x) into a set of learned features to enable behavior classification. The interference isolation sub-network learns to isolate any adverse noise introduced into the sensor input. The feature extraction sub-network learns to reduce the isolated noise into a set of latent features. The specific architectures for the interference isolation and feature extraction sub-networks are dependent on the type of sensor data used by the LES. For this study, sensor data is limited to static images, and the feature extraction sub-network has been implemented with a ResNet [87] architecture (a common method for image processing tasks).

The extracted features are provided as input to the context regression sub-network, which is responsible for estimating an operating context (y_0) to describe the given sensor input (i.e., a *perceived context*). For this study, the context regression sub-network comprises a fully-connected [73] layer with ReLU [157] activation and a second fully-connected layer of n units with linear activation (where n is the number of context parameters).

Output from the context regression sub-network is relayed to the behavior classification sub-network to classify an expected behavior category (y_1) for the perceived context (i.e., an *inferred behavior*). The behavior classification sub-network for this study comprises an fully-connected layer with ReLU activation and a second fully-connected layer of m units with *softmax* activation (where m is the number of behavior categories). Thus, ENLIL-ORACLE receives sensor data, perceives an operating context for the given data, and infers a behavior category for the LES in response to the context.

A *gradient descent* training method (Algorithm 5.2) minimizes the error of the ENLIL-ORACLE DNN by adjusting its weight values via *back propagation* [73]. The training procedure is implemented in two stages. First, the behavior classification sub-network is trained in isolation to minimize the *categorical cross entropy* between t_1 and y_1 for synthetic examples of each of the archived contexts. Second, the behavior classification sub-network is frozen, and the whole network is trained to minimize the *mean-squared-error* of t_0 and y_0 for synthetic examples of each of the archived contexts. The two-stage approach prevents the output of the behavior classifica-

tion sub-network from directly influencing the types of features extracted by the feature extraction sub-network. Instead, features are extracted to have a greater relevance to the network's perceived context.

Algorithm 5.2 ENLIL: Behavior Modeling (Step 3)

```
1:  function TRAIN-BEHAVIOR-MODEL(archive, train_data, n_epochs)
2:      dnn ← INITIALIZE-DNN()
3:      subnet ← GET-BEHAVIOR-SUBNET(dnn)
4:      for 0 to n_epochs do
5:          for (context, behavior) in archive do
6:              x ← SELECT-RANDOM(train_data)
7:              x ← TRANSFORM(context, x)
8:              t_0, t_1 ← context, behavior
9:              subnet ← FIT-WEIGHTS(subnet, x, (t_0, t_1))
10:     dnn ← FREEZE-BEHAVIOR-SUBNET(dnn, subnet)
11:     for 0 to n_epochs do
12:         for (context, behavior) in archive do
13:             x ← SELECT-RANDOM(train_data)
14:             x ← TRANSFORM(context, x)
15:             t_0, t_1 ← context, behavior
16:             dnn ← FIT-WEIGHTS(dnn, x, (t_0, t_1))
17:     return dnn
```

5.3 Empirical Validation

This section presents results from an empirical evaluation of the ENLIL framework. Experiments have been performed on an LES that uses an image-processing DNN for object detection. Multiple trials have been conducted to evaluate the impact of specific weather conditions on the LES. Synthetic examples of each weather condition are generated using both ENLIL and a Monte Carlo method. Next, predictive behavior oracles are trained using examples generated by each method for comparison. Results from these experiments answer the following research questions:

RQ1.) Can ENLIL generate a balanced distribution of operating contexts with respect to system behavior more efficiently than Monte Carlo sampling?

RQ2.) Can operating contexts from ENLIL train a more accurate behavior oracle when compared to an oracle trained by Monte Carlo contexts?

5.3.1 Datasets

Training and validation images have been acquired from a forward-facing camera mounted on an automobile, sourced from the KITTI Vision Benchmark Suite [70] and the Waymo Open Dataset [205] for autonomous driving. These datasets have been chosen, because they are publicly available, well-documented, and contain high-quality images taken from real-world driving scenarios. A majority of images from each dataset depict clear-weather scenes, and therefore, it is unknown how a DNN trained only by default data will react under introduced adverse weather conditions. Validating our approach on two independently collected datasets helps to illustrate that our techniques are not dataset dependent.

5.3.2 Experimental Setup

Object detection DNNs for this study have been implemented using Tensorflow [1] machine learning libraries and a RetinaNet [65, 139] architecture. When trained and evaluated with unaltered KITTI images, the object detector achieved a mean recall of 88% overall (94% for vehicles and 70% for pedestrians/cyclists). For Waymo images, the object detector achieved a mean recall of 95% overall (95% for vehicles and 91% for pedestrians/cyclists).

This study considers an assortment of weather conditions. Specific conditions include *fog* conditions, *lens flare* on the camera lens, *raindrops* placed on the camera lens, and *rainfall* occluding portions of the image. These weather conditions are simulated by performing parameterized image processing techniques on real-world images taken from the original KITTI and Waymo datasets. Examples of each weather condition can be seen in Figure 5.6. An unaltered image from the KITTI dataset is shown in Figure 5.6a, followed by example contexts of fog, lens flares, raindrops, and rainfall in Figures 5.6b, 5.6c, 5.6d, and 5.6e, respectively. Table 5.1 lists parameters for

each weather condition with respective value ranges. Each set of sample values drawn for these parameters is considered a unique operating context for the corresponding weather condition.

The behavior categories referenced in these experiments are the same as those specified in Section 5.2. Thresholds have been set such that *CAT* 1 ("expected object misdetection") is assigned when an introduced phenomenon causes recall to decrease more than 5% over a set of validation images, *CAT* 2 ("expected pedestrian misdetection") is assigned when there is greater than a 1% decrease in recall for pedestrians over a set of validation images, and *CAT* 3 ("expected successful detection") is assigned for all other cases. Categories are assigned to a given operating context after executing the evaluation procedure in Step 2 (Section 5.2) to compare the recall for a set of validation images with and without the newly-introduced phenomenon.

Table 5.1: Parameters for Environmental Effects

PARAMETER	PERMISSIBLE VALUES
$fog\_density$	0 to 100%
$fog\_intensity$	0 to 100%
$lensflare\_blur\_radius$	3 to 20% image width
$lensflare\_chromatic\_distortion$	0 to 100%
$lensflare\_ghost\_spacing$	0 to 100% image radius
$lensflare\_halo\_width$	0 to 100% image radius
$lensflare\_light\_x$	0 to 100% image width
$lensflare\_light\_y$	0 to 50% image height
$lensflare\_light\_radius$	0 to 10% image width
$raindrop\_x$	0 to 100% image width
$raindrop\_y$	0 to 100% image height
$raindrop\_radius$	0 to 33% image width
$raindrop\_blur$	3 to 6% image width
$rainfall\_angle$	$\pi/4$ to $3\pi/4$ radians
$rainfall\_density$	0 to 100%
$rainfall\_intensity$	0 to 100%
$rainfall\_seed$	0 to 100

Examples Contexts of Each Environmental Effect

(a) Unaltered Image

(b) Fog Examples

(c) Lens Flare Examples

(d) Raindrop Occlusion Examples

(e) Rainfall Examples

Figure 5.6: Examples of contexts generated for each environmental effect: (a) shows the original, unaltered image, (b) shows examples of fog applied to the scene, (c) shows examples of lens flare applied to the camera, (d) shows examples of raindrops on the camera lens, and (e) shows examples of rainfall applied to the scene.

5.3.3 Evaluation of Context Generation

To evaluate the distribution of operating contexts generated by ENLIL, multiple archives have been created for each environmental condition, with contexts divided into the specified behavior categories (*CAT* 1, *CAT* 2, and *CAT* 3). Separate trials were executed with alternative random seeds and varying archive sizes, ranging from 300 to 1,200 contexts. The distribution of contexts in each archive are compared to a Monte Carlo selection of contexts (i.e., parameters in Table 5.1 have been randomly sampled).

Tables 5.2 and 5.3 show the distribution of contexts generated by ENLIL and Monte Carlo sampling when applied to the KITTI and Waymo datasets, respectively. All archives produced by ENLIL were found to contain an even distribution of contexts belonging to each behavior category. In contrast, a Monte Carlo sampling method was found to consistently produce an uneven distribution of contexts with respect to each behavior category. Table 5.2 and 5.3 also show the number of samples each method would need to generate in order to achieve an even distribution at each respective archive size. For ENLIL, this is the number of candidate contexts produced by its EA. For the Monte Carlo method, this number is a projection based on the observed distribution. A key benefit for using ENLIL over a Monte Carlo method is its ability to efficiently uncover an equal proportion of contexts in cases where one behavior category is significantly less likely to occur (i.e., corner cases). To train a model that can effectively classify contexts into each behavior category, it is necessary to have relatively equal representation for each data category.

5.3.4 Evaluation of Predictive Behavior Models

To compare the usefulness of ENLIL-generated operating contexts to Monte Carlo context, separate behavior oracles have been trained with each archive generated by each method using the procedure described in Step 3 (Section 5.2). The accuracy of each behavior oracle has been measured for an independent set of test images with random environmental contexts applied, with mean results shown in Tables 5.4 and 5.5 for the KITTI and Waymo datasets, respectively.

For each archive generated, the accuracy of behavior oracles trained with ENLIL-generated con-

Table 5.2: Context Generation Results for KITTI Data

		Enlil		Monte Carlo	
	Archive	Ratio	Samples to	Ratio	Samples to
Effect	Size	*Cat* 1:2:3	Balance	*Cat* 1:2:3	Balance
Fog	300	1:1:1	574	12:1:4	1,250
	600	1:1:1	910	25:1:7	4,500
	1,200	1:1:1	1,691	19:1:6	15,285
Lens	300	1:1:1	500	1:1:15	1,810
Flare	600	1:1:1	817	1:1:15	3,384
	1,200	1:1:1	1,431	1:1:13	6,286
Rain	300	1:1:1	438	1:1:4	703
drop	600	1:1:1	810	1:1:4	1,312
	1,200	1:1:1	1,428	1:1:4	2,712
Rain	300	1:1:1	610	22:1:1	2,457
fall	600	1:1:1	883	19:1:1	4,279
	1,200	1:1:1	1,581	19:1:1	8,581

Table 5.3: Context Generation Results for Waymo Data

		Enlil		Monte Carlo	
	Archive	Ratio	Samples to	Ratio	Samples to
Effect	Size	*Cat* 1:2:3	Balance	*Cat* 1:2:3	Balance
Fog	300	1:1:1	528	4:2:1	711
	600	1:1:1	789	5:2:1	1,670
	1,200	1:1:1	1,437	6:2:1	3,650
Lens	300	1:1:1	482	1:7:55	6,667
Flare	600	1:1:1	797	1:7:51	11,966
	1,200	1:1:1	1,409	1:8:50	24,233
Rain	300	1:1:1	481	1:4:17	2,566
drop	600	1:1:1	826	1:4:18	4,690
	1,200	1:1:1	1,421	1:3:16	8,103
Rain	300	1:1:1	533	8:1:1	1,087
fall	600	1:1:1	826	8:1:1	2,094
	1,200	1:1:1	1,473	9:1:1	4,362

Table 5.4: Accuracy of KITTI Behavior Oracles

Effect	Archive Size	Enlil Accuracy (%) Cat 1 / 2 / 3	Enlil Accuracy (%) Mean	Monte Carlo Accuracy (%) Cat 1 / 2 / 3	Monte Carlo Accuracy (%) Mean
Fog	300	97.9 / 34.2 / 87.3	**73.1**	95.8 / 0.0 / 93.1	**63.0**
	600	98.7 / 89.0 / 73.5	**87.1**	96.6 / 0.0 / 93.1	**63.2**
	1,200	98.5 / 90.1 / 76.4	**88.3**	96.9 / 0.0 / 93.2	**63.4**
Lens Flare	300	84.6 / 53.9 / 82.3	**73.6**	32.4 / 0.0 / 95.6	**42.7**
	600	85.4 / 67.7 / 83.0	**78.7**	50.5 / 0.0 / 92.6	**47.7**
	1,200	85.0 / 75.3 / 84.5	**81.6**	75.0 / 40.2 / 94.8	**70.0**
Rain drop	300	87.9 / 67.9 / 92.0	**82.6**	84.3 / 26.9 / 96.5	**69.2**
	600	91.8 / 76.1 / 85.4	**84.4**	83.0 / 57.1 / 94.5	**78.2**
	1,200	89.3 / 79.2 / 88.3	**85.6**	93.2 / 60.1 / 79.7	**77.5**
Rain fall	300	87.9 / 75.1 / 26.5	**63.1**	99.6 / 1.6 / 41.6	**47.6**
	600	88.2 / 74.5 / 26.6	**63.1**	99.3 / 3.3 / 45.7	**49.4**
	1,200	89.7 / 73.3 / 27.3	**63.4**	96.6 / 4.7 / 71.5	**57.6**

Table 5.5: Accuracy of Waymo Behavior Oracles

Effect	Archive Size	Enlil Accuracy (%) Cat 1 / 2 / 3	Enlil Accuracy (%) Mean	Monte Carlo Accuracy (%) Cat 1 / 2 / 3	Monte Carlo Accuracy (%) Mean
Fog	300	95.5 / 78.3 / 72.4	**82.1**	98.7 / 75.5 / 72.2	**82.1**
	600	97.2 / 83.9 / 69.0	**83.4**	98.7 / 84.9 / 62.0	**81.9**
	1,200	97.8 / 81.1 / 76.4	**85.1**	98.0 / 88.2 / 53.3	**79.8**
Lens Flare	300	64.8 / 21.1 / 89.7	**58.5**	10.6 / 0.9 / 99.9	**37.1**
	600	80.3 / 38.4 / 75.8	**64.8**	21.8 / 2.5 / 99.4	**41.2**
	1,200	80.4 / 40.3 / 80.3	**67.0**	51.2 / 3.8 / 99.2	**51.4**
Rain drop	300	69.8 / 44.5 / 65.7	**60.0**	0.0 / 0.0 / 99.9	**33.3**
	600	75.3 / 53.6 / 86.3	**71.7**	13.8 / 6.3 / 98.3	**39.5**
	1,200	77.0 / 52.8 / 88.4	**72.8**	56.1 / 37.1 / 96.7	**63.3**
Rain fall	300	82.5 / 58.2 / 24.3	**55.0**	97.9 / 12.7 / 24.4	**45.0**
	600	81.5 / 69.6 / 25.1	**58.8**	97.9 / 10.3 / 32.6	**46.9**
	1,200	82.1 / 60.7 / 36.0	**59.6**	97.7 / 12.7 / 37.9	**49.4**

texts were either comparable to or better than those trained with Monte Carlo contexts. For smaller archive sizes, ENLIL behavior oracles were significantly more accurate. Since the distribution of Monte Carlo contexts are heavily biased, more examples are required to adequately represent all behavior categories, making behavior classification difficult for small sample sizes. These results show that ENLIL is more likely to produce an accurate behavior oracle when compared to an oracle trained only with randomly-sampled contexts, independent of archive size.

5.3.5 Threats to Validity

Since these experiments use synthetically generated data, results may not directly apply to the real world. Use of ENLIL is recommended when the "reality gap" [96] is insignificant. The simulated phenomena have been implemented to visually match real-world analogues and may help guide follow-on studies under real weather conditions to alleviate concerns about the reality gap. Figure 5.7 has been included to demonstrate an ENLIL-ORACLE applied to images of real raindrops. In these examples, ENLIL-ORACLE was able to correctly perceive the presence of a raindrop and predict that it could possibly interfere with the performance of the object detector.

The use of stochastic functions may influence the results of this study. Since the DNNs in this study have been trained by a stochastic gradient descent method and since Monte Carlo sampling is also stochastic, some variability in the results presented may be expected. For variation, all results reported have been averaged over multiple runs.

5.4 Related Work

This section overviews work related to this chapter, including model testing and predictive behavior modeling.

5.4.1 Model Testing

Automated techniques are essential when evaluating the numerous possible interactions between a cyber-physical system and its physical environment. *Model testing* has been advocated to shift

testing away from the physical implementation of a cyber-physical system to a more abstract representation [24]. The removal of hardware constraints facilitates a more efficient search of the system's interactions within a simulated environment. Related tools such as Veritas [63] and Proteus [62] have been introduced to perform adaptive run-time testing of a system by monitoring *utility functions* [36, 215] that measure a degree of system *requirements satisfaction* [45]. ENLIL's approach can be considered a form of *statistical model checking* [132], where multiple runs of a system are evaluated under varying operating conditions and statistical evidence is gathered to support claims about the system's behavior. For example, when evaluating a DNN, statistical behavior metrics may include accuracy or a *confusion matrix* based on *validation data*. By model testing with a widely diverse set of contexts, ENLIL can efficiently gather unique *counterexamples* [33] that can infer a more generalized description of how the system interacts with its environment.

ENLIL-ORACLE Demonstration on Real Raindrops

(a) Real-world Example

(b) Real-world Example

(c) "Expected misdetection"

(d) "Expected misdetection"

Figure 5.7: Proof-of-concept validation of ENLIL-ORACLE on real-world examples. (a) and (b) show images with real raindrops from the Waymo dataset [205]. (c) and (d) show that ENLIL-ORACLE was able to perceive the possibly disruptive raindrop in each image (highlighted in red).

5.4.2 Predictive Behavior Modeling

This chapter extends prior research in the use of machine learning to infer behavior models for software testing. Walkinshaw *et al.* [61, 170] described a framework that uses model inference to emulate software components. In their work, the inferred behavior model is trained on functional inputs and outputs of a tested software component. This chapter describes a comparable conceptual framework but differs in a few key aspects. First, ENLIL interacts with the external interface of a cyber-physical LES, rather than modeling the direct input and output of a single unit software function or component. ENLIL trains a model that emulates a cyber-physical system's sensor data and its expected behavior patterns. Second, based on the hypothesis that a diverse training set will enable accurate behavior models to be trained more efficiently, ENLIL uses a diversity-driven method to generate training examples for its behavior models, rather than a greedy optimization method. Finally, a major motivation of this chapter is to assess the robustness of DNNs and model the impact of uncertain conditions on DNNs, which has not been addressed in the work of Walkinshaw *et al.*

5.5 Summary

It is non-trivial but necessary to determine the behavior of safety-critical LESs under all operating contexts. This chapter has shown how the ENLIL framework enables the creation of a behavior oracle (e.g., ENLIL-ORACLE) that can predict how an LES will behave under conditions of uncertainty. ENLIL has been demonstrated to uncover a wide variety of training contexts that result in behaviors that would be underrepresented by Monte Carlo sampling. Furthermore, ENLIL can train behavior models more efficiently (i.e., requiring fewer training examples) to achieve equivalent or greater accuracy than Monte Carlo methods for introducing adversity. By using novelty search to achieve a more diverse sampling, ENLIL requires additional overhead in comparison to Monte Carlo sampling. The number of parameters in the operating specification, the archive size, and the complexity of the given evaluation procedure are major factors to consider when using ENLIL. For this study, each trial took approximately one additional hour for context generation with ENLIL versus Monte Carlo sampling on a single workstation. Since these procedures are intended to be executed at design

time, the additional overhead is considered tolerable.

Behavior oracles can potentially inform developers about training gaps and system limitations that require additional development to enable proactive mitigation strategies. Self-adaptive systems can use behavior oracles to determine how the managed system will react to conditions for which it has not explicitly been trained. For example, an autonomic manager can harness an Enlil-Oracle to anticipate failure of a learning component and trigger preventative actions at run time. Chapter 7 explores in detail a self-adaptive framework that harnesses behavior oracles to prevent unsafe operations of an autonomous vehicle at run time. Chapter 8 further explores how behavior oracles can be used in a service-oriented architecture for the purpose of establishing more trustworthy AI in the presence of environmental uncertainty.

CHAPTER 6

ASSURANCE CASES FOR SELF-ADAPTIVE SYSTEMS

For self-adaptive cyber-physical systems, such as autonomous robots, steps need to be taken to ensure requirements remain satisfied across run-time adaptations. ROS is widely used in both research and industrial applications as a middleware infrastructure for robotic systems [176]. However, ROS by default does not provide any software assurance for systems that utilize it. This chapter introduces AC-ROS (developed in collaboration with Cheng *et al.* [34]), a framework to manage run-time adaptations for ROS systems with respect to established assurance cases. Assurance cases provide structured arguments to certify that a system satisfies requirements and can be modeled graphically with GSN. AC-ROS implements a ROS-based autonomic manager, informed by GSN models, to assure system behavior adheres to requirements across run-time adaptations. For this study, AC-ROS is implemented and tested on EvoRally, a 1:5-scale autonomous vehicle (described in Chapter 2).

The remainder of this chapter is organized as follows. Section 6.1 overviews and motivates this study. Section 6.2 describes how functional assurance cases can be modeled for run-time assessment with AC-ROS. Section 6.3 describes related work. Finally, Section 6.4 provides a concluding summary for this chapter.

6.1 Overview

When autonomous cyber-physical systems are introduced into safety-critical domains, requirements violations can lead to property damage, physical injuries, and even loss of life (e.g., automobiles, aircraft, medical devices, power grids, etc.). Assurance *certification* is one approach to ensure such systems operate correctly, where *assurance cases* [158] are used to specify relationships between development artifacts (e.g., models and code) and the required evidence (e.g., test cases) needed to verify the system operates as intended. However, many cyber-physical applications rely on frameworks that do not directly address software assurance (e.g., ROS-based systems). Therefore,

a challenge is how to integrate assurance cases into such applications to certify they operate and adapt as expected. This chapter demonstrates how assurance case driven self-adaptation can be realized in ROS-based systems.

Over the past decade, research efforts have addressed assurance for self-adaptive systems [30, 218, 219], including the development of formal analysis techniques [28, 57, 234], formal modeling [93], and proactively mitigating uncertainty [8, 152, 222]. For assurance certification purposes, the GSN has been developed to specify assurance cases graphically [2]. A GSN model defines specific claims for how requirements can be satisfied and what evidence is required to support those claims. Traditionally, the certification process has largely been done manually [72, 104, 125, 186]. While work has been conducted to automate analysis of GSN models [46, 138], including security and safety properties analysis for self-adaptive systems [29, 97], GSNs have not been used to address assurance for self-adaptive ROS-based systems.

This chapter introduces AC-ROS, a MAPE-K framework that uses GSN assurance case models to manage run-time adaptations for ROS-based systems. A key insight is that the modular ROS publish-subscribe infrastructure is ideal for (1) implementing the MAPE-K control loop and event-driven adaptation and (2) supporting run-time monitoring of the environment and onboard system state. AC-ROS realizes the MAPE-K infrastructure in ROS, using GSN models at run time to guide adaptations that satisfy assurance requirements. Through automatic run-time analysis of GSN models, AC-ROS facilitates the development of assured, adaptive, and autonomous robotic systems.

This chapter presents the main components of AC-ROS and describes a "proof-of-concept" implementation for EvoRally, a 1:5-scale off-road autonomous vehicle. The sensing, control, and actuation software of EvoRally is entirely ROS-based. In addition, a simulation of EvoRally provides a means to validate AC-ROS prior to deployment. Once validated, the ROS code used for simulation can be directly uploaded to EvoRally, thereby providing an assurance-driven *digital twinning* capability [115] for self-adaptive ROS-based systems.

6.2 Functional Assurance for Robotic Self-Adaptive Systems

This section describes the core components and processes of AC-ROS, a framework for assuring the functional behavior of self-adaptive ROS-based systems. AC-ROS systematically integrates assurance information from GSN models with platform-specific information to guide run-time adaptations for a self-adaptive ROS-based autonomous platform. As such, AC-ROS enables ROS-based platforms to conform to assurance cases at run time, while adapting as necessary. The remainder of this section describes GSN modeling in more detail, how GSN models are created and analyzed at design time, and how they are used at run time to manage the adaptations.

6.2.1 Goal Structuring Notation for Functional Assurance

Assurance cases provide a means to certify that software operates as intended [72]. With assurance arguments, claims are made about how functional and non-functional requirements are met, and each claim requires supporting evidence for validation. One way to document an assurance case argument is through GSN modeling [2]. GSN allows an argument to be defined as a graphical model, with each claim represented as a *goal* and each piece of supporting evidence (e.g., test cases, documentation, etc.) represented as a *solution*. Additional elements, such as *assumptions*, *justifications*, *contexts*, and *strategies*, are provided within the notation to further expound upon an assurance argument.

A simple GSN example is shown in Figure 6.1 along with an abbreviated legend of selected elements and relationships defined by the GSN standard [2]. The figure depicts a single module (*M0*) of an assurance argument that claims a rover can successfully patrol its environment (*M0-G1*). Three sub-goals are included to support the top-level claim (*M0-G1.1*, *M0-G1.2*, and *M0-G1.3*, respectively). *M0-G1.1* claims that each waypoint is visited within an allotted timeframe, *M0-G1.2* claims that the rover avoids collisions, and *M0-G1.3* claims that the rover maintains a safe power level. *M0-G1.1* has an assumption, *M0-A1.1.1*, indicating that all the waypoints are reachable. Both *M0-G1.1* and *M0-G1.2* reference a separate module (*M1*), where additional supporting assurance

arguments are defined. Strategies *M0-S1.1.1* and *M0-S1.3.1* provide scope and clarification on *how* supported claims should be resolved. Solutions *M0-S1.1.1.1.1* and *M0-S1.3.1.1.1* define supporting evidence.

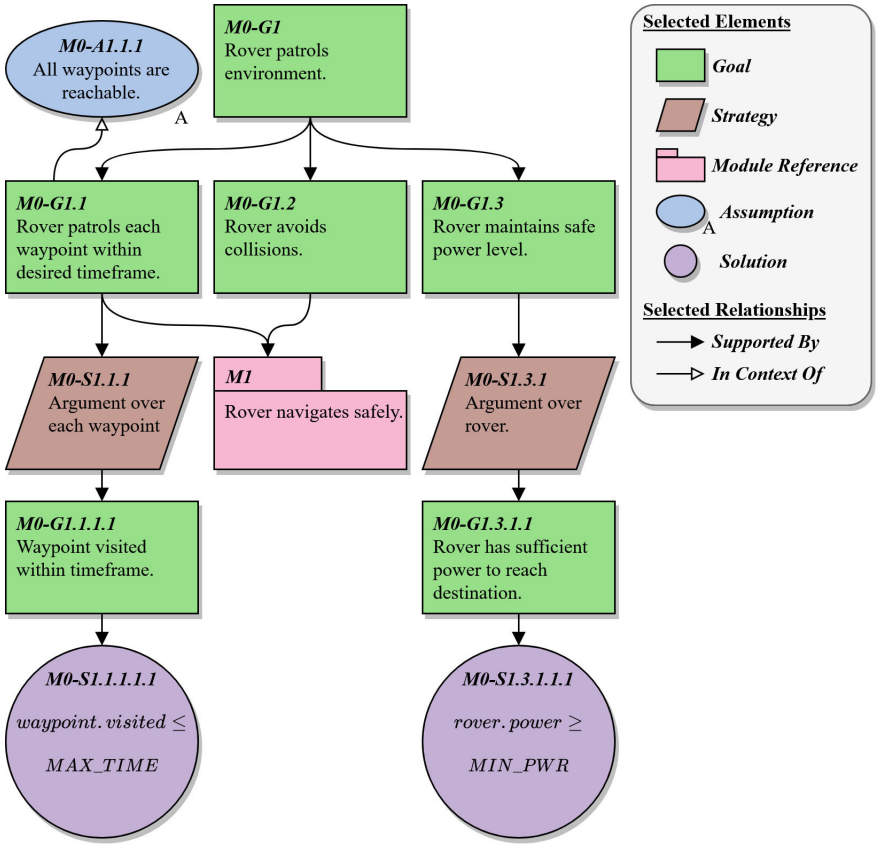


Figure 6.1: Example module of a GSN model. This module depicts an assurance argument for the claim that an autonomous rover can successfully patrol its environment.

As a graphical notation, GSN enables stakeholders with varying degrees of expertise to trace and audit the logic of an assurance case. In large-scale systems, however, the volume of documentation can become cumbersome and difficult to navigate manually [125], motivating recent work to explore automated GSN creation and refinement [46, 47], and automated GSN analysis [29, 97, 138]. Models have also been used at run time to manage self-adaptation [15, 18, 153], including assurance-focused approaches [35]. AC-ROS takes inspiration from these works to support a GSN-model driven approach to address assurance of a cyber-physical ROS-based system at run time.

6.2.2 Constructing and Evaluating an Assurance Case Model

Since AC-ROS requires assurance cases in the form of GSN models, a graphical tool is needed for developers to construct GSN models and export them into a format that can be parsed for run-time evaluation. This chapter introduces GSN-Draw for this purpose. We developed GSN-Draw to be used as a graphical user interface to enable developers to construct an assurance case with standard GSN (Version 2) conventions [2]. GSN-Draw has been implemented as a web application to enable the construction of a GSN model. GSN elements can be selected and instantiated into a "drawing" pane. Instantiated elements can then be linked together by any supported relationship type to form a connected graph. Constructed GSN models can be exported into a parsable XML[1] format. To demonstrate, Figure 6.2 shows a GSN model being constructed with GSN-Draw.

The GSN standard allows for a single GSN model to be composed of multiple interconnected modules. These modules are developed individually within a given project using the GSN-Draw tool. Figure 6.3 depicts a set of GSN modules that describe an assurance case for the EvoRally platform. For example, Figure 6.3a shows the top-level module (*M0*) of a GSN model created for an autonomous rover, whose mission is to visit a set of waypoints. This module makes the most general claim (*M0-G1*) about a rover's ability to complete its mission. A supporting argument is expressed by all elements connecting to *M0-G1*. Additional sub-modules, *M1* through *M5* (depicted in Figure 6.3b through Figure 6.3f, respectively) are developed in a similar manner but constructed separately to improve readability and enable the reuse of common sub-arguments. (Pink-tabbed boxes refer to sub-modules for a given project.)

The assurance argument laid out by a GSN model must be supported by evidence. Solution elements (such as *M0-S1.1.1.1.1* in Figure 6.3a) state what specific evidence is required to satisfy parent claims. For this study, solution elements are expressed by *utility functions* [37, 106, 152, 215]. Utility functions are derived from system and/or environmental properties that can be monitored at run time to determine the satisfaction of system requirements [177]. The utility function for

[1]The Extensible Markup Language (XML) is used to encode information in a format that is both human-readable and machine-readable [225].

M0-S1.1.1.1.1 checks the last time each waypoint was visited by the rover, and it is satisfied when each waypoint has been visited within the maximum allotted time. In a similar manner, the entire GSN model can be assessed by evaluating all utility functions against the current state of the rover. Once all utility functions have been evaluated, graph tracing can determine if the root-level claim is satisfied. GSN-INTERPRET, described by Cheng *et al.* [34] implements the functionality to evaluate GSN models from GSN-DRAW for the AC-ROS framework.

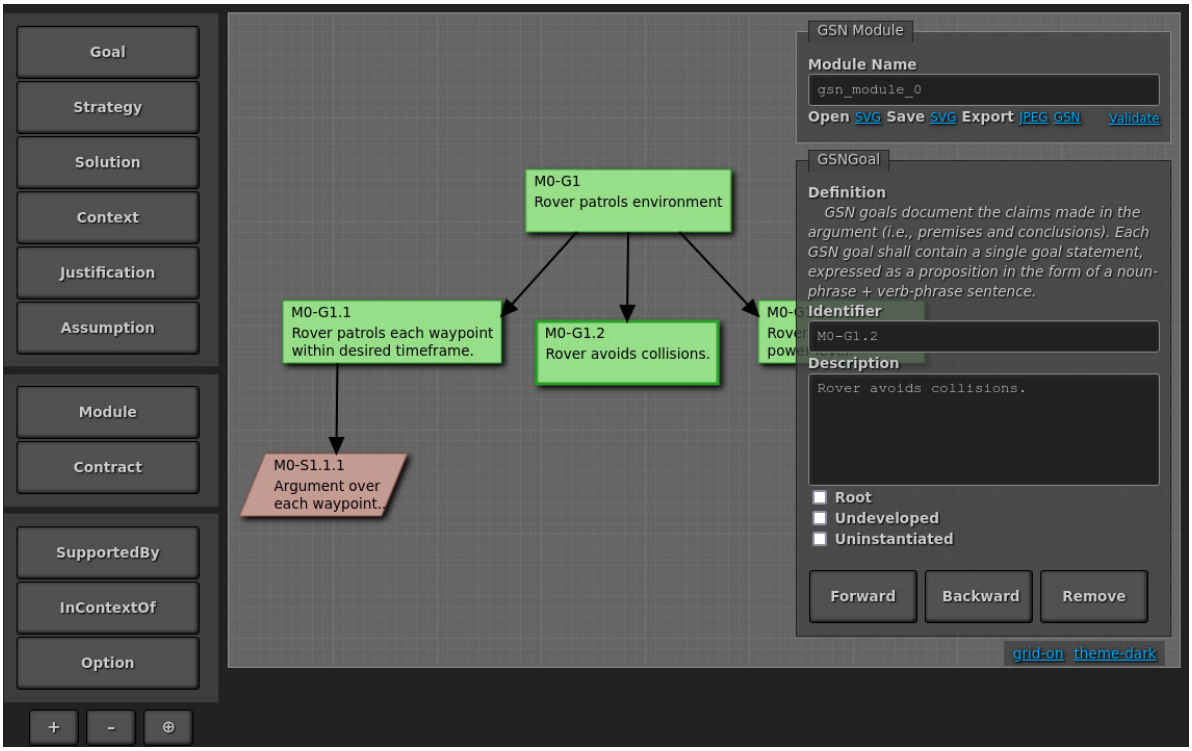
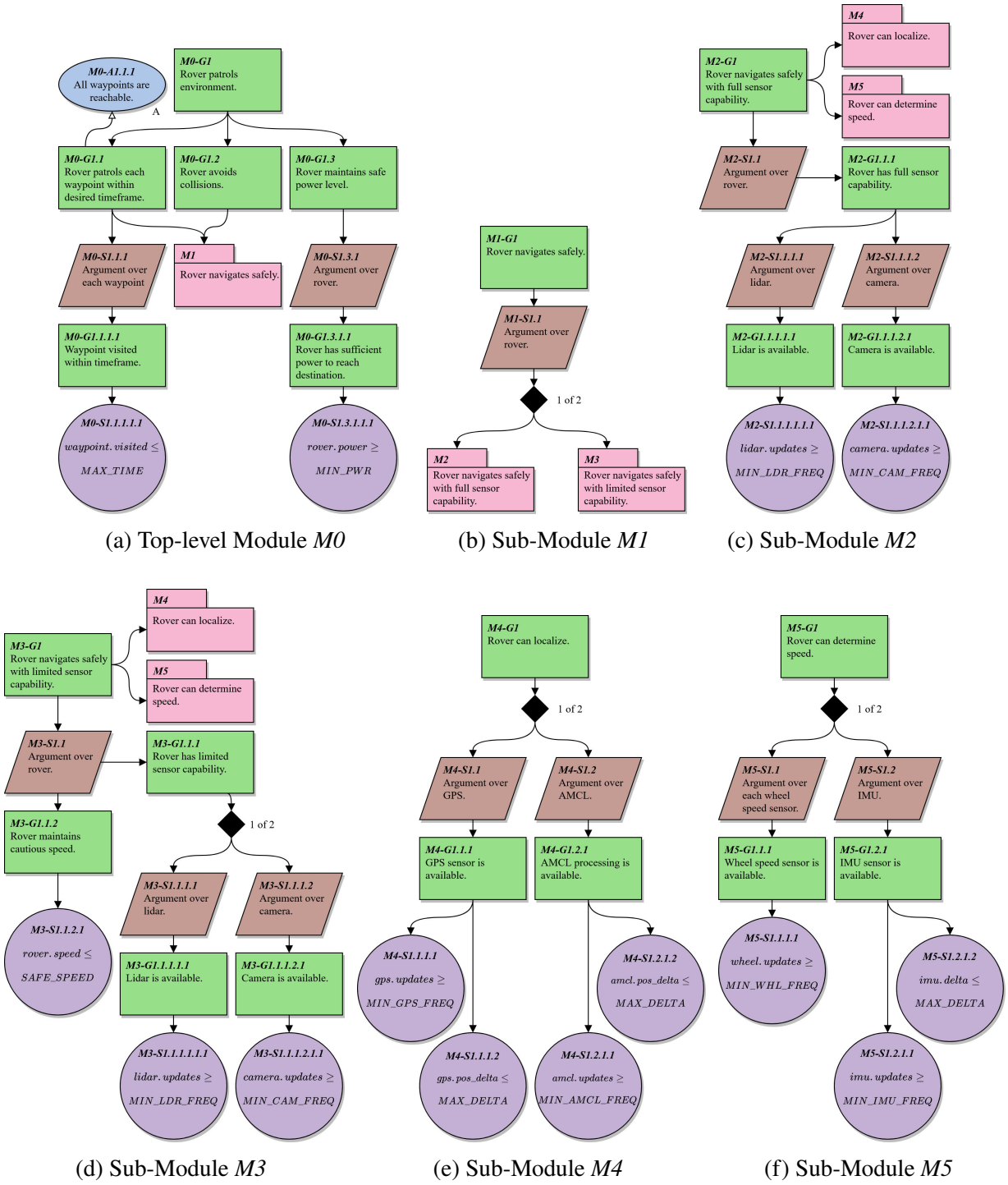


Figure 6.2: Screenshot of the GSN-DRAW user interface. GSN-DRAW enables the creation of standard GSN models that can be imported into AC-ROS.

Other assurance case editing tools provide varying degrees of support for the GSN standard [145]. A primary motivation for developing GSN-DRAW is to facilitate the use of utility functions as evidence for GSN solution elements, which is not supported by existing tools. Additionally, for adaptive systems, it is expected that assurance cases will include *optional* branches and alternative strategies for assuring different system adaptations. GSN-DRAW enables a GSN model to be created fully compliant with the GSN (Version 2) standard, including the extensions to support argument patterns (i.e., optionality and modules).



(a) Top-level Module *M0* (b) Sub-Module *M1* (c) Sub-Module *M2*

(d) Sub-Module *M3* (e) Sub-Module *M4* (f) Sub-Module *M5*

Figure 6.3: A standard GSN model of an assurance case for a patrolling rover. (a) Module *M0* argues that a rover can successfully patrol its environment. (b) *M1* argues that a rover can navigate its environment safely. (c) *M2* argues that a rover can safely navigate with full sensor capabilities. (d) *M3* argues that a rover can safely navigate with only limited sensor capabilities. (e) *M4* argues that a rover can localize its position. (f) *M5* argues that a rover can determine its speed.

6.2.3 A Shared Knowledge Base for the Managing Adaptations

AC-ROS extends the generic MAPE-K control loop design, which shares a knowledge base across each iteration of the control loop. For AC-ROS, the knowledge base is an aggregate collection of static and dynamic data stores relevant to ROS-based platform and the GSN models for its assurance. For brevity, this section uses the prefix "KB" to refer to individual data stores within the aggregate knowledge base for AC-ROS. Descriptions for each data store follow.

KB1. GSN Models

GSN models for all relevant assurance cases are stored in KB1 in an XML format.

KB2. Mapping to ROS Topics

In order to determine how utility function parameters relate to platform-specific ROS topics, KB2 contains a mapping for each utility parameter referenced in KB1 to a ROS topic for the managed platform. For example, the utility function for *M0-S1.1.1.1.1* in Figure 6.3a references a *rover.power* parameter, which is linked to the `charge_percent` ROS topic via this mapping. This mapping is not generated automatically, since it is platform-specific and requires an understanding of the available ROS topics.

KB3. ROS Launch Files

Information needed to configure and initialize the run-time monitoring processes for AC-ROS are stored in KB3. This information is stored as ROS launch files, which are automatically generated to provide the launched ROS nodes with details about which ROS topics to monitor and their corresponding utility parameters (based on the data in KB2).

KB4. Adaptation Tactics

Adaptation tactics [37] for the ROS-based platform are stored in KB4. Each tactic comprises a *context* (i.e., under which conditions a given adaptation is applicable), sequence of *actions*, and consequences of adaptation (i.e., effect(s) of adaptation). Each adaptation tactic is a high-level

sequence of commands to enable changes in behavior "modes" (e.g., a mode change could switch sensors from cameras to lidar for obstacle detection). These tactics are described by configuration files that are loaded at the beginning of run time.

KB5. System State

Platform-specific information gathered by AC-ROS at run time is stored in KB5. This data store may include raw data (e.g., wheel speeds), meta information (e.g., frequency and distributions of data stream samples), and various configuration parameters (e.g., settings for obstacle detection mode).

6.2.4 Design-time Activities

The DFD in Figure 6.4 illustrates the design-time steps for modeling an assurance case and using it to configure the run-time ROS environment. Descriptions for each step follow.

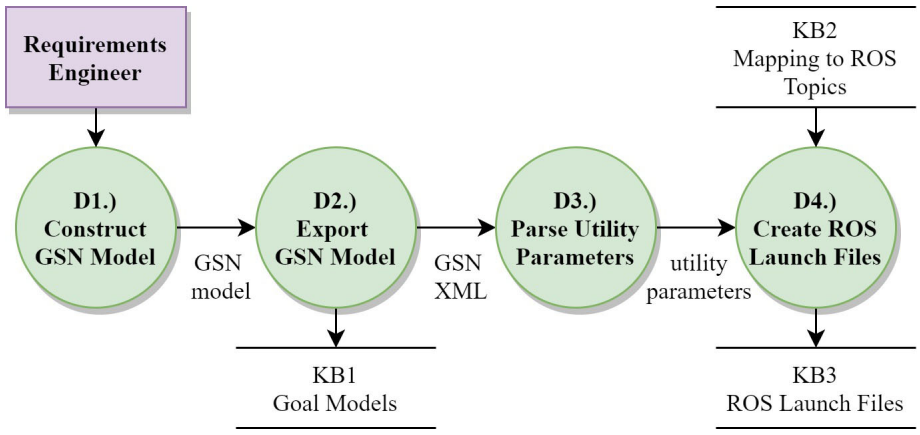


Figure 6.4: Data flow for the process AC-ROS takes to prepare GSN models and ROS configuration at design time. Circles, boxes, arrows, and parallel lines respectively represent computational steps, external entities, data flow, and data stores.

Step D1. Construct GSN Model

GSN-Draw enables a requirements engineer to interactively develop a graphical model for an assurance case in the form of one or more GSN modules.

Step D2. Export GSN Model

GSN-Draw exports XML files to KB1, with each corresponding to a respective GSN module. Exported files preserve the structure and contents of the GSN model, including element properties, relationship dependencies, and associated utility functions.

Step D3. Parse Utility Parameters

XML files exported from GSN-Draw are parsed to determine the utility functions associated with each solution of the GSN model and then extracts the referenced functional parameters. For example, this step would parse the utility parameter *rover.power* from solution *M0-S.1.3.1.1.1* in Figure 6.3a.

Step D4. Create ROS Launch Files

For run-time monitoring, AC-ROS references the utility parameters specified in the GSN model (e.g., *rover.power*). However, to collect relevant data from the managed ROS-based platform for each utility parameter, AC-ROS requires a mapping to the relevant ROS topics. Step D4 creates ROS launch files that define how its run-time monitoring ROS nodes should be instantiated (i.e., to which topics each should subscribe and the corresponding utility function in the GSN model). The resulting launch files are stored in KB3 for later use by AC-ROS at run time. The mapping between utility parameters and ROS topics are provided by KB2. This use of indirection and abstraction enables the requirements engineer to design a GSN model without needing to know implementation details of as specific ROS-based platform, thereby facilitating reuse of the GSN model for alternative platforms.

6.2.5 Assuring System Adaptations at Run time

The DFD in Figure 6.5 illustrates the run-time steps for AC-ROS to manage system adaptations such that they satisfy an assurance case for the managed ROS-based autonomous platform. These steps follow the general design of a MAPE-K control loop, and descriptions for each step follow.

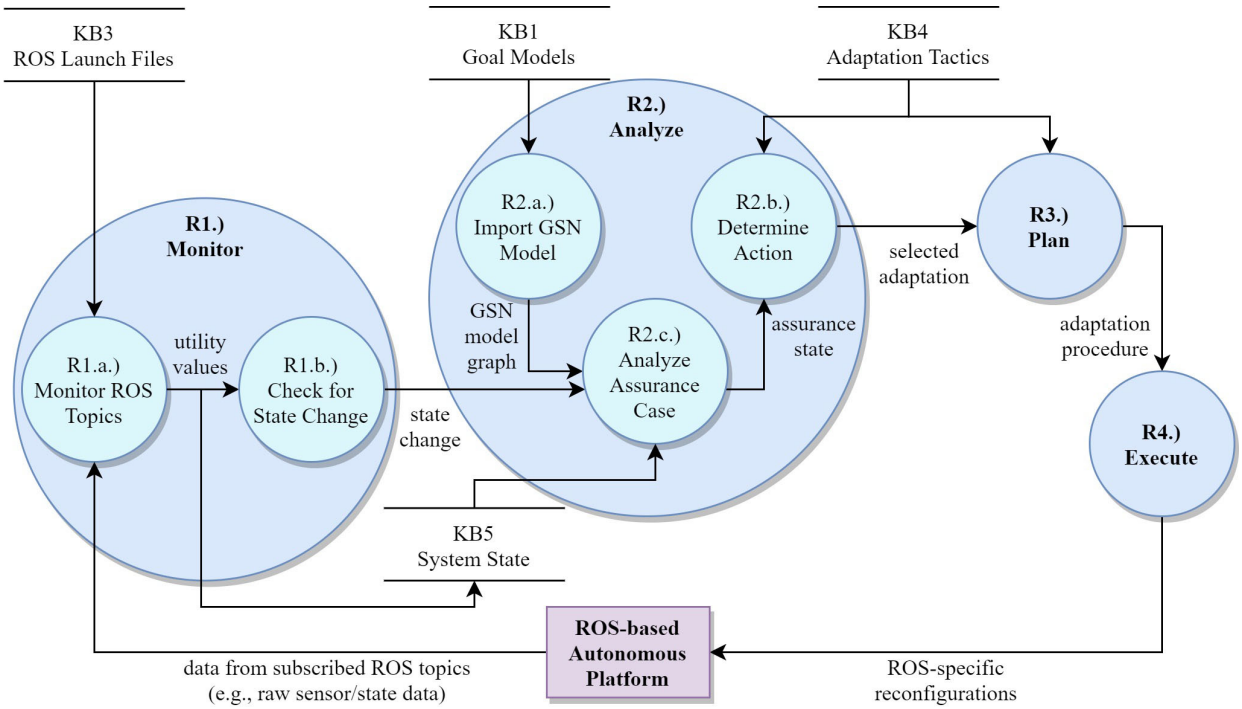


Figure 6.5: Data flow for run-time processes involved with managing adaptations of a ROS-based platform.

Step R1. Monitor

Step R1 comprises one or more *monitor* processes (i.e., ROS nodes) that observe the managed autonomous platform and update the knowledge base (KB5) accordingly. Launch files (KB3) for each monitor process determine which subsystem of the managed platform to observe (e.g., camera, lidar, etc.), the relevant ROS topics, and the associated utility parameters. *Monitor* processes are also capable of performing preprocessing to further refine measurements from raw ROS data (e.g., measure frequency, infer speed from GPS measurements, etc.). As each *monitor* process observes data from the managed platform and updates the knowledge base (Step R1.a), it also checks for any violations to its associated utility function (Step R1.b).

Monitor processes are instantiated to account for every corresponding utility function parameter referenced by the GSN model, including all graph branches in the case of optionality. During run time, it is assumed all optional branches of the GSN model graph are available to be satisfied and are thus monitored. However, if needed, AC-ROS supports adaptive monitoring by allowing nodes to subscribe to, and unsubscribe from, the relevant ROS topics during run time.

Step R2. Analyze

Step R2 (i.e., the *analyze* process) determines whether the managed platform must adapt to ensure that it delivers acceptable behavior with respect to the assigned assurance case. When Step R2 is initiated, GSN models are loaded from KB1 and stored in an internal graph representation (Step R2.a). The source XML data specifies a root goal element for each GSN module and parent-child relationships for each GSN element. Subgraphs are constructed for each module to preserve the specified relationships from root to leaf nodes. Links are then established to the module subgraphs in order to produce an internal graph of the entire GSN model. Step R2 is activated periodically, as well as when a *monitor* process from Step R1 reports a utility function violation. AC-ROS retrieves current state information from KB5 and references the given GSN models to determine if the managed platform complies with the modeled assurance case (Step R2.b). AC-ROS evaluates the GSN graph via a depth-first traversal to determine the satisfaction of leaf-level solution elements (by evaluating their associated utility functions). Example utility functions can be found in the leaf nodes of GSN modules in Figure 6.3. Based on this evaluation of the GSN model, AC-ROS determines whether any adaptations are necessary When the GSN model is not satisfied, the existing adaptation tactics are reviewed based on the descriptions of their contexts and consequences of adaptation impact. For this work, it is assumed that corresponding adaptation tactics have been predefined to cover each combination of utility function violations. Ongoing research includes the use of reinforcement learning to determine which adaptation tactics are best suited for the given context. Once an appropriate adaptation tactic has been identified (Step R2.c), a reference to the selected adaptation is sent to the *plan* process.

Step R3. Plan

When Step R3 (i.e., the *plan* process) receives a reference to a selected adaptation tactic, the appropriate sequence of actions to implement the tactic (i.e., the tactical procedure) are fetched from KB4. The tactical procedure ensures that the managed system is in a quiescent state before the system adapts [117]. For example, a tactic for changing obstacle detection modes might require

that the robot slow down first to reduce the possibility of collision.

Step R4. Execute

Step R4 (i.e., the *execute* process) receives an adaptation procedure and translates it into platform-specific commands. These commands are published to ROS topics to be carried out by the managed platform.

6.3 Related Work

This section overviews related work in the area of assurance-driven cyber-physical systems, self-adaptive cyber-physical systems, and the evaluation of assurance cases.

While other work has explored model engineering with ROS [7, 19, 88] and self-adaptation with ROS [4, 32], this chapter specifically brings together self-adaptation with MAPE-K feedback to support assurance through GSN assurance cases for ROS-based systems. Calinescu *et al.* introduced ENTRUST [29], a method for assurance-driven adaptation. Like AC-ROS, ENTRUST implements a MAPE-K control loop for self-adaptive systems, guided by GSN models. However, during the analysis step of the MAPE-K loop, ENTRUST uses a probabilistic verification engine to assess stochastic finite state transition models (i.e., Markov chains) of the controlled system. In contrast, AC-ROS has been designed to take advantage of ROS, and instead of relying on a model of the autonomous platform for assurance assessments, AC-ROS directly monitors real system data via ROS topics and evaluates assurance arguments based on utility functions.

The RoCS framework [179], created by Ramos *et al.*, implements a MAPE-K control loop to manage adaptations for robotic systems, with potential for integration into ROS. However, a key difference between RoCS and AC-ROS is that AC-ROS uses GSN models to guide and assure adaptations at run time. While RoCS and AC-ROS both realize the MAPE-K control loop, AC-ROS ensures that safety requirements are upheld with each executed system adaptation.

Alternative methods to automated evaluation of assurance cases have been explored. Lin *et al.* [138] introduced a method that uses Dempster-Shafer theory to calculate confidence for an assurance case. AC-ROS acquires evidence directly via the evaluation of utility functions,

and thus confidence is not explicitly considered when evaluating evidence for an assurance case. Traditionally, assurance cases have focused on certifying functional requirements for the purpose of safety, although recent work has addressed assurance for security requirements [97]. Thus far, AC-ROS has been implemented to manage only functional assurance cases.

6.4 Summary

This chapter introduced the AC-ROS framework to support an assurance case driven approach to develop and manage ROS-based adaptive systems. AC-ROS leverages the ROS publish-subscribe communication paradigm to integrate an assurance case (modeled by GSN) into a MAPE-K control loop to enable assured run-time adaptation. Chapter 7 further explores how system requirements models can drive MAPE-K adaptations to maintain safe operation of an LES. Finally, Chapter 8 combines these technologies into a single cohesive service-oriented framework to develop more trustworthy LESs.

CHAPTER 7

GOAL MODELS FOR LEARNING-ENABLED SYSTEMS

Increasingly, safety-critical systems include AI and machine learning components (i.e., LECs). However, when behavior is learned in a training environment that fails to fully capture real-world phenomena, the response of an LEC to untrained phenomena is uncertain, and therefore cannot be assured as safe. Automated methods are needed for self-assessment and adaptation to decide when learned behavior can be trusted. This work introduces MoDALAS (developed in collaboration with Chan *et al.* [129]), a model-driven framework to manage self-adaptation of an LES to account for run-time contexts for which the learned behavior of LECs cannot be trusted. The MoDALAS framework enables an LES to monitor and evaluate goal models at run time to determine whether or not LECs can be expected to meet functional objectives. Using this framework enables stakeholders to have more confidence that LECs are used only in contexts comparable to those validated at design time.

The remainder of this chapter is organized as follows. Section 7.1 overviews and motivates this chapter. Section 7.2 describes how goal models are processed by MoDALAS. Section 7.3 describes how MoDALAS manages LESs at run time to mitigate the impact of uncertain conditions. Section 7.4 presents an implementation of MoDALAS for the EvoRally autonomous platform. Section 7.5 reviews related work. Finally, Section 7.6 provides a concluding summary for this chapter.

7.1 Overview

The integration of machine learning into autonomous systems is potentially problematic for high-assurance [30], safety-critical [125] applications, particularly when training coverage is limited and fails to fully represent run-time environments. In addition to meeting functional requirements, safety-critical LESs must account for all possible operating scenarios and guarantee that all system responses are safe [212]. However, machine learning components, such as DNNs [73], are

associated with uncertainties concerning generalizability [103], robustness [230], and interpretability [112]. A rigorous *software assurance* [186] process is needed to account for these issues of uncertainty. This chapter proposes a goal-oriented modeling approach to address the assurance of LECs and manage the run-time adaptation of a cyber-physical LES.

Although verification of an LEC can include steps to validate learning algorithms *offline*, additional *online* steps are needed to provide confidence that an LES will perform reliably and safely at run time [84, 193]. At design time, mathematical proofs can show that convergence criteria of a learning algorithm are satisfied, and empirical testing through cross validation can help estimate the generalizability of a trained LEC. However, when all conceivable situations cannot be included in training/validation data, methods are needed to dynamically monitor and assess the trustworthiness of LECs to determine whether assurance evidence collected at design time remains valid for previously unseen run-time conditions. More importantly, an LES must be able to determine when results from an LEC can, or *cannot*, be trusted to correctly respond to current conditions.

This chapter presents a framework for Model-Driven Assurance for Learning-enabled Autonomous Systems (MoDALAS) through the use of goal-oriented *models at run time* [129]. With run-time self-assessment of its LECs, a goal model-driven LES can adapt to satisfy system requirements when exposed to environmental uncertainty. MoDALAS supports the run-time monitoring of an LES with respect to functional goal models, assesses the trustworthiness of LECs, and adapts the LES to mitigate the use of LECs in untrusted contexts. Specifically, predictions of LEC behavior under various operating conditions are directly referenced by goal models that include potential conflicts and resolutions relating to different LEC behaviors (including untrusted contexts). Moreover, MoDALAS enables seamless monitoring and management of both LECs and "traditional" system components (both hardware and software) that do not involve machine learning.

MoDALAS establishes online system verification by the run-time monitoring of KAOS goal models [124] for functional requirements [106]. Controlled by a self-adaptive feedback loop [105], MoDALAS includes *behavior oracles* (introduced in Chapter 5) for each LEC. Analogous to a *test*

oracle [9], a behavior oracle predicts how an LEC will behave in response to given inputs. With MoDALAS, behavior oracles are used to assess the capability of LECs under varying run-time conditions. The resulting self-adaptive LES can then detect when its LECs are operating outside of performance boundaries and adapt accordingly, including possible transitions to fail-safe modes in extreme circumstances. By combining goal models with behavior oracles for an LES, developers can specify requirements concerning the confidence in an LEC and implement alternative strategies to ensure assurance claims are supported.

A proof-of-concept prototype of MoDALAS is described for an autonomous rover LES equipped with a camera-based object detector LEC [100]. DNNs play two roles in this work: 1) a DNN provides object detection capabilities for an autonomous rover and 2) a separate DNN acts as a behavior oracle within MoDALAS to assess the object detector's performance at run time. The object detector has been trained offline by a supervised training dataset, which includes mostly clear-weather examples. However, the autonomous rover must be assured to also function as expected in adverse weather. Without MoDALAS, the object detector would be used regardless of how closely run-time contexts match its trained experience, which could be detrimental for adverse environmental conditions (e.g., haze from a dust plume at a construction site). In contrast, MoDALAS determines when the rover's object detector is operating outside of training coverage. Furthermore, MoDALAS enables the rover to adapt accordingly by entering a more cautious operating mode or, under extreme conditions, a fail-safe mode.

7.2 Constructing Goal Models for Autonomous Systems

This section describes how modeling and specification technologies are applied at design time in MoDALAS to model high-level system goals for an LES.

7.2.1 Developing Assurance Cases

MoDALAS assumes that assurance cases and goal models have been constructed and validated at design time through methods such as *model checking* [41]. In this chapter, assurance cases

are modeled using GSN, though alternative methods may also be used to describe an assurance case [186]. A simple example GSN model is shown in Figure 7.1, which claims a rover will navigate its environment safely (claim *C1*). Strategies are implemented to support claim *C1* through offline validation (strategy *S1*) and run-time analysis (strategy *S2*). At design time, software testing, model checking, and formal analysis are conducted offline to support assurance claim *C2*, with results provided as evidence in solutions *Sn1*-*Sn3*. At run time, evaluation of a KAOS goal model for the rover provides evidence (solution *Sn4*) to demonstrate system requirements remain satisfied under changing run-time conditions (claim *C3*). As such, a GSN model provides context for our work, where evidence generated for assurance solution *Sn4* is provided by the evaluation of KAOS goal models at run time.

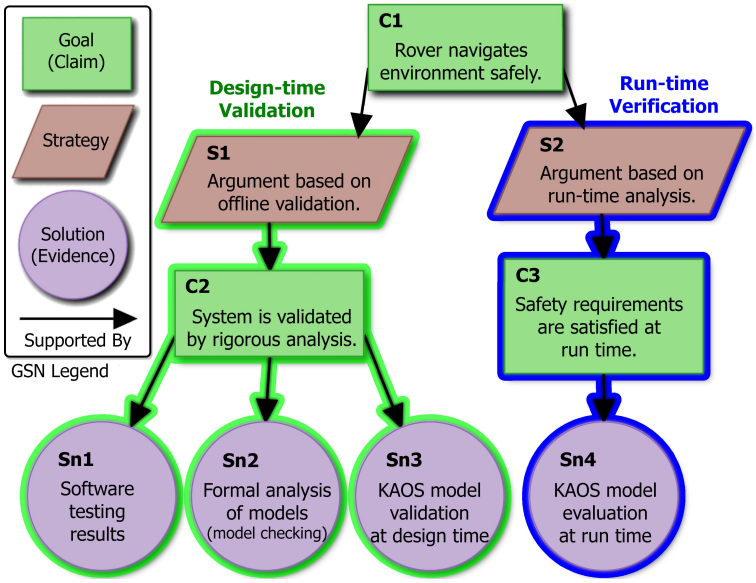


Figure 7.1: Example GSN assurance case for *design-time* and *run-time* validation of a rover. At design time, validation is supported by formal proofs, test results, and simulation (highlighted in green). At run time, verification is supported by the evaluation of a KAOS goal model (highlighted in blue).

7.2.2 Developing Goal Models

This section describes how KAOS [124] goal models are used to hierarchically decompose high-level system goals into leaf-level system requirements in MoDALAS. Whereas the focus of GSN

is on software certification, KAOS goal modeling supports a hierarchical decomposition of high-level functional and performance objectives into leaf-level system requirements (i.e., goal-oriented requirements engineering [131]). KAOS goal models enable a formal goal-oriented analysis of how system requirements are interrelated as well as threats to requirements satisfaction. *Goals* represent atomic objectives of a system at varying levels of abstraction, with sub-goals refining and clarifying higher-level goals. Any event threatening the satisfaction of a specific goal is represented as an *obstacle*. Resolutions for obstacles can be specified by attaching additional sub-goals with alternative system requirements to the corresponding obstacle. Finally, *agents* (i.e., system components) are assigned responsibility for each system requirement. KAOS goal models enable developers to decompose the expected behavior of a software system, including information about threats to specific system requirements and how system requirements relate to each system component.

Figure 7.2 shows a legend for reading KAOS goal models, while Figure 7.3 shows an example KAOS goal model for a rover that must navigate its environment. In this example, a rover is expected to sense objects in its environment and plan its trajectory around objects according to object type (e.g., when pedestrians are present (*G10*) or not (*G9*)). The rover implements a DNN-based *object detector* that can *locate* zero or more objects within a camera image and *classify* the type of each object [100]. The trustworthiness of the object detector depends on how similar the run-time environment is to its training experience. The rover also ensures there is sufficient braking power (*G20*) using sensor values from the tire pressure monitoring system and friction sensor. In Figure 7.3, *utility functions* (shown in yellow) are attached to the bottom of each goal. Utility functions enable the KAOS goal model to be evaluated at run time to determine goal satisfaction.

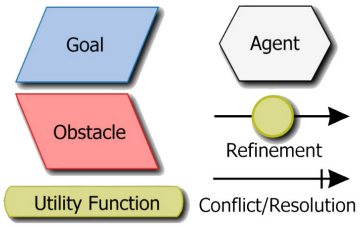


Figure 7.2: Legend key for interpreting the KAOS goal model notation.

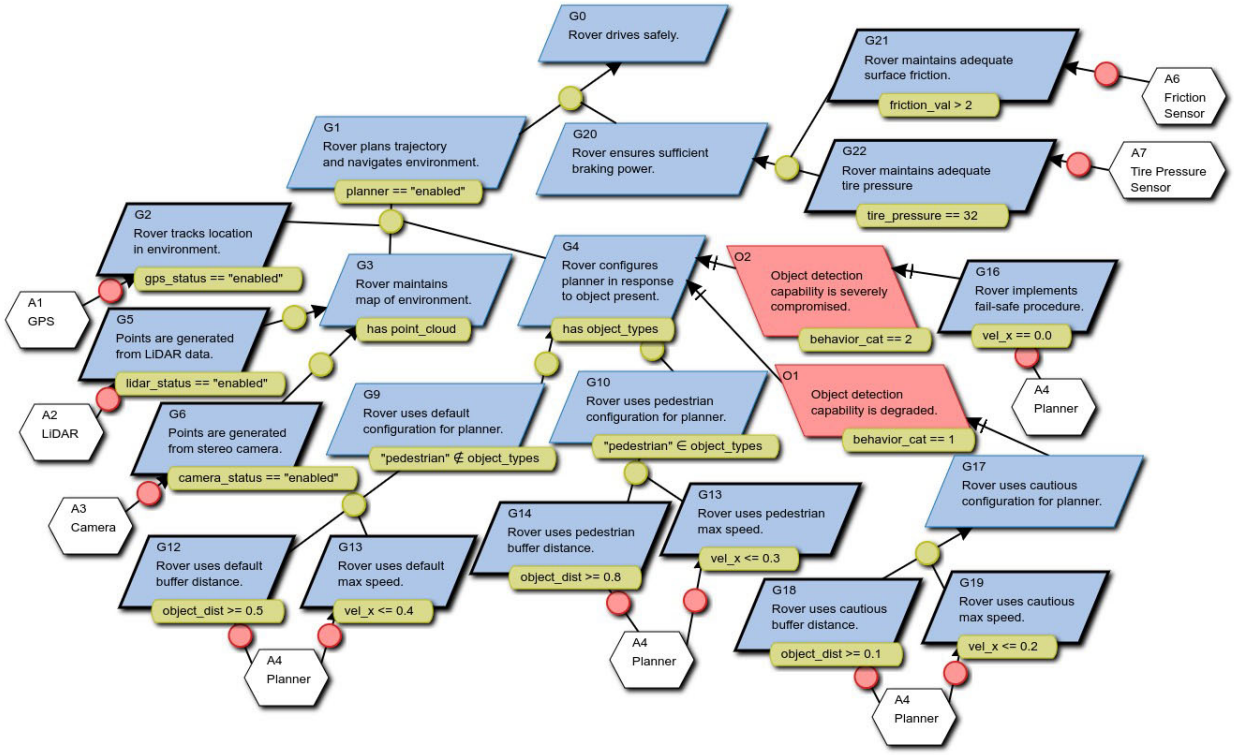


Figure 7.3: Example KAOS goal model. *Goals* (blue parallelograms) represent system objectives. The top-level goal (*G1*) is refined by sub-goals down to leaf-level system requirements. *Agents* (white hexagons) represent entities responsible for accomplishing requirements. *Obstacles* (red parallelograms) represent threats to the satisfaction of a goal (e.g., *O1* and *O2*).

In KAOS notation, any event that can threaten the satisfaction of a goal is represented as a KAOS *obstacle*. In Figure 7.3, obstacles *O1* and *O2* represent events in which the object detector is operating in a state not explored during design time. Obstacle *O1* represents events where the object detector's performance is *degraded* (i.e., statistical performance is less than ideal), and obstacle *O2* represents events where the object detector is *compromised* (i.e., statistical performance is unacceptable). When the object detector is compromised, goal *G16* is given as an obstacle resolution, where the rover is expected to perform a fail-safe procedure (e.g., halt movement). When the object detector is only degraded, goal *G17* is given as a resolution, where the rover is expected to slow down and increase its minimum buffer distance from objects encountered in the environment. Additional KAOS obstacles and resolutions can be included, depending on the LEC, targeted behavior categories, and LES requirements.

Similar to the GSN-DRAW tool introduced in Chapter 6, a graphical tool is needed for developers

to construct KAOS models in a format that can be parsed for run-time evaluation. This chapter introduces KAOS-Draw for this purpose. KAOS-Draw has been implemented as a web application to enable the construction of a KAOS model. KAOS elements can be selected and instantiated into a "drawing" pane. Instantiated elements can then be linked together by any supported relationship type to form a connected graph. Constructed KAOS models can be exported into a parsable XML format. To demonstrate, Figure 7.4 shows a KAOS model being constructed with KAOS-Draw.

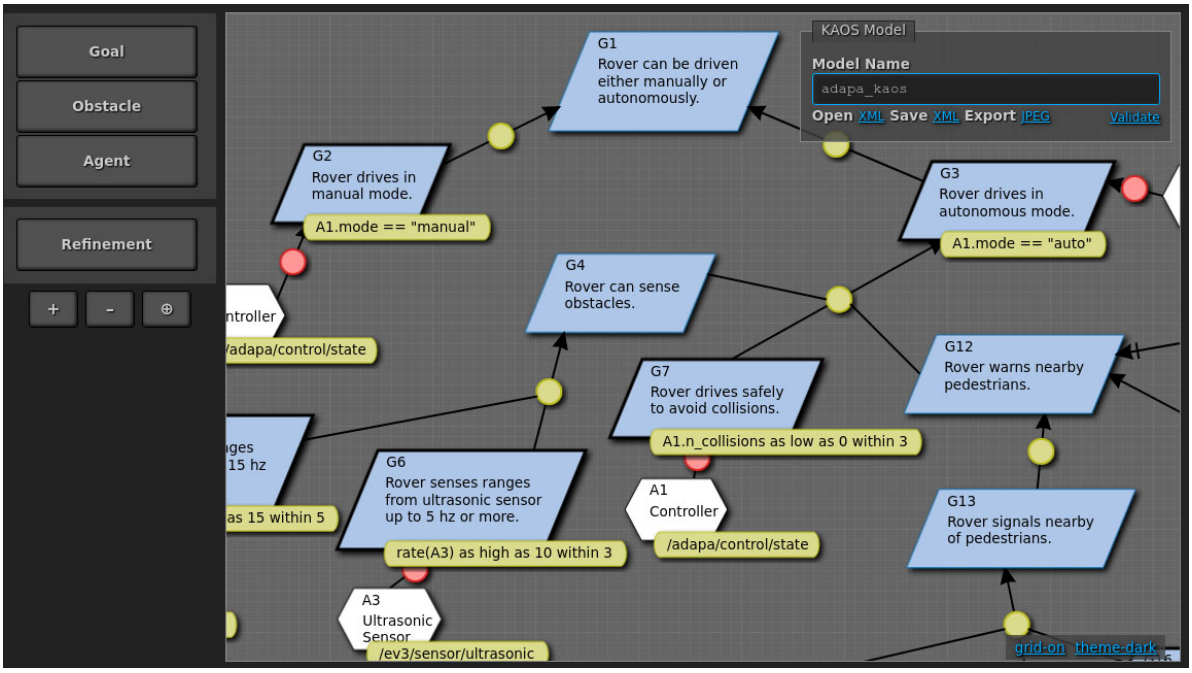


Figure 7.4: Screenshot of the KAOS-Draw user interface. KAOS-Draw enables the creation of standard KAOS models that can be imported into MoDALAS.

7.2.3 Relaxing Goal Models

In this chapter, the RELAX language [222] is used to explicitly specify uncertainties affecting an LES. RELAX is a requirements specification language that enables developers to identify, evaluate, and "relax" brittle requirements to address and mitigate uncertainty factors during run time. During requirements engineering, developers may describe system behaviors with strict and highly constrained properties. However, due to the numerous sources of uncertainty potentially impacting an LEC, it may not always be possible to strictly satisfy all requirements. RELAX

allows for non-invariant requirements be temporarily unsatisfied due to uncertain environmental and onboard conditions. RELAX operators add flexibility to the conditions for which a given requirement is considered satisfied, thereby adding the notion of degrees of satisfaction (i.e., "satisficement" [124, 135]) in a goal model. For example, RELAX operators such as *AS CLOSE AS POSSIBLE* can be used to reduce the brittleness of a given goal to three RELAX elements (i.e., **ENV**, **MON**, and **REL**) [222]. Table 7.1 provides examples of RELAX operators, with the names of the operators provided in the first column and corresponding descriptions provided in the second. RELAX also includes traditional temporal logic operators, but they have been mapped to fuzzy logic operators instead [222].

Table 7.1: Example of RELAX operators and uncertainty factors [63, 222]

Operator	Description
Modal operators	
SHALL	Requirement must hold.
MAY...OR	Requirement specifies one or more alternatives.
Temporal operators	
EVENTUALLY	Requirement that must hold eventually.
UNTIL	Requirement must hold until a future position.
BEFORE/AFTER	Requirement must hold before or after a particular event.
AS EARLY AS POSSIBLE	Requirement should hold as soon as possible.
AS LATE AS POSSIBLE	Requirement should be delayed as long as possible.
AS CLOSE AS POSSIBLE TO [frequency t]	Requirement happens repeatedly, though frequency may be relaxed.
Ordinal operators	
AS FEW/MANY AS POSSIBLE	Requirement specifies a countable quantity, though the exact count may be relaxed.
AS CLOSE AS POSSIBLE TO [quantity q]	Requirement specifies a countable quantity, though the exact count may be relaxed.
Uncertainty factors	**Description**
ENV	Defines a set of properties that define the system's environment.
MON	Defines the set of properties that can be monitored by the system.
REL	Defines the relationship between **ENV** and **MON** properties.
DEP	Defines dependencies between (relaxed and invariant) requirements.

RELAX enables developers to create more flexible requirements to ensure robustness against uncertainties. However, modifications to textual requirements require run-time evaluation to verify a threshold of satisfaction. During run time, LESs monitor system values and use utility functions to assess whether system performance and/or configuration satisfy the current goal model. Traditionally, utility functions return a Boolean value (i.e., 0 or 1) based on goal satisfaction. To address run-time uncertainty, RELAX operators are mapped to *fuzzy logic* semantics [79, 231]. Fuzzy logic enables developers to specify a partially satisfied goal, where a utility function can return real values ranging from 0 (i.e., not satisfied) to 1 (i.e., satisfied). A goal that returns a partially satisfied utility function is known as *satisficed* [124], permitting system requirements to be verified as satisfied with a degree of fuzziness.

Since fuzzy logic allows utility functions to return real numbers, goal refinement (i.e., *AND* and *OR* goal decompositions) for parent goal evaluation must be redefined. In the parent goal of *RELAX*-ed goals, the goal's utility function is evaluated by applying mathematical operations (e.g., min and max) on sub-goals' satisficement values. While several popular approaches to fuzzy logic evaluation exist, this work uses Zadeh fuzzy operators [111], a common convention for resolving fuzzy logic (e.g., conjunctions, disjunctions, and implications). Using Zadeh fuzzy operators, when the sub-goals are related by an *OR* relationship, the maximum value of all sub-goals' utility functions determine the evaluation of the parent goal. If the sub-goals are part of an *AND* relationship, then the minimum values determine the parent goal instead. A parent goal may be converted to Boolean satisfaction if the evaluation of the value of the *RELAX*-ed sub-goals exceeds a specified threshold (e.g., 0.5). To illustrate an example of a RELAX specification, consider a component of an autonomous vehicle that detects obstacles. A traditional requirement may be as follows:

S1: The system *SHALL* detect obstacles within 10 meters.

This requirement represents an ideal situation. However, instead of a rigid requirement, a developer may wish to relax the requirement to account for uncertainty factors (e.g., speed variance of two vehicles, the sensitivity of the sensors, etc.). For example, **S1** may be modified to the

expression **S1′** if the vehicle is traveling below 10 meters per second, since there is more time for the system to react to detected obstacles.

S1′: The system *SHALL* detect obstacles *AS CLOSE AS POSSIBLE* to 10 meters.

ENV: location of obstacle

MON: obstacle detection system

REL: system detects obstacle

Using **S1′**, the system can still handle the requirement of "detect obstacle within 10 meters", and also support a more flexible requirement should the system detect an obstacle within 8 meters. Specifically, developers can trade-off rigid Boolean utility functions (i.e., the system detected obstacle within 10 meters or not) for fuzzy logic utility functions (e.g., degree of *satisficement* from 0 to 1) under RELAX. As such, the system can adapt and temporarily trade-off non-critical requirements to maintain the satisfaction of more critical requirements.

To address environmental uncertainties, developers may use RELAX to temporarily allow specific requirements to be relaxed within acceptable ranges. During the design step, the developers identify non-invariant requirements that may be relaxed. Next, developers specify specific requirements in terms of goal model, including various RELAX operators for goals that face uncertainty factors. During this step, developers must also define utility value thresholds for goals that convert fuzzy logic utility function values to Boolean utility function values.

Figure 7.5 shows a modified version of the KAOS goal model from Figure 7.3 where RELAX operators are used in goals *G20*, *G21*, and *G22*. In the new goal model, *G20* has been modified with the RELAX language to allow partial goal satisfaction. The fuzzy logic utility functions of the RELAX-ed goal models are modified to return a real number ranging from 0 to 1 to represent the degree of satisficement for the specific goal. Specifically, *G20* is considered satisfied if the threshold value of both the tire pressure sensor monitor and the friction monitor are satisfied to the degree of 0.5.

To demonstrate the use of RELAX with an autonomous rover, consider goal *G20* in Figure 7.5,

where the rover ensures that there is sufficient braking power for the rover to stop should it detect a potential collision. Prior to being RELAX-ed, *G20* returns a Boolean value to parent goals indicating whether there is enough braking power or not. *G21* and *G22* return Boolean values depending on whether or not the specific sensor values are satisfied. In order to add flexibility and account for environmental uncertainties, *G21* and *G22* are RELAX-ed to account for uncertainty (e.g., if the visibility is poor and the rover is traveling under 10 m/s). To check if *G20* is satisfied, the system ensures that i) the friction sensor detects a friction rate of 2 Newtons, with an acceptable range of −0.5 Newtons and ii) the tire pressure is within 221 kPa (kilopascals), with an acceptable range of ±14 kPa. Fuzzy logic is used to express the degree of *satisficement* in the RELAX utility functions. For example, if the system detects that the value of a RELAX-ed goal is satisfied (e.g., tire pressure is at 221 kPa), then the corresponding utility function returns 1. The value returned reduces to 0 as the value reaches the lower-bound of the acceptable range.

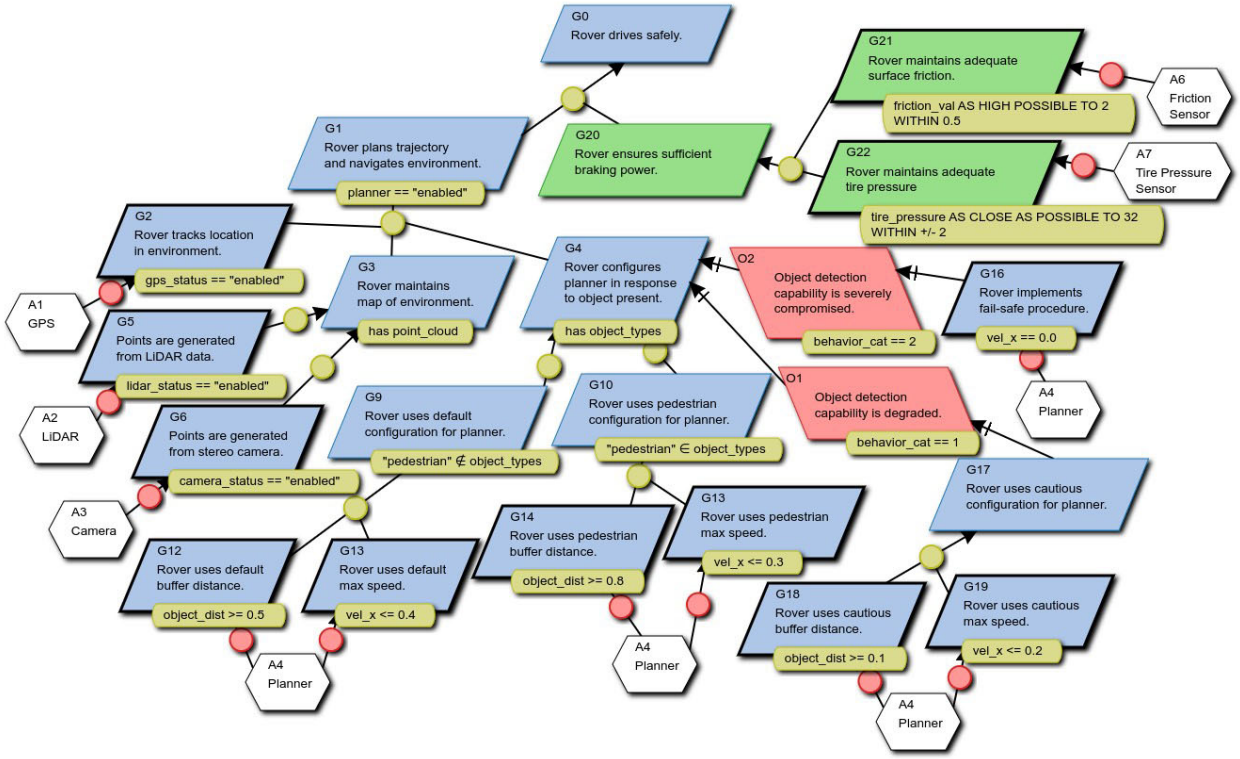


Figure 7.5: A RELAX-ed version of the KAOS goal model shown in Figure 7.3. Goals *G20*, *G21*, and *G22* (shown as green) have been RELAX-ed and use fuzzy logic to express a degree of goal satisficement.

Figure 7.6 shows an example of the range of values returned for *G21*. The utility function used to evaluate *G21* is shown in Figure 7.7. *G21* returns 1 if the value of the friction sensor reads 2 Newtons. The returned value linearly decreases as the sensor value reduces to the lower-bound of the acceptable range. In contrast, Figure 7.8 shows an example of the range of values returned in *G22*. Unlike *G21* where the goal has an acceptable range below a set value, *G22* allows for satisficement in both directions (i.e., a triangular fuzzy logic function is used). The utility function used to derive the return value for *G22* is shown in Figure 7.9. It returns 1 when the tire pressure monitor reads 221 kPa. Since the acceptable range of the utility function is defined as 221 ± 14, the value returned by the utility function decreases linearly to 0 as it approaches 207 kPa or 235 kPa, forming a triangular relationship.

To obtain the return value of *G20*'s utility function, the system evaluates the values of *G21* and *G22*'s refinement and compares it to a threshold defined by the domain expert. Figure 7.5 shows the relationship between *G20*, *G21*, and *G22*, where the sub-goals form an *AND* relationship. The utility function used to evaluate *G20*'s value is defined as:

$$utility(G20) = \begin{cases} 1.0 & \text{if } \min(utility(G21), utility(G22)) > \text{threshold} \\ 0.0 & \text{otherwise} \end{cases}$$

While this chapter demonstrates the use of RELAX to explicitly specify uncertainty, MoDALAS can also accommodate other types of requirement specification languages and corresponding utility functions used to address uncertainty, such as FLAGS [8], probabilistic utility functions, etc.

7.3 Goal Model-Driven Self-Adaptation of Learning-Enabled Systems

This section describes the proposed MoDALAS framework for goal-based self-adaptation of an LES. Figure 7.10 shows a DFD of the framework. Circles represent computational steps, boxes represent external entities, directed arrows show the flow of data, and persistent data stores are shown within parallel lines. Design-time steps (green) include the construction of an assurance case, a goal-oriented requirements model of the LES, and a behavior oracle for each LEC. Run-time steps (blue) implement a MAPE-K feedback loop driven by the models constructed at design time.

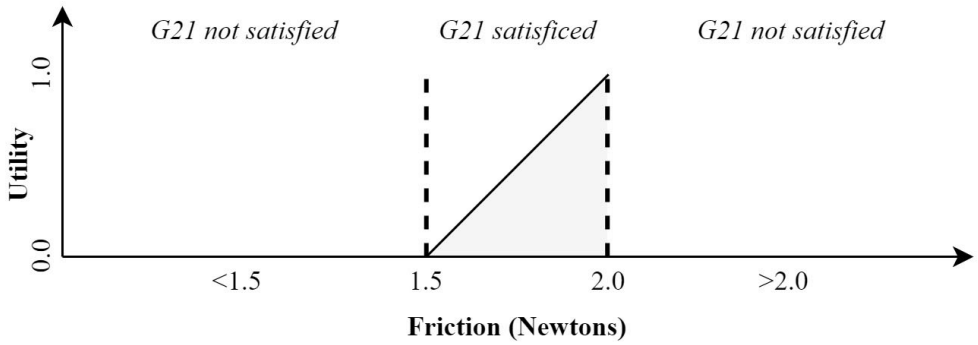


Figure 7.6: Range of values returned by the utility function for evaluating the friction sensor (*G21* in Figure 7.5) using the RELAX language with fuzzy logic.

```
function FRICTION-SENSOR-UTILITY(measured, desired=2, bounds=0.5)
    if (measured ≥ bounds) then
        return 1.0
    else if (measured ≤ desired − bounds) then
        return 0.0
    else
        return (desired − bounds − measured)/bounds
    return archive
```

Figure 7.7: Utility function for evaluating friction sensor satisficement (*G21* in Figure 7.5).

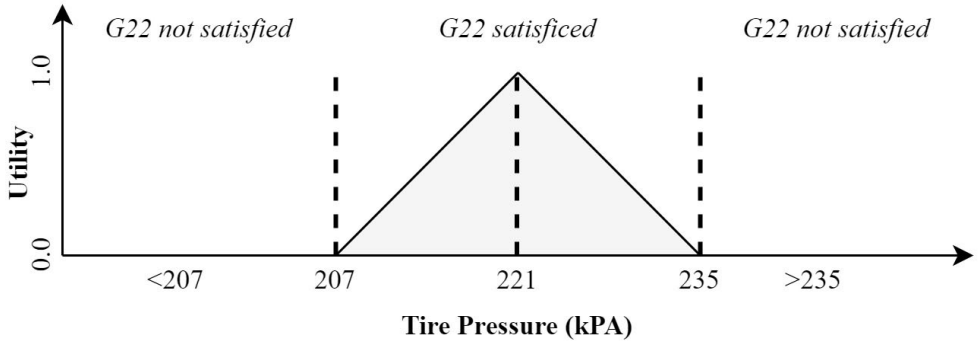


Figure 7.8: Range of values returned by the utility function for evaluating the tire pressure monitor system (*G22* in Figure 7.5) using the RELAX language with fuzzy logic.

```
function TIRE-PRESSURE-MONITOR-UTILITY(measured, desired=221, bounds=14)
    if ((measured ≤ desired−bounds) ∨ (measured ≥ desired+bounds)) then
        return 0.0
    else if (measured ≤ desired) then
        return (desired − bounds − measured)/bounds
    else
        return (desired + bounds − measured)/bounds
    return archive
```

Figure 7.9: Utility function for evaluating the tire pressure monitor satisficement (*G22* in Figure 7.5).

Although MoDALAS is platform-independent, to aid the reader, the following descriptions include an example of an autonomous rover with a learning-enabled object detector. Specific implementation details on how MoDALAS is applied to the autonomous rover are provided in Section 7.4. Each step in Figure 7.10 is described in detail as follows.

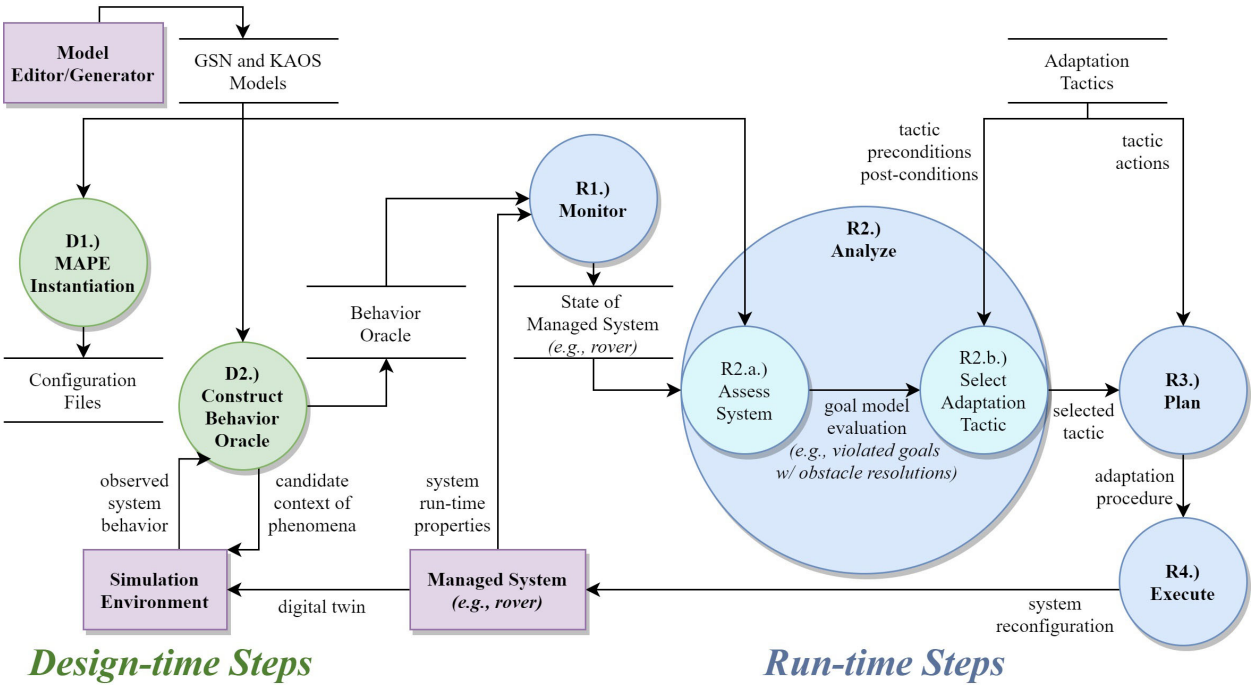


Figure 7.10: High-level data flow diagram of MoDALAS. Processes are shown as circles, external entities are shown as boxes, and persistent data stores are shown within parallel lines. Directed lines between entities show the flow of data.

7.3.1 Preparation at Design Time

To prepare the MoDALAS framework for execution, steps need to be taken to properly initialize the framework. Configuration files must be prepared to instantiate the autonomic manager (Step D1), and *behavior oracles* must be constructed prior to run time execution (Step D2).

Step D1. MAPE Instantiation

An autonomic manager (implemented as a MAPE-K loop) is instantiated to manage adaptations of the LES. To determine the system state and evaluate KAOS goal models at run time, the

autonomic manager must be configured to monitor the same system attributes referenced by KAOS goal models. KAOS goal models are parsed, and utility functions are extracted from each KAOS element. MoDALAS requires that KAOS goal models have been converted into a machine parsable file format (e.g., XML) that includes attributes for each KAOS element and its associated utility function. A set of *utility parameters* is then compiled by identifying each unique variable referenced by a utility function. Since utility parameters may refer to abstract concepts, a manual mapping must be made by the user to link each utility parameter to a platform-specific property of the LES. For example, for the utility function `object_dist >= 0.8` in Figure 7.3, goal *G14*, the utility parameter `object_dist` refers to the buffer distance between the rover and any object in the environment. It is the responsibility of the autonomic manager to link this abstract parameter to a real, platform-specific property of the rover. Configuration files are generated by Step D1 to initialize the monitor processes of the MAPE-K loop with references to the platform-specific properties to observe.

Step D2. Constructing Behavior Oracles

To monitor and assess the trustworthiness of LECs at run time, MoDALAS leverages *behavior oracles* generated by Enlil [127] for each individual LEC. Behavior oracles are implemented as DNNs to infer behavior of each LEC when exposed to new forms of environmental uncertainty under simulation. For example, when a rover implements a learning-enabled object detector that has been trained only in clear weather, Enlil can be used to simulate adverse weather conditions and model the capability of the object detector under a variety of adverse conditions. The resulting behavior oracle can then predict different *behavior categories* for the object detector when presented with sensor data under various weather conditions. These categories are application-specific and must be defined according to the user for the given task and LECs involved. In essence, behavior oracles serve as a utility function for assessing the type of LEC behavior to expect under varying run-time conditions.

The KAOS goal model in Figure 7.3 reflects that three different behavior categories have been specified for the behavior oracle of a rover's object detector. Two of these (`behavior_cat == 1`

and `behavior_cat == 2`) are attached to obstacles *O1* and *O2*, respectively. The third is the default and not explicitly shown in Figure 7.3 (`behavior_cat == 0`) . (The number of behavior categories depends on the granularity and spectrum of available behaviors and also the number of alternative resolutions required to satisfy system requirements.) Categories are determined by assessing the object detector's performance under a variety of adverse environmental contexts in simulation. In this example, the object detector's *recall* is measured when a newly-introduced adverse condition is present versus when it is not. The change in recall (δ_{recall}) is then used to measure the effect on object detector's performance. The value of δ_{recall} is computed statistically by measuring the object detector's recall for a set of validation images with and without exposure to the given environmental phenomena. Table 7.2 provides a description of each behavior category reflected in Figure 7.3. When $\delta_{recall} < 5\%$, the given context is considered to have *"little impact"* on object detection (*Category 0*). When $5\% <= \delta_{recall} < 10\%$, the object detector is considered *"degraded"* (*Category 1*). Finally, when $\delta_{recall} > 10\%$, the object detector is considered *"compromised"* (*Category 2*).

Table 7.2: Behavior categories for an object detector.

Category	Classification	Definition
0	*"little impact"*	$0\% \leq \delta_{recall} < 5\%$
1	*"degraded"*	$5\% \leq \delta_{recall} < 10\%$
2	*"compromised"*	$10\% \leq \delta_{recall}$

ENLIL automatically assesses an LEC by generating unique contexts of simulated environmental phenomena (via *evolutionary computation* [52]) to uncover examples that lead to each targeted behavior category. Figure 7.11 shows a scatter plot generated by ENLIL when simulating dust clouds. Each point represents a different dust cloud context with the resulting recall for the object detector LEC. Colors correspond to the observed behavior category for each respective point (i.e., categories 0, 1, and 2 are *green*, *yellow*, and *red*, respectively). Data collected during this assessment phase is used by ENLIL to train a behavior oracle that can map LEC inputs to expected behavior categories (i.e., *model inference*).

Behavior oracles created in Step D2 are used at run time to predict the resulting behavior category of an LEC for any given run-time context. For the example object detector, inputs to the behavior oracle are the same camera inputs given to the object detector. Output from the behavior oracle includes a description of the *perceived context* of the environmental condition and the *inferred behavior category* for the object detector. Figure 7.12 shows three behavior categories to represent different degrees of impact dust clouds can have on an object detector. Effectively, this information is used to assess the *trustworthiness* of the object detector.

7.3.2 Self-Adaptation at Run Time

A MAPE-K loop autonomic manager is executed at run time to monitor and reconfigure the managed LES with respect to the models constructed at design time. Responsibilities include assessing the current state of the LES, predicting the capability of LECs via behavior oracles, determining when system requirements are not satisfied by referencing KAOS goal models, and planning adaptations to ensure mitigating actions are taken to maintain requirements satisfaction.

Step R1. Monitor

In order to inform self-adaptations, *monitor* processes observe and record relevant attributes of the managed LES, which includes executing behavior oracles from Step D2. In KAOS notation, *agents* indicate which system components are responsible for each system requirement (e.g., *A1-A4* in Figure 7.3). Specific attributes of a system component are monitored when referenced by *utility functions* in the models constructed at design time (Step D1). Monitor processes are responsible for observing functional system components (e.g., controllers, mechanical parts, physical sensors, etc.) as well as behavior oracles for LECs. For example, when using a behavior oracle for a camera-based object detector, the behavior oracle is executed for each new camera input to predict the impact of run-time conditions on object detector performance. Through the use of utility functions, MoDALAS enables LECs to be monitored in a similar manner to traditional system components, using behavior oracles to determine whether or not LECs can be trusted in their run-time state.

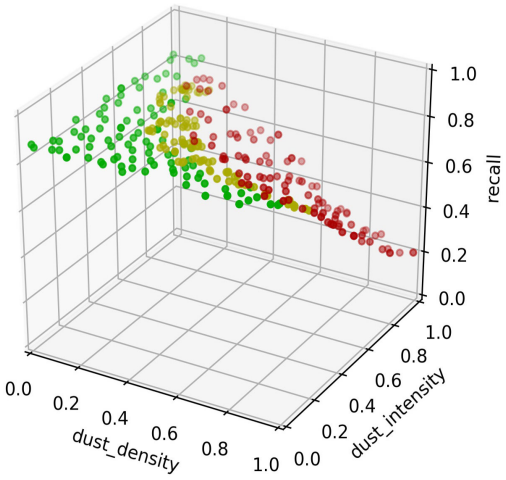
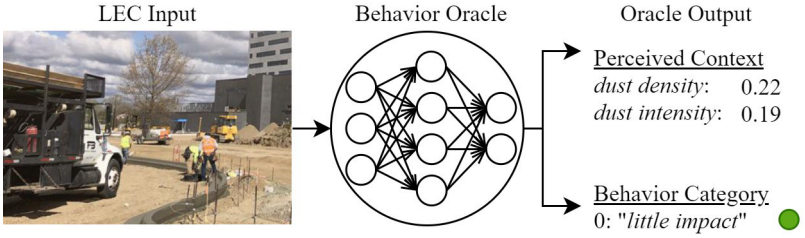
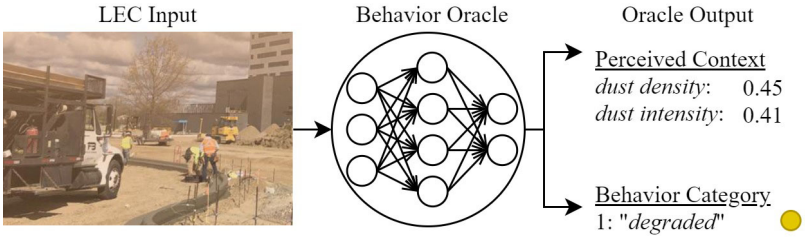


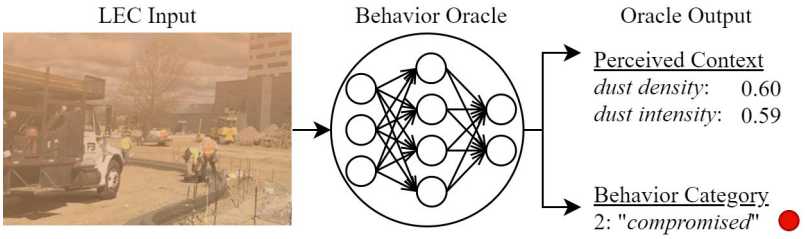
Figure 7.11: Scatter plot of object detector recall when exposed to simulated dust clouds from ENLIL. Each point represents a different dust cloud context with the corresponding *density* and *intensity*. Green, yellow, and red points correspond to behavior categories 0, 1, and 2, respectively.



(a) "light" dust cloud resulting in a *category 0* classification



(b) "medium" dust cloud resulting in a *category 1* classification



(c) "heavy" dust cloud resulting in a *category 2* classification

Figure 7.12: Example behavior oracle input/output for an image-based object detector LEC. Input is identical to the input given to the LEC. Output is a "*perceived context*" to describe the environmental condition and a "*behavior category*" to describe the expected LEC behavior. Examples of behavior categories 0, 1, and 2 are shown in (a), (b), and (c), respectively.

Step R2. Analyze

KAOS goal models of the LES are evaluated in Step R2.a to determine if adaptation is needed to resolve violated system requirements. Utility functions from the KAOS goal model are extracted, and a logical expression is formed via top-down graph traversal of the KAOS goal model. For example, Figure 7.13 shows the logical expression parsed from the KAOS goal model in Figure 7.3. Variables in this expression are substituted with corresponding values recorded by Step R1, and the entire expression is evaluated to determine satisfaction of the KAOS goal model. If the logical expression is satisfied, then no adaptation is needed. However, in the event that the resulting evaluation is unsatisfied, then the type of adaptation is determined based on the set of violated utility functions.

```
planner_status == “enabled”
AND gps_status == “enabled”
AND has point_cloud
|   AND lidar_status == “enabled”
|   |   OR camera_status == “enabled”
AND has object_types
|   AND
|   |   AND “pedestrian” ∈ object_types
|   |   |   AND object_dist ≥ 0.5
|   |   |   AND vel_x ≤ 0.4
|   |   OR “pedestrian” ∉ object_types
|   |   |   AND object_dist ≥ 0.8
|   |   |   AND vel_x ≤ 0.3
|   AND IF behavior_cat == 1
|   |   THEN object_dist ≥ 1.0
|   |   |   AND vel_x ≤ 0.2
|   AND IF behavior_cat == 2
|   |   THEN vel_x == 0
```

Figure 7.13: Logical expression parsed from KAOS model in Figure 7.3.

Methods for adaptation are implemented as *adaptation tactics* [37], which are stored in a repository accessible by the MAPE-K loop (example tactic in Figure 7.14). Each tactic comprises a *precondition*, *post-condition*, and set of *actions* to perform on the managed LES. Preconditions and postconditions for tactics reference the satisfaction of KAOS *goals/obstacles*, where preconditions are defined by the utility functions for KAOS obstacles and post-conditions are defined by the

utility functions associated with the resolution goals for KAOS obstacles. Step R2.b retrieves a tactic from the repository with preconditions that most closely match (e.g., via logical implication) the current evaluation of the KAOS goal model. For example, in the event that *O1* is satisfied and goal *G17* is not satisfied, the tactic in Figure 7.14 with a precondition matching the utility function for *O1* is selected. The post-condition in Figure 7.14 includes the utility functions for *G17* and its sub-goals (*G18* and *G19*). The actions associated with the tactic are platform-independent operations required to satisfy the post-condition. When multiple tactics fit the given preconditions, the tactic with *higher priority* is chosen, where priorities can be manually assigned and/or adjusted based on the success of subsequent goal-model evaluations in future iterations of the MAPE-K loop.

tactic "Reconfigure to Cautious Mode"
- **precondition:** (*KAOS O1*)
  - `behavior_cat == 1`
- **actions:**
  - `set object_dist ← 1.0`
  - `set vel_x ← 0.2`
- **post-condition:** (*KAOS G17* ∧ *G18* ∧ *G19*)
  - `object_dist ≥ 1.0`
  - `AND vel_x ≤ 0.2`

Figure 7.14: Example tactic for reconfiguring a rover to a "cautious mode". *Preconditions* and *post-conditions* refer to KAOS elements and utility functions (see Figure 7.3). *Actions* are abstract operations to achieve the post-condition.

Step R3. Plan

After a platform-independent adaptation tactic has been selected in Step R2, a platform-specific procedure is generated for implementing the *actions* associated with the selected tactic. For example, a platform-independent action to turn the autonomous platform 15° will be translated into the corresponding operations for a wheeled rover versus a legged-robot. Additionally, actions may be modified to consider the dynamic state of the system during execution of a tactic (e.g., actions may change or be preempted to account for emergent mechanical issues in a rover) [169].

Step R4. Execute

After an adaptation procedure has been planned, Step R4 is responsible for interfacing with and reconfiguring the managed LES. Depending on the nature of the adaptation and the current system state, different methods of adaptation may be considered to ensure the managed LES functions correctly while safely transitioning into its new configuration (e.g., *one-point*, *guided*, or *overlap* adaptations) [233]. Since adaptations may not be safe to perform in all states of the LES, Step R4 is responsible for determining *quiescent states* where the LES can be safely reconfigured (e.g., prevent halting a rover during a high-speed turn) [117].

7.4 Demonstration With Autonomous Rover

To illustrate the operation of MoDALAS, we consider a scenario where an EvoRally autonomous rover (see Section 2.5) is used within a construction site. Compared to autonomous automobiles operated on public roads, autonomous construction vehicles operate within relatively tight behavioral constraints and physical areas, leading to rapid growth in this market segment [149]. In addition to large earth-moving vehicles, smaller rovers are used to carry tools and materials for construction workers, periodically record the progress of construction, and provide surveillance of the site past operation hours [146]. For such rovers, detecting and avoiding objects in the environment, including pedestrians and other vehicles, are safety-critical requirements [217]. Increasingly, machine learning techniques have been used to provide object detection in such applications [49]. However, ensuring requirements satisfaction of learning-enabled autonomous rovers is a challenging task, as transient environmental conditions (i.e., rainfall or dust clouds) can impede object detection and potentially lead to serious accidents and even fatalities. To demonstrate the operation of MoDALAS in the construction site application domain, we have implemented a prototype and integrated it into the software for an autonomous robot in our laboratory.

The EvoRally rover's software infrastructure is based on ROS [176], a popular middleware platform for robotics. A ROS implementation comprises a set of processes, called ROS *nodes*, that communicate with other ROS processes using a publish-subscribe mechanism called ROS *topics*.

Multiple ROS nodes can publish messages on a ROS topic, and multiple ROS nodes can subscribe to the same ROS topic. Commonly, and in our case, ROS is implemented atop the Linux operating system with ROS nodes realized as Linux processes. For a non-trivial robot such as our rover, this design produces an intricate software infrastructure that can be visualized with a ROS *graph*. The full ROS graph for our rover software comprises over 30 nodes that implement tasks such as processing of sensor data, localization, path planning, and generating the corresponding commands to control the vehicle. Over 200 ROS topics are used to convey raw and preprocessed sensor data, exchange of information among controller nodes, and delivers commands to actuators for throttle control, steering and braking.

Figure 7.15 shows an (elided) ROS graph of the MAPE-K loop implemented for the rover. The `\knowledge` ROS node is a process that manages access to the data stores depicted in Figure 7.10. Data stores for goal models and adaptation tactics are populated at startup time and remain static during operation. However, the *managed system state* data store is highly dynamic, comprising sensor readings and other state information that are updated continually. The MAPE-K *monitor* step (Step R1 in Figure 7.10) is implemented as a collection of ROS nodes (e.g., `\monitor_lidar`, `\monitor_wheels`, `\monitor_camera`) that receive raw sensor data collected by hardware-specific ROS nodes. These nodes preprocess data streams and publish results to the `\update_state` topic in order to modify the managed system state. Examples include direct measurements (e.g., wheel speed), derivative measurements (e.g., rate of battery drain), and operational status of hardware components (e.g., delays in GPS localization reporting). The remaining three MAPE-K steps (Steps R2-R4) are implemented as singleton ROS nodes, respectively, `\analyze`, `\plan`, and `\execute`.

KAOS goal model evaluation by the `\analyze` node is triggered by state changes published on the `\state_change` ROS topic. If the KAOS goal model is not satisfied and an adaptation is necessary, then the `\analyze` node determines the type of adaptation needed and relays the adaptation type to the `\plan` node via the `\plan_action` topic. The `\plan` node retrieves actions for the corresponding tactic from the knowledge base and forwards an adaptation procedure to the

\execute node. The \execute node directly interfaces with and reconfigures components of the target platform.

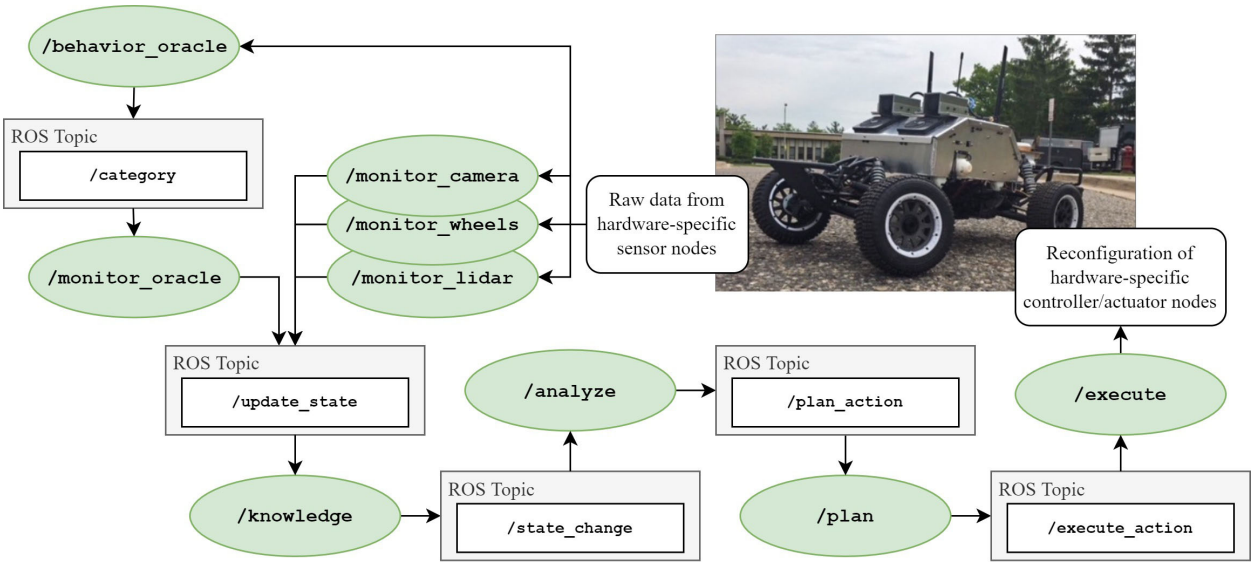


Figure 7.15: Elided ROS graph for MAPE-K loop in rover software. ROS nodes shown as green ellipses and ROS topics as yellow boxes. Arrows indicate data flow.

7.4.1 Assessing Visual Inputs With Behavior Oracles

In our proof-of-concept demonstration, we use images from the mounted cameras atop the rover for *object detection* [100, 141] and *triangulation* from stereo vision [83]. A three-dimensional *point cloud* [188] is generated by fusing stereo camera triangulations and lidar sensor readings. As shown in Figure 7.15, both the \monitor_camera and \behavior_oracle nodes receive raw camera data published from onboard cameras. The \monitor_camera node processes camera data and delivers relevant information (e.g., frame rate, etc.) to the \knowledge node. For example, lack of input or a slow frame rate might indicate a problem with one or both cameras, thus necessitating an adaptation.

The \behavior_oracle node processes camera images *online* with the behavior oracle DNN that was trained *offline* by ENLIL for model inference. (The DNN is initialized at startup time from a configuration file.) Specifically, the \behavior_oracle node infers the behavior of the onboard object detector LEC by evaluating input images given to the object detector. The behavior

category produced by the `\behavior_oracle` node is published on the `\category` ROS topic, which is monitored by the `\monitor_oracle` node. At run time, if the `\monitor_oracle` node reports any change in the behavior category, then the `\analyze` node will execute to address the situation, as follows.

7.4.2 Run-Time Adaptation

We consider a scenario in which the behavior oracle triggers run-time adaptations to the rover. At design time, the behavior oracle was created to account for three types of adverse environmental conditions that can impact object detection: *rainfall*, *dust clouds*, and *lens flares* (where a bright light source obscures part of the image). Additional environmental phenomena can be included by introducing them into the simulation environment used by ENLIL. Figure 7.16 shows examples of each simulated phenomenon, with different levels of intensity, and the resulting object detector behavior category inferred by the behavior oracle. Referencing the behavior categories in Table 7.2, examples in columns (i), (ii), and (iii) are expected result in *Categories 0*, *1*, and *2*, respectively (i.e., "*little impact*," "*degraded*," and "*compromised*").

Scenario 1. Dust Clouds

To demonstrate MoDALAS in practice, we explore a scenario where the autonomous rover navigates a construction site to periodically record progress on the project at different locations. The rover begins with `behavior_category = 0`. As the rover approaches a construction worker, a dust cloud is produced by a dump truck leaving the construction area. When the `\behavior_oracle` node receives images from the rover's cameras, the dust-obscured images are evaluated by the behavior oracle DNN, which infers that the object detector will be *degraded* by the current environment. Thus, the `\behavior_oracle` node publishes `behavior_category = 1` on the `\category` topic. The `\monitor_oracle` node forwards this change to the `\knowledge` node. The state change triggers execution of the `\analyze` node to evaluate the logical expression (Figure 7.13) of the KAOS goal model depicted in Figure 7.3. Upon evaluation, the `\analyze` node determines that adaptation is

(a) Unaltered Image of Construction Site

i. “little impact” ii. “degraded” iii. “compromised”

(b) Simulated Rainfall

i. “little impact” ii. “degraded” iii. “compromised”

(c) Simulated Dust Cloud (Haze)

i. “little impact” ii. “degraded” iii. “compromised”

(d) Simulated Lens Flare

Figure 7.16: Example of object detection at a construction site. A pedestrian is detected by an image-based object detector (a). New environmental phenomena are introduced in simulation, such as (b) rainfall, (c) dust clouds, and (d) lens flares. ENLIL explores different contexts to find examples that have (i) little impact, (ii) degrade, or (iii) compromise the object detector’s ability to achieve validated design-time performance.

necessary, since the pre-condition associated with KAOS obstacle *O1* applies but the resolving goal *G17* is not satisfied. The tactic in Figure 7.14 is selected and forwarded to `\plan`, which finds that the tactic's actions involve reducing the maximum velocity of the rover and increasing the buffer distance between the rover and objects in the environment. The `\plan` node then maps abstract tactic actions to a platform-specific procedure. Our rover uses a Timed Elastic Band (TEB) [183] planner provided with ROS to compute trajectories around objects in the environment. The abstract actions in Figure 7.14 can be accomplished by setting the `\min_obstacle_dist` and `\max_vel_x` parameters of the TEB planner. Finally, the platform-specific procedure is forwarded to the `\execute` node, which is responsible for executing the reconfiguration of the rover. As a result, the rover moves slower and takes a wider berth around objects in the environment while the dust cloud is present.

Eventually, as the dust settles, the behavior oracle determines that the new environmental condition is expected to have *little impact* on the object detector (i.e., *Category 0*). Through the same sequence of steps described above, the `\analyze` node is triggered to execute by the state change. The `\analyze` node then determines that KAOS obstacle *O1* no longer applies and the KAOS goal model is satisfied. The `\analyze` node publishes a message to notify the `\plan` node that the selected tactic is no longer applicable. The `\plan` node then triggers the `\execute` node so that the previous operating parameters are restored (i.e., reset the minimum object distance and maximum rover velocity to their prior values).

Scenario 2. Lens Flare

In a second scenario, the rover is navigating around a parked vehicle. Suppose the reflection of the sun on the windshield of the vehicle causes a momentary lens flare that blinds the cameras. The behavior oracle processes the camera image and determines that the impacted images will *compromise* the ability of the rover's object detector to perform as validated at design time (i.e., *Category 2*). The `\monitor_oracle` node publishes `behavior_category = 2` to `\knowledge`, triggering the `\analyze` node similar to the dust cloud scenario. The KAOS goal model is evaluated,

but this time obstacle *O2* applies and its resolving goal *G16* is not satisfied. A tactic with a pre-condition matching *O2* and post-condition matching *G16* is selected and forwarded to the `\plan` node. The actions associated with the selected tactic are to halt the rover, and the `\plan` node generates a procedure to set the rover's maximum velocity to zero. Finally, the adaptation procedure is forwarded to the `\execute` node to update the rover accordingly, thereby transitioning it to a fail-safe state. When the lens flare eventually disappears (e.g., due to changing angle of the sun or cloud movements), the `\monitor_oracle` node publishes `behavior_category = 0`. The change in behavior category triggers the `\analyze` node to re-evaluate the KAOS goal model. The `\analyze` node then determines that *O2* no longer applies, subsequently triggering the `\plan` and `\execute` nodes to reset the selected tactic and restore the rover to its original configuration.

Scenario 3. Relaxation of Goals

In a third scenario, we explore how RELAX may be used to explicitly deal with uncertainty on the rover. Suppose the rover uses the KAOS goal model in Figure 7.3, where the KAOS goal model is initially not RELAX-ed to address run-time uncertainties. Figure 7.17 shows example utility values published by the rover during operation. Table 7.3 shows the resulting goal model evaluation to be *unsatisfied* since the original KAOS goal model expects a friction rate of 2 and a tire pressure of 221 kPa (individual goal results of *G21* (0.0) and *G22* (0.0)). In this instance, an unsatisfactory evaluation of the goal model would trigger MoDALAS to execute an adaptation tactic to mitigate brake failure (e.g., notifying a human supervisor for intervention). However, the rover may be able to operate under the given values as the deviation is insignificant (e.g., inaccurate readings due to sensor noise). Thus, if we use the RELAX-ed KAOS goal model in Figure 7.5, the system uncertainties may be tolerated and avoid the need for an immediate mitigation strategy that could negatively impact performance. Table 7.4 shows the resulting evaluation of the same rover configuration from Figure 7.17, but instead using the RELAX-ed goal model from Figure 7.5. Using fuzzy logic, the new model is tolerant to the sensor values with an accepted deviation range (individual goal results of *G21* (0.799) and *G22* (0.95)).

```
utility_params = {
    'behavior_cat':     0,
    'camera_status':    'enabled',
    'friction_val':     1.9,
    'gps_status':       'enabled',
    'lidar_status':     'enabled',
    'object_dist':      3.0,
    'object_types':
        ['pedestrian', 'car', 'car'],
    'planner':          'enabled',
    'point_cloud':      True,
    'tire_pressure':    31.9,
    'vel_x':            0.1,
}
```

Figure 7.17: Example set of utility parameter values for Scenario 3

Table 7.3: Example evaluation of non-RELAX-ed KAOS goal model (Figure 7.3)

Goal #	Evaluation Result
'G1'	1.0
'G2'	1.0
'G3'	1.0
'G4'	1.0
'G5'	1.0
'G6'	1.0
'G9'	0.0
'G10'	1.0
'G12'	1.0
'G13'	1.0
'G14'	1.0
'G16'	0.0
'G18'	1.0
'G19'	1.0
'G21'	0.0
'G22'	0.0
'O1'	0.0
'O2'	0.0
Total Evaluation	0.0
Total Satisficement	0.0

Table 7.4: Example evaluation of RELAX-ed KAOS goal model (Figure 7.5)

Goal #	Evaluation Result
'G1'	1.0
'G2'	1.0
'G3'	1.0
'G4'	1.0
'G5'	1.0
'G6'	1.0
'G9'	0.0
'G10'	1.0
'G12'	1.0
'G13'	1.0
'G14'	1.0
'G16'	0.0
'G18'	1.0
'G19'	1.0
'G21'	0.799
'G22'	0.95
'O1'	0.0
'O2'	0.0
Total Evaluation	0.799
Total Satisficement	1.0

7.5 Related Work

This chapter explores methods for the assurance of cyber-physical LESs via models at run time [43]. Related studies have investigated the verification of robotic systems [90], including construction site applications [77]. Those works apply formal methods for verification but do not explicitly address LECs faced with uncertain conditions. RoCS [179] has been introduced as a self-adaptive framework for robotic systems, but in contrast to MoDALAS, it is not model-driven nor focused on software assurance. To the best of our knowledge, MoDALAS is the first to include run-time assessment of LECs with respect to goal-oriented (i.e., KAOS) models.

Self-adaptive frameworks have used different approaches to address assurance. Zhang and Cheng [233] developed a state-based modeling approach for model checking assurance properties of SASs. Weyns and Iftikhar [216] proposed the use of model-based simulation to evaluate system requirements and determine adaptation procedures. ENTRUST [29] supports the development of an SAS driven by GSN assurance cases and verified by probabilistic models at run time. Similarly, AC-ROS (described in Chapter 6) is a GSN model-driven self-adaptive framework for ROS-based applications, which includes self-assessment through utility functions as assurance evidence. In contrast, MoDALAS uses KAOS goal models to assess the satisfaction of system requirements of an LES at run time. Furthermore, these other approaches do not address uncertainty for LECs. MoDALAS enables an LES to self-adapt to mitigate failure from the use of LECs in untrusted contexts.

A number of design-time approaches have addressed how LECs handle uncertainty [175, 182]. Smith *et al.* [201] have also explored the construction of assurance cases at design time to categorize LEC behavior with respect to hazardous behaviors. However, these methods do not enable self-assessments at run time and have limited applications for handling uncertain environmental conditions. MoDALAS differs from these works by using model inference (via *behavior oracles*) to assess LEC behavior at run time for *known unknown* environmental conditions.

Another direction in which research has addressed environmental uncertainties for LECs is during requirements modeling and specification. Whittle *et al.* [221, 222] proposed RELAX as

a requirements specification language that allows for the relaxation of requirements to adapt to environmental uncertainties. Fredericks *et al.* [63] and Ramirez *et al.* [177] proposed automation of relaxation of goal models and derivation of utility functions using RELAX. Letier *et al.* [135], Ramirez *et al.* [177], and Bencomo *et al.* [14] proposed various utility functions (e.g., probabilistic) used to evaluate and quantify partial satisfaction of a goal. Letier *et al.* [136] also proposed using Monte Carlo simulation to calculate the consequences of certain uncertainty factors. The MoDALAS framework has been designed to accommodate different requirements specification languages and the corresponding analysis techniques to address uncertainty.

Recently, other researchers have explored system assurance for LESs and LECs. Asaadi *et al.* [5] proposed a probabilistic quantification of LES system confidence based on functional capabilities and dependability attributes. Boursinos *et al.* [21] proposed a conformal prediction framework, leveraging previous normal values to check for abnormalities. Weyns *et al.* [220] proposed combining MAPE, control theory, and machine learning for better adaptive systems. In contrast, MoDALAS focuses on self-adaptation to mitigate LEC failure through the use of *behavior oracles* and the run-time evaluation of KAOS goal models with optional RELAX-ed goals.

7.6 Summary

This chapter introduced the MoDALAS framework for using requirements models at run time to automatically address the assurance of safety-critical systems with machine learning components. Due to uncertainties about the ability for LECs to generalize to complex environments, methods are needed to assess the capability of LECs at run time and adapt LESs to mitigate the use of LECs in unsafe run-time conditions. MoDALAS assesses the trustworthiness of LECs with *behavior oracles* and reconfigures an LES to maintain satisfaction of system requirements at run time.

MoDALAS addresses uncertainties about the assurance of an autonomous LES when facing uncertain run-time conditions (e.g., *known unknown* phenomena). This chapter described a proof-of-concept prototype of MoDALAS for an autonomous rover LES with an object detector LEC. MoDALAS adapts the rover to maintain safety requirements under run-time conditions where the

object detector is deemed unreliable. This chapter has also demonstrated how MoDALAS can leverage the RELAX language and fuzzy logic run-time evaluation to manage uncertainties in requirements. Chapter 8 explores in detail how to apply model-driven self-adaptation into a single service-oriented framework to develop more trustworthy LESs, validated on an autonomous rover with vision-based deep learning components.

CHAPTER 8

ADDRESSING ROBUSTNESS AND RESILIENCY AT RUN TIME

Trustworthy artificial intelligence (Trusted AI) is of utmost importance when LECs are used in autonomous, safety-critical systems. When reliant on deep learning, these systems need to address the reliability, robustness, and interpretability of learning models. In addition to developing specific strategies to address each of these concerns, appropriate software architectures are needed to coordinate LECs and ensure they deliver acceptable behavior under uncertain conditions. This work proposes a model-driven framework of loosely-coupled modular services designed to monitor and control LECs with respect to Trusted AI assurance concerns. The proposed framework is composable, deploying independent services to improve the resilience and robustness of AI systems. The overarching objective of this framework is to support software engineering principles such as modularity, composability, and reusability in order to facilitate development and maintenance tasks, while also increasing stakeholder confidence in Trusted AI systems. To demonstrate this framework, it has been implemented to manage the operation of an autonomous rover's vision-based LEC under uncertain environmental conditions.

The remainder of this chapter is organized as follows. Section 8.1 overviews and motivates this study. Section 8.2 describes the aggregate elements of a proposed service-oriented architecture for Trusted AI. Section 8.3 demonstrates a use case of the proposed framework on an autonomous rover platform. Finally, Section 8.5 provides a concluding summary for this chapter.

8.1 Overview

A critical factor for the use of AI in *safety-critical* tasks is the need for stakeholders to *trust* AI systems to perform as intended, despite many uncertainties unique to LECs [224]. Data-driven LECs, such as DNNs [73], are often black box and more complex than traditional software components, and their use can require a "leap of faith" from stakeholders [112, 228]. However, the risk of using inadequate AI in safety-critical applications is severe, possibly leading to human

injury and casualties (e.g., autonomous driving accidents [164, 165]). Various high-level "Trusted AI" guidelines have been proposed to systematically address AI assurance concerns [92, 94, 151, 161, 190, 200]. As "best practice" guidelines, these frameworks increase awareness of safety issues unique to AI systems by decomposing assurance topics into categories such as *reliability*, *fairness*, *robustness*, *interpretability*, and *uncertainty quantification*. However, because the specific techniques used to address each assurance category are highly application-dependent, it can be challenging to generalize solutions for multiple applications. As machine learning software matures from a largely academic, research-focused domain to mainstream software, we need to apply well-established *software engineering* practices for Trusted AI [31, 150]. This chapter proposes a modular and composable approach to develop Trusted AI in support of the different dimensions of uncertainty recognized by existing guidelines [92, 94, 151, 161, 190, 200].

Current solutions to Trusted AI concerns, such as *adversarial robustness* and *adversarial detection* [142], are tightly-coupled to specific problem domains, leading to *monolithic* applications that are difficult to scale, reuse, and maintain [11, 64]. When "robustifying" DNNs, techniques have been proposed to augment training data, training procedures, or network topologies, with updates interwoven into a single, monolithic learning model [198]. When proposed solutions are tightly-coupled to a base learning model, it is challenging to repurpose them for alternative learning models. Furthermore, when addressing uncertainty for Trusted AI systems, many context-dependent solutions are needed to mitigate the various forms of uncertainty (e.g., robustness to malicious weather effects versus cybersecurity concerns). When solutions are monolithic, any change with respect to a single form of uncertainty can require the entire learning model to be retrained and validated. As new forms of adversarial conditions are uncovered, monolithic solutions require extensive updates to the entire learning model, rather than isolating changes to decomposable functional units. For systems that address multi-dimensional problems, such as Trusted AI, alternative software development and assessment practices should be considered, as current monolithic approaches are difficult to scale and maintain [154].

This chapter proposes a modular, composable approach to address multiple dimensions of

uncertainty in Trusted AI. Rather than using monolithic solutions to address all issues of uncertainty for Trusted AI, this chapter proposes the use of a framework comprising loosely-coupled services each of which are responsible for individual assurance concerns. In contrast to monolithic architectures, *microservice* [1] architectures realize software as a collection of independently deployable services that interact using a common Application Programming Interface (API) and communication protocol (e.g., TCP/IP) [6, 55]. Microservice architectures are ideal for systems that are both *goal-oriented* and focused on *replaceability* [155]. Microservice architectures facilitate a wide range of reusable code, because each microservice can be implemented with different technologies and programming languages. Furthermore, individual microservices can be executed on separate hardware platforms, enabling more scalable deployments. When realizing Trusted AI systems as microservice architectures, separate *reusable* services can be deployed to manage the reliability, robustness, and interpretability of an underlying LEC, rather than incorporating the same functionality into a single component.

This chapter introduces the ANUNNAKI[2] framework as a collection of microservices to facilitate the development of Trusted AI. While other works have considered AI as a microservice, this chapter is the first to explore Trusted AI as an aggregation of microservices that addresses multiple dimensions of uncertainty [154, 159, 172, 189]. The ANUNNAKI framework leverages existing techniques, such as ENLIL [127] to automatically assess the resiliency of LECs under a variety of adverse phenomena (e.g., rain, fog, etc.) and ENKI [129] to automatically generate more robust learning model alternatives, trained with synthetically augmented data. Furthermore, this chapter introduces UTU[3] as a collection of model-driven services within the ANUNNAKI framework to monitor and control LECs at run time, with respect to existing assurance and requirements artifacts (e.g., GSN [3] assurance cases and *KAOS* [124] requirement models). In contrast to previous work on AC-ROS [34] and MoDALAS [130], UTU is not dependent on the internal functionality of the managed AI platform, which promotes reusability, portability, and extensibility for more flexible

[1] Microservices are small, autonomous services within a loosely-coupled service-oriented architecture [155].
[2] The Anunnaki are a group of ancient Sumerian deities [118].
[3] Utu is an ancient Sumerian deity responsible for enforcing divine justice [118].

run-time monitoring. In execution, ANUNNAKI microservices run in parallel and independent of managed LECs. As such, the ANUNNAKI framework enables developers to reuse common services to generate robust alternatives to LECs, detect when LECs have entered untrusted states, and mitigate the use of LECs in untrusted states.

To demonstrate the ANUNNAKI framework, it has been applied to an autonomous rover with a vision-based obstacle detector LEC. In its default form, the obstacle detector exhibits a reasonable degree of accuracy on known validation data. However, uncertainties arise when new forms of adverse phenomena are considered (e.g., changes in lighting, occluded visibility, etc.). This chapter demonstrates how aggregate services within the ANUNNAKI framework can be leveraged to reconfigure the rover and mitigate the use of its object detector under conditions deemed untrustworthy. Through the use of independent microservices to assess trustworthiness and enact change in system behavior, the ANUNNAKI framework is a non-monolithic, scalable, and maintainable approach to addressing uncertainty in Trusted AI systems.

8.2 A Service-Oriented Framework for Trusted AI

This section provides a high-level overview of the ANUNNAKI framework. The aggregate collection of microservices provided by the ANUNNAKI framework collectively serve to manage the operation of LECs in the presence of uncertain conditions and to mitigate faults resulting from their use in untrusted conditions. Figure 8.1 illustrates the major processes within the ANUNNAKI framework as a DFD, where processes are depicted as interconnected circles. Rectangles depict systems external to the ANUNNAKI framework. Labeled arrows show data flow between processes, and persistent data stores are shown within parallel lines. Each process shown in Figure 8.1 is a separate microservice executed in parallel and independent of the managed AI system. The remainder of this section describes each of these processes.

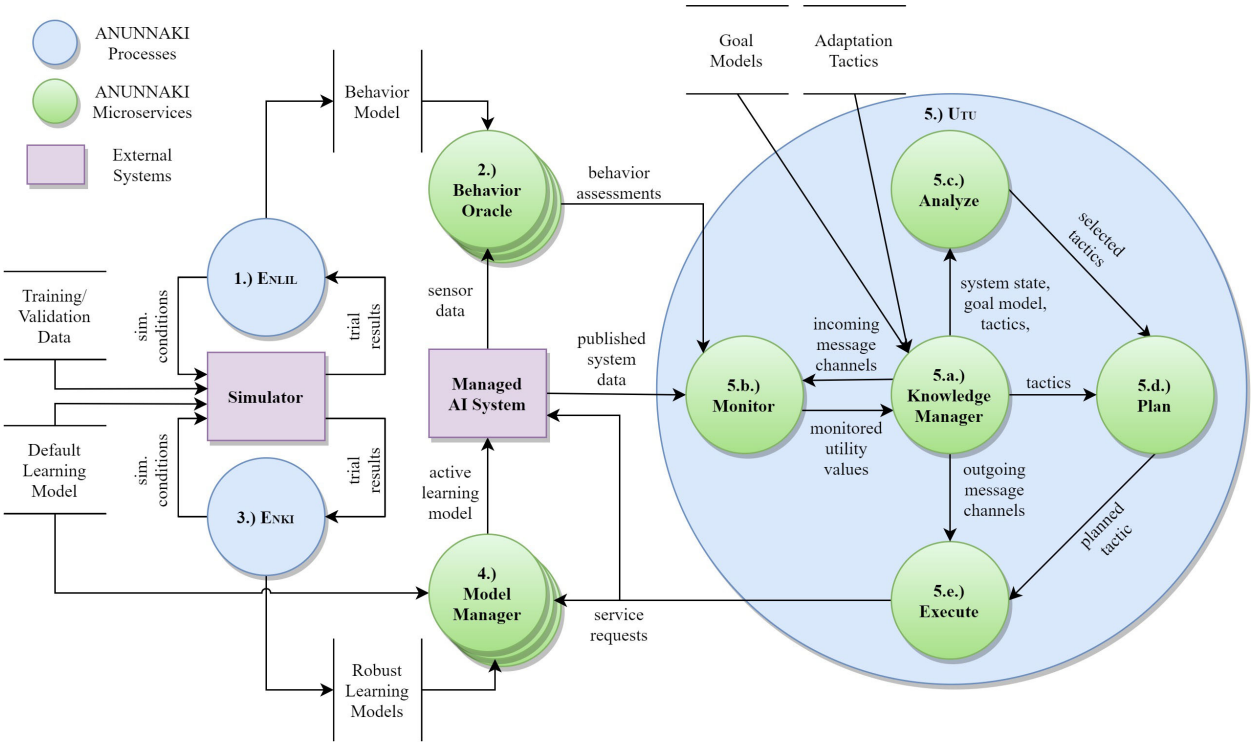


Figure 8.1: A high-level DFD of the ANUNNAKI framework. ANUNNAKI processes are shown as circles, interacting with external systems, such as the managed AI system and a simulator, shown as rectangles. Labeled arrows show data communicated between processes, with persistent data stores shown within parallel lines.

8.2.1 Resiliency Through Predictive Behavior

The ANUNNAKI framework leverages model inference and behavior models of an LEC for the purpose of *adversarial detection*. Adverse interference can include any malicious noise or environmental phenomena that result in undesirable behavior from an LEC. Behavior models are used to predict the impact of adverse conditions absent from existing training/validation data, thus enabling the ANUNNAKI framework to prevent the use of LECs under conditions they would normally perform unreliably (e.g., poor lighting conditions). ENLIL constructs behavior models of an LEC by assessing the impact of various environmental phenomena within an external simulator (Figure 8.1, Step 1) [127]. Next, ENLIL generates a behavior model that can be executed, independent of the LEC as a *behavior oracle* (Figure 8.1, Step 2). One or more behavior oracles run in parallel to the managed AI system and subscribe to the same sensor data received by managed LECs. As sensor data is received, behavior oracles output behavior assessments, which include both a *perceived con-*

text for any apparent adversarial noise and an *inferred behavior category* to summarize the impact of the adversarial noise. As microservices, behavior oracles publish behavior assessments to any other subscribing microservice, thus enabling the ANUNNAKI framework to assess the reliability of a managed AI system under a variety of *known unknown* environmental conditions and react to adverse run-time phenomena.

8.2.2 Robustifying Learning Models

To address the *adversarial robustness* of LECs, the ANUNNAKI framework uses alternate learning models trained using data synthetically augmented to include adverse phenomena (e.g., rain, fog, etc.). Using ENKI (Figure 8.1, Step 3), robust learning models are generated by running a simulator to uncover examples of adverse phenomena that lead to a diverse array of behavior patterns for the given LEC [130]. The selected adversarial examples are then used to retrain the default learning model. Once generated, alternative learning models can be swapped in place of the default learning model by a *model manager* microservice that determines which specific learning model is active at run time (Figure 8.1, Step 4). This microservice approach enables separate learning models to be robustified with respect to specific forms of adverse interference and swapped in place of each other based on run-time contexts. When no adverse interference is detected, the default learning model can be activated. By decoupling the problem of robustification from a single learning model to separate, independent learning models, the ANUNNAKI framework enables more flexibility to the developers on what forms of adverse phenomena are addressed by any given implementation of the managed AI system. Furthermore, this approach enables developers to maintain and augment specific context-dependent models without needing to retrain and validate the base learning model. For example, a robust learning model generated for rainy environments can be updated without needing to retrain/validate the default learning model. Furthermore, additional robustified models can be created for alternative phenomena (e.g., foggy weather, poor lighting, etc.) that are also independent from each other and the default learning model. Thus, the ANUNNAKI framework provides a more modular and composable solution to robustifying LECs.

8.2.3 Constructing Goal Models

The ANUNNAKI framework leverages goal models to capture the high-level objectives of the managed AI system. KAOS goal modeling [124] supports a hierarchical decomposition of high-level functional and performance objectives into leaf-level system requirements (i.e., goal-oriented requirements engineering [131]). KAOS goal models enable a formal goal-oriented analysis of how system requirements are interrelated as well as threats to requirement satisfaction. *Goals* represent atomic objectives of a system at varying levels of abstraction, with sub-goals refining and clarifying higher-level goals. Any event threatening the satisfaction of a specific goal is represented as an *obstacle*. Resolutions for obstacles can be specified by attaching additional sub-goals with alternative system requirements to the corresponding obstacle. Finally, *agents* (i.e., system components) are assigned responsibility for each system requirement. KAOS goal models enable developers to decompose the expected behavior of a software system, including information about threats to specific system requirements and how system requirements relate to each system component.

An example KAOS goal model is shown in Figure 8.2, comprising system objectives for a rover with a vision-based object detector. Blue parallelograms represent system goals (e.g., *G12*: "Rover warns nearby pedestrians.") that can be decomposed into sub-goals with AND/OR refinements (shown as connecting lines with overlaying circles). Any potential hazards or obstacles that could prevent the satisfaction of a goal are shown as red parallelograms (e.g., *O1*: "Object detector is degraded/compromised."). At the leaf-level, agents are shown as white hexagons to indicate which system components are responsible for achieving associated goals. The ANUNNAKI framework extends goal models by allowing *utility functions* to be associated with each goal and obstacle [36, 215]. Utility functions map attributes of the managed system to quantifiable metrics that establish a degree of goal satisfaction (i.e., *satisficement*) [45, 124]. For example, the utility function "A1.buzzer == true" is attached to goal *G14*. Thus, when the "buzzer" attribute of agent *A1* is set to *true*, the goal *G14* is evaluated as satisfied.

Through the use of utility functions, the ANUNNAKI framework can interpret a KAOS goal model as a logic tree of run-time system checks to determine the satisfaction of high-level system

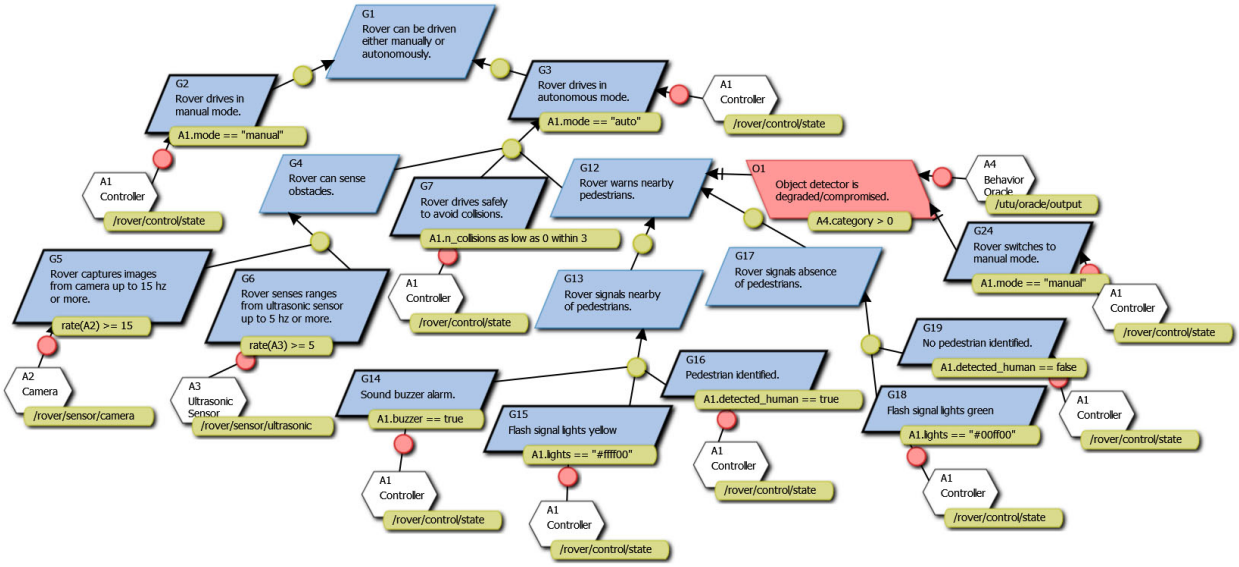


Figure 8.2: An example KAOS goal model to graphically depict system requirements of a robot rover as a hierarchy of logically interconnected goals. Blue parallelograms represent system goals and red parallelograms represent potential obstacles to the satisfaction of goals. White hexagons represent system components responsible for achieving leaf-level goals. The ANUNNAKI framework extends goal models by attaching utility functions to goals/obstacles (shown in yellow ellipses). Agents can be associated with specific message topics (also shown in yellow ellipses) to inform a MAPE-K controller.

```
[G1]: OR
├─[G2]: equal(A1.mode, "manual")
└─[G3]: AND
  ├─[G3]: equal(A1.mode, "auto")
  └─[G3]: AND
    ├─[G12]: AND
    │ ├─[G12]: OR
    │ │ ├─[G13]: AND
    │ │ │ ├─[G14]: equal(A1.buzzer, True)
    │ │ │ ├─[G15]: equal(A1.lights, "#ffff00")
    │ │ │ └─[G16]: equal(A1.detected_human, True)
    │ │ ├─[G17]: AND
    │ │ │ ├─[G18]: equal(A1.lights, "#00ff00")
    │ │ │ └─[G19]: equal(A1.detected_human, False)
    │ └─[O1]: CONDITIONAL
    │   ├─[O1]: IF
    │   │ └─[O1]: greater(A4.category, 0.0)
    │   └─[O1]: THEN
    │     └─[G24]: equal(A1.mode, "manual")
    ├─[G4]: AND
    │ ├─[G5]: greater_equal(rate(A2), 15)
    │ └─[G6]: greater_equal(rate(A3), 5)
    └─[G7]: fuzzy_right(A1.n_collisions, 0, 3)
```

Figure 8.3: A logic tree representation of the KAOS goal model in Figure 8.2. The ANUNNAKI framework automatically parses and interprets goal models as logic trees of utility functions for run-time evaluation of goal satisfaction.

objectives. For example, Figure 8.3 shows a logic tree interpretation of the KAOS goal model in Figure 8.2. The ANUNNAKI framework also extends goal models by allowing message channels to be associated with each agent to specify which channels each respective agent publishes state data. For example, the message channel "/utu/oracle/output" is attached to agent *A4*, indicating that attributes for the behavior oracle can be monitored by observing the corresponding message channel. These extensions enable developers to map the same goal model to different platforms by simply redefining the associated message channels and system attributes.

8.2.4 Model-Driven Monitor and Control

To monitor and control the managed AI system, the ANUNNAKI framework implements UTU as a goal model-driven MAPE-K controller to monitor, analyze, and reconfigure the use of LECs in response to uncertain environmental conditions. In order to mitigate faults from the use of LECs in untrusted conditions, UTU assesses the run-time state of the managed AI system and issues reconfiguration requests in response to the run-time environment. UTU requires goal models described in either a GSN [3] or KAOS [124] format to specify the expected system requirements. UTU also requires a predefined set of tactics to determine what actions should be taken to mitigate faults resulting from violated goal models [37]. Because goal models and tactics are not hard-coded into UTU and are instead model-driven, the ANUNNAKI framework can be deployed with alternative goals and adaptation tactics by simply re-instantiating UTU microservices with new goal models and tactics.

8.2.5 MAPE-K Microservices

To manage adaptation of the managed AI system, UTU comprises five separate microservices to instantiate the MAPE-K loop, each of which are described as follows.

Step 5.a. UTU Knowledge Manager.

A *Knowledge Manager* microservice manages pre-defined goal models and adaptation tactics developed at design time. To enable run-time evaluation of the managed AI system's goal satisfaction,

system variables are parsed from the utility functions associated with the given goal models. The Knowledge Manager acts as a database of agents referenced by given goal models and the run-time values for each respective utility variable.

Step 5.b. UTU Monitor.

A *Monitor* microservice dynamically subscribes to outgoing message traffic of the managed AI system in order to track the system's run-time utility. Subscribed message channels are based on the agents referenced by the active goal model managed by Knowledge Manager. Once subscribed to a message channel, incoming utility data is forwarded back to the Knowledge Manager at a frequency set by the user upon instantiation.

Step 5.c. UTU Analyze.

An *Analyze* microservice determines if any reconfiguration of the managed AI system is required. When any change in a monitored utility variable is detected by the Knowledge Manager, the active goal model and current utility values are forwarded to the Analyze microservice for evaluation. By treating the goal model as a logic tree of utility functions, the Analyze step can substitute run-time utility values into each respective utility function variable to determine goal satisficement. When the entire goal model is found to be unsatisfied, adaptation tactics with preconditions that matches the run-time goal model evaluation are selected and forwarded for planning.

An example adaptation tactic is shown in Figure 8.4, where a tactic is defined in an XML format. Tactics are defined with a set of preconditions, actions, and postconditions. In the given example, a "fail-safe" tactic is defined with a precondition to trigger when *G3* in Figure 8.2 is found to be unsatisfied. The tactic defines a set of actions to be executed once triggered. For the example fail-safe tactic, the actions are to (1) request a mode-change to "manual" mode for the rover and (2) email a notification to the user. Finally, a postcondition is given in the example to state that goal *G3* is expected to be satisfied upon execution of the given actions. Thus, adaptation tactics define specific actions to perform within the ANUNNAKI framework and the managed system, triggered by specific evaluations of the provided goal model.

```
<!-- FAILSAFE TACTIC -->
<tactic id="fail-safe">
  <pre_conditions>
    <goal id="G3" satisfied="false"/>
  </pre_conditions>
  <actions>
    <!-- REQUEST CHANGE MODE TO MANUAL MODE  -->
    <service service_name="/robot/control/mode_change"
             service_type="/robot/ModeChangeService">
      <request mode="manual"/>
    </service>
    <!-- EMAIL USER NOTIFICATION
    <email smtp_server="domain" smtp_port="587">
      <sender email="user@domain" password="***"/>
      <recipient email="anunnaki.utu@domain"/>
      <subject text="Utu Automated Message"/>
      <body text="Failsafe tactic executed."/>
    </email>
  </actions>
  <post_conditions>
    <goal id="G3" satisfied="true"/>
  </post_conditions>
</tactic>
```

Figure 8.4: Example adaptation tactic defined in XML. This "fail-safe" tactic is triggered when precondition goal *G3* (from Figure 8.2) is unsatisfied. Fail-safe actions include a request for the managed robot system to switch to "manual" mode and a request to email the user. Upon executing these actions, the postcondition states that goal *G3* is expected to be satisfied.

Step 5.d. UTU Plan.

A *Plan* microservice prepares adaptation tactics for execution on the managed AI system. When multiple tactics with matching preconditions are given, a single tactic is selected for execution based on a pre-defined priority ranking of tactics (determined at design time). Actions are parsed from the selected tactic and translated into a message format that is compatible with the managed AI system.

Step 5.e. UTU Execute.

An *Execute* microservice realizes adaptation tactics to reconfigure the managed AI system. Adaptation tactics received from the Plan microservice are published to corresponding message channels for the managed AI system.

8.2.6 Software Composability For Trusted AI

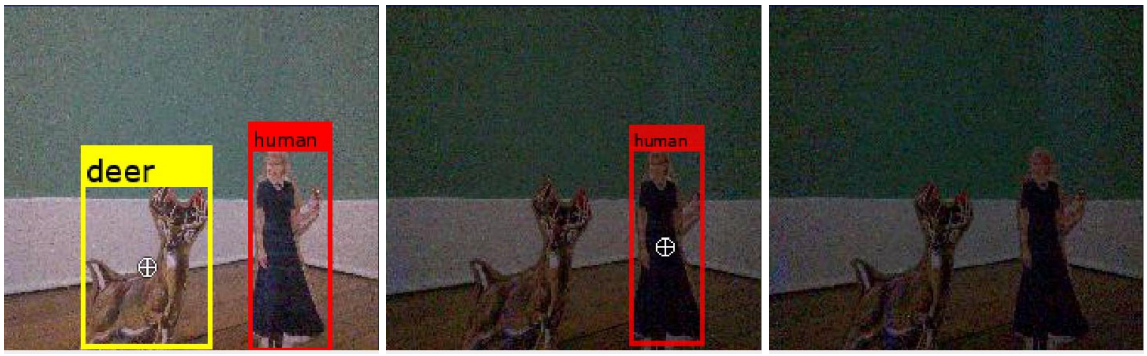
The loose-coupling of microservices in the ANUNNAKI framework enables a range of solutions to be instantiated and composed in order to create a more robust and resilient AI system. When addressing adversarial detection, multiple behavior oracles can be instantiated, each focused on a different form of interference and each publishing behavior assessments over their own respective message channels. When addressing adversarial robustness, multiple alternative learning models can be created to address different respective forms of interference. At run time, these robustified models can be activated when behavior oracles determine the corresponding interference would hinder the default learning model. Finally, the ANUNNAKI framework provides an autonomic controller (i.e., UTU) that is model-driven to enable developers to maintain how the managed AI system responds to uncertain and untrusted environment conditions, without needing to rebuild the managed AI system or its LECs. Instead, developers simply need to update existing goal models and define adaptation tactics to cover unsatisfactory goal states. The composability and autonomy of its microservice architecture enables each process in the ANUNNAKI framework to execute on different hardware platforms, to provide more flexible and scalable deployment options. Furthermore, because ANUNNAKI's microservices can be replaced with any process that honors the framework's API, developers can update behavior oracles and MAPE-K functionality without needing to rebuild the entire ANUNNAKI framework or the managed AI system.

8.3 Demonstration With Autonomous Rover

To demonstrate use of the ANUNNAKI framework to develop Trusted AI, we have implemented a prototype autonomous rover with a vision-based LEC and deployed it in an environment with uncertain run-time conditions (see Section 2.5.3). This section describes the potential impact of *known unknowns* on rover's LEC, and how the ANUNNAKI framework may be used to mitigate faults from using the LEC in the presence of adverse conditions.

Despite the promising results (an F-score of 96.8%) when testing the object detector with validation images, uncertainty remains about the robustness and reliability of the object detector

in the presence of phenomena missing from both training and validation images. For example, Figure 8.5 shows examples of the object detector's performance in a variety of lighting conditions. In Figure 8.5a, the impact of dimmed lighting is shown. As light intensity decreases from Figure 8.5ai. to 8.5aiii., the ability of the object detector diminishes. However, the exact threshold and conditions at which this degradation occurs is *unknown*. Similarly, in Figure 8.5b, the ability of the object detector is degraded as a bright light source is introduced into the scene, either behind the camera (Figure 8.5bii.) or behind the obstacles (Figure 8.5biii.). Though the object detector has been observed to have a high precision and recall under *known* conditions, it remains unclear how it will perform in these *known unknown* conditions.



i. ii. iii.

(a) Impact of dimmed lighting on detection.

i. ii. iii.

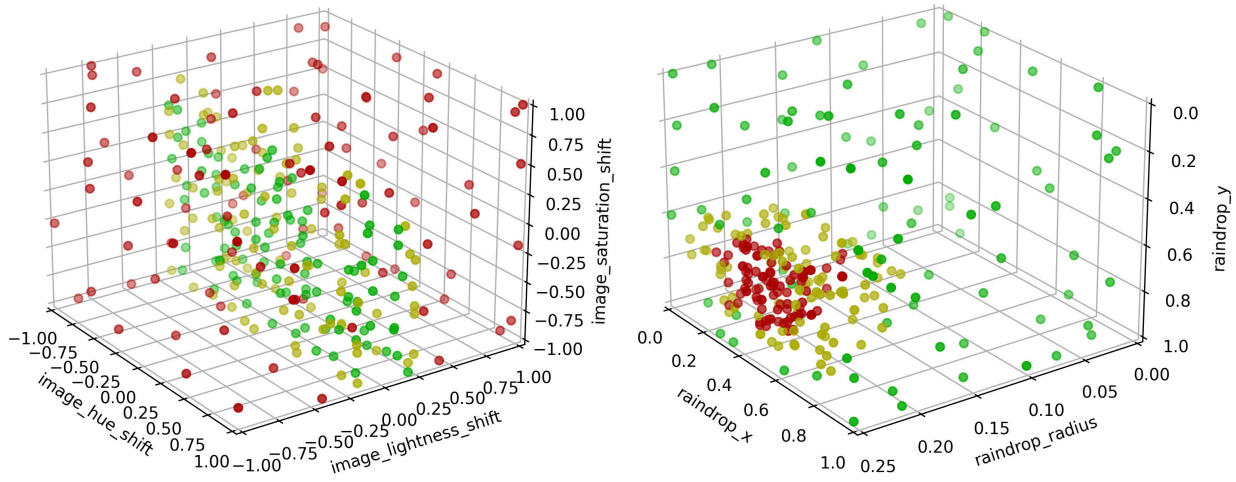
(b) Impact of bright light interference.

Figure 8.5: Examples of real adverse phenomena for vision-based object detection. Objects are correctly identified in normal lighting ((a)i. and (b)i.). Detection is degraded in dim lighting ((a)ii. and (a)iii.). Detection is also degraded when a bright light is placed behind the camera ((b)ii.) or behind the objects ((b)iii.). The boundary between contexts leading to acceptable and unacceptable performance is unknown. Hence, these are *known unknown* phenomena.

8.3.1 Creating Behavior Oracles

Because the threshold between an acceptable environmental and unacceptable environmental condition (Figure 8.5ai. and 8.5aii., respectively) is unknown for the rover's object detector, we need a method to determine when resulting object detections can be trusted. The ANUNNAKI framework can leverage ENLIL to create a behavior oracle to determine this threshold. ENLIL creates an oracle by automatically assessing the object detector's performance boundaries under simulated environmental conditions. For example, ENLIL can automatically assess the object detector's performance under a range of hue, saturation, and lightness (HSL) conditions and create a behavior oracle to predict the object detector's performance under any given HSL context. When additional *known unknown* phenomena are discovered (e.g., a raindrop occluding the camera's view), additional behavior oracles can be generated to predict how the rover's object detector will be impacted by each respective phenomenon.

The scatter plot in Figure 8.6a shows ENLIL's automated behavior assessments under a range of HSL contexts, with each point corresponding to a different context. Green points represent cases in which the object detector's performance was not impacted. Yellow points represent cases in which the object detector's performance is degraded (i.e., >5% decrease in F-score). Red points represent cases in which the object detector's performance is compromised (i.e., >10% decrease in F-score). From these results, ENLIL can generate a behavior oracle that correctly predicts the behavior of the object detector under any HSL context with an 83% accuracy. Similarly, Figure 8.6b shows ENLIL's assessments of the object detector's performance when its view has been occluded by raindrops placed on the camera lens. ENLIL can generate a behavior oracle that correctly predicts the impact of a raindrop occluding the view of the rover's object detector with an 87% accuracy. The ANUNNAKI framework can leverage these behavior oracles to prevent the rover from relying on its object detector under environmental conditions in which it is expected to fail.



(a) HSL Assessment

(b) Raindrop Assessment

Figure 8.6: Scatter plots of ENLIL's automated assessments of a vision-based object detector under HSL variations (a) and with raindrop occlusion (b). Each point represents a unique context of the respective phenomena. Green points represent acceptable conditions, yellow points represent degraded conditions, and red points represent fully compromised conditions. With this data, ENLIL can generate a behavior oracle for each respective phenomena.

8.3.2 Creating Robustified Learning Models

Instead of updating the rover's object detection DNN to be robust to all environmental conditions, our approach is to create a range of context-dependent operational modes, with DNNs robustified for each respective *known unknown* phenomenon. The ANUNNAKI framework uses ENKI to create these robustified DNNs. When evaluating the default object detector under a random sampling of HSL variations, we found that its F-score decreased from 96.8% to only 2%. This significant decrease demonstrates that the object detector is not sufficiently robust to different lighting conditions. However, using ENKI, we were able to create a robustified version of the object detector's DNN that achieves an F-score of 60.7% under random HSL variations. Similarly, we found that the default object detector was not very robust to raindrop occlusion, observing that its F-score decreased from 96.8% to 5% when evaluated with a random sampling of occluding raindrops. Using ENKI, we were able to create a separate DNN more robust to raindrops, with an F-score of 87% for random raindrops. By using separate DNNs that target each respective environmental phenomenon, the integrity of the default object detector is preserved (i.e., it is not influenced by ENKI or any synthetic

data). However, if an adverse condition is uncovered by the ANUNNAKI framework at run time and the object detector's default DNN is expected to fail, the corresponding robustified DNN created by ENKI can be used in place of the default DNN.

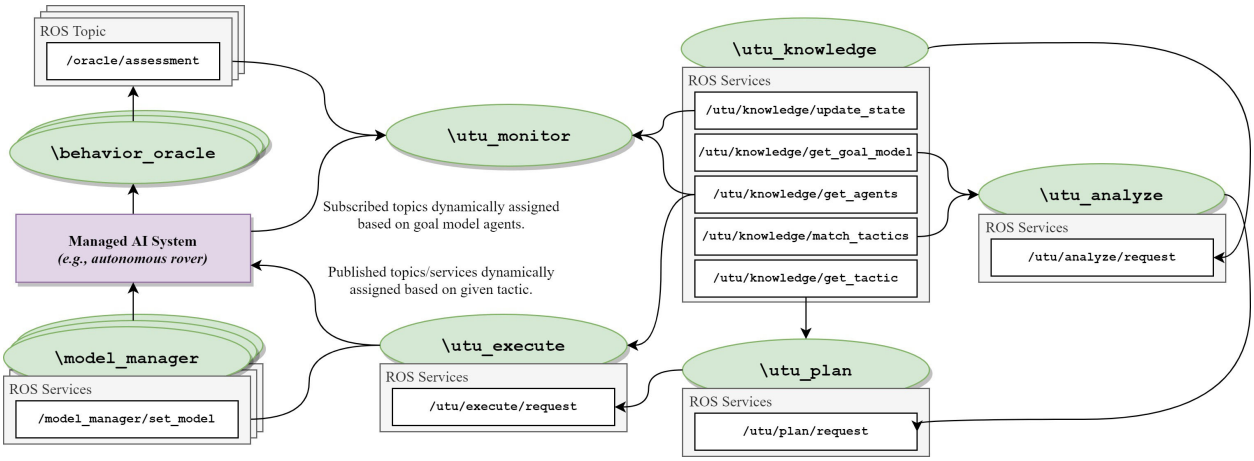


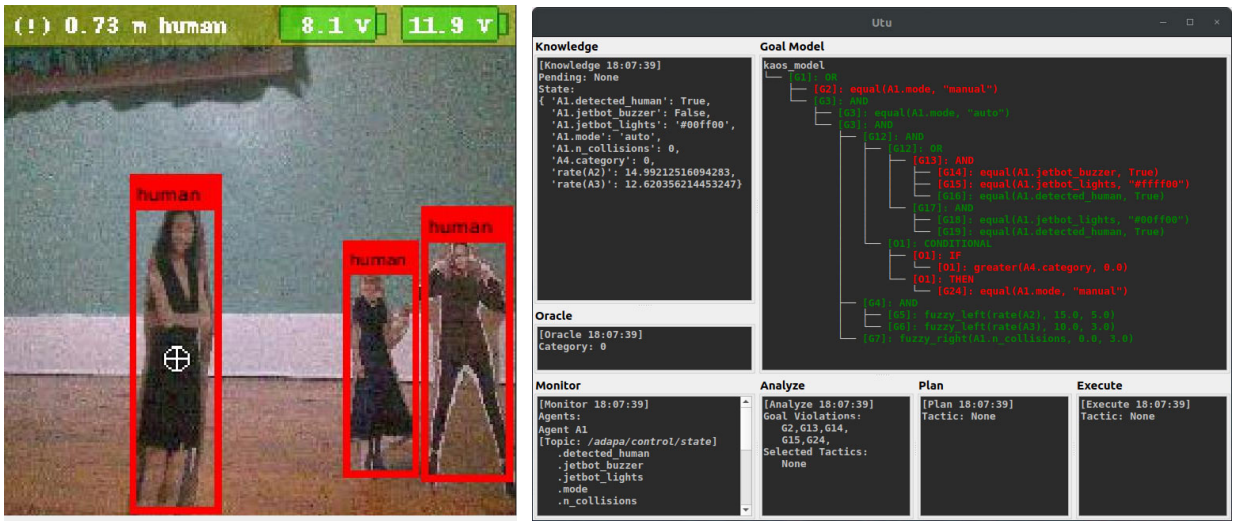
Figure 8.7: ANUNNAKI ROS graph. Ellipses represent ROS nodes. ROS topics and ROS services are shown as rectangles. ANUNNAKI nodes dynamically subscribe and publish to topics of the managed AI system by referencing agents found in given goal models.

8.3.3 Implementing ANUNNAKI Microservices

The ANUNNAKI framework has been implemented using *Python* [173] for platform portability. Each of the microservices in Figure 8.1 can be instantiated as separate ROS nodes within a single ANUNNAKI ROS package. Figure 8.7 illustrates a graph of ANUNNAKI ROS nodes (shown as ellipses) and ROS message topics (shown as rectangles) used for communication. When executed on the same network as the autonomous rover, ANUNNAKI ROS nodes can publish and subscribe to ROS topics provided by the rover in order to monitor and reconfigure the behavior of the rover. ROS nodes are instantiated for each behavior oracle created by ENLIL (e.g., `\behavior_oracle` in Figure 8.7). As ROS nodes, behavior oracles can continuously monitor the rover's sensor data and predict how the object detector will perform at run time, publishing behavior assessments to any ROS node on the same network. Separate ROS nodes are also instantiated for each UTU MAPE-K step (e.g., `\utu_monitor`, `\utu_analyze`, `\utu_plan`, `\utu_execute`, and `\utu_knowledge`). The `\utu_monitor` node monitors ROS message traffic published by the rover and any behavior oracles

that have been instantiated. The `\utu_analyze` node evaluates the active goal model and selects an adaptation tactic when the goal model is not satisfied. The `\utu_plan` and `\utu_execute` nodes then translate the selected tactic into ROS messages that can be published to the rover or into ROS services that can be requested from the rover. Additionally, a `\model_manager` node is instantiated to handle the swapping of ENKI learning models when an adaptation tactic requests a robustified model should be substituted for the default learning model. Thus, the ANUNNAKI framework is realized as a package of coordinated ROS node microservices that automatically monitor and control the rover's object detector with respect to user-defined goal models.

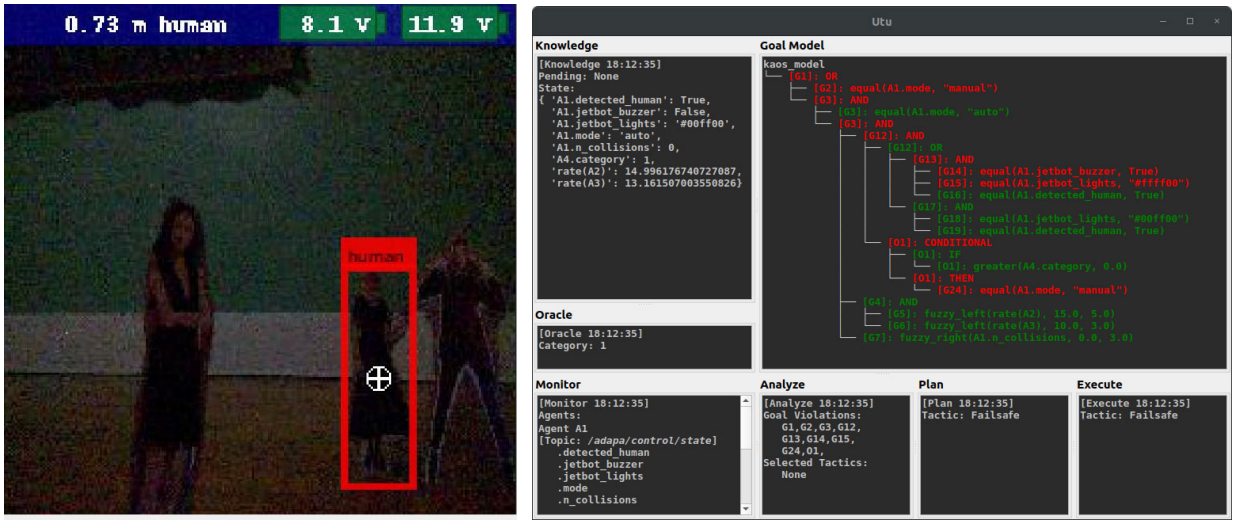
A graphical user interface (GUI) of the ANUNNAKI framework is provided for users to observe the UTU MAPE-K controller at run time (see Figure 8.8 and Figure 8.9 for example scenarios corresponding to ideal and adverse conditions, respectively). Figure 8.8a shows an example of the autonomous rover operating in an ideal lighting condition, where the rover's object detector can properly detect all pedestrians. In Figure 8.8b, the ANUNNAKI GUI displays the state of each UTU MAPE-K step, the output of each behavior oracle, and the current evaluation of the active goal model (from Figure 8.2). The goal model is shown as a logic tree of goals and associated utility functions. Individual goals are highlighted in green when satisfied and red when unsatisfied. In Figure 8.8b, the behavior oracle predicts that the current environment has no adverse impact on the object detector (i.e., Category 0). Thus, the overall goal model in Figure 8.8b is satisfied (i.e., root goal G1 is green), and no adaptation is selected to reconfigure the rover. In contrast, Figure 8.9a shows an example of the rover operating in a dim lighting condition, where the rover's object detector fails to recognize two of the pedestrians in front of the rover. The output of the behavior oracle in Figure 8.9b indicates that the object detector is degraded (i.e., Category 1). The resulting evaluation shows that the goal model is unsatisfied (i.e., root goal G1 is red), and therefore the "fail-safe" tactic from Figure 8.4 is executed to switch the rover from autonomous operation to a manual mode. Thus, the ANUNNAKI framework can prevent a pedestrian collision that would otherwise result from the use of the rover's object detector in dim lighting.



(a) Rover's Perspective

(b) ANUNNAKI Graphical Interface

Figure 8.8: Example of UTU monitoring an autonomous rover. The rover's vision-based object detector can correctly identify pedestrians in bright lighting (a). A GUI displays the state of each UTU MAPE-K microservice (b). UTU evaluates the goal model in Figure 8.2 as a logic tree, highlighting satisfied goals green and unsatisfied goals red. Because the behavior oracle detects no adverse interference (Category 0), the overall goal model is satisfied.



(a) Rover's Perspective

(b) ANUNNAKI Graphical Interface

Figure 8.9: Example of UTU reconfiguring an autonomous rover to prevent use of its object detector LEC in poor lighting (a). A GUI displays the state of each UTU MAPE-K microservice (b). UTU evaluates the goal model in Figure 8.2, highlighting satisfied goals green and unsatisfied goals red. Because the behavior oracle detects an adverse environment condition that degrades object detection (Category 1), a fail-safe tactic has been executed by UTU to reconfigure the rover into manual operation.

8.4 Related Work

This section overviews related work in the area of Trusted AI and addressing issues of uncertainty with DNNs.

Adversarial detection techniques have been introduced to address uncertainty with respect to the reliability of a DNN [142]. *Model inference* enables the creation of behavior models to map specific run-time conditions of a software component to corresponding patterns of expected behavior [156]. Complementary to the behavior oracles introduced in this dissertation, *out-of-distribution techniques* can also assess a degree of confidence by comparing a DNN's run-time inputs to the distribution of its training data [42, 69]. Run-time inputs that are found to fall outside of a DNN's training distribution can be marked as uncertain and trigger alternative actions to prevent use of the DNN in a potentially hazardous state. A key difference between the use of behavior oracles and out-of-distribution techniques is that behavior oracles can target specific *known unknowns* to predict the presence and their respective impact on DNN performance, whereas out-of-distribution techniques typically only identify the presence of inputs that are dissimilar to training inputs.

To address the robustification of DNNs, techniques have been proposed to automatically generate synthetic data for retraining DNNs when real examples of adverse interference are absent (i.e., *known unknown* phenomena) [144, 207, 211, 226, 235]. Typically, synthetic data of simulated interference is generated by transforming existing real-world data. Naïve techniques generate interference by adding random perturbations to given inputs (i.e., fuzzing) [168]. More sophisticated techniques use search-based methods to uncover interference patterns that maximize certain aspects of the AI system (e.g., neuron coverage, Kullback-Leiber divergence, etc.).

Another obstacle to the use of DNNs in safety-critical applications is the lack of model transparency, since DNN architectures are often black box with complex architectures. *Interpretability* refers to the ability for a user to understand a reason for inferences made by a DNN, as well as the applicability of any given DNN output. Visualization methods such as Grad-CAM [196] inform users by highlighting important regions of image inputs that most impact DNN predictions. Other

methods such as *network inversion* [51] reconstruct image inputs from a DNN's internal hidden layers to visualize the fidelity of information extracted by a DNN. Aside from understanding how a DNN processes data and forms inferences, users can also be provided error/uncertainty bounds for each DNN output [210]. Bayesian Neural Networks have been proposed to determine confidence ranges for DNN outputs by learning probability distributions for a DNN's weights and executing specific inputs with different priors [48, 66, 102, 206]. Ensemble methods have also been proposed to use multiple DNN models to form a consensus result [42, 123, 140]. For Trusted AI systems, steps need to be taken to increase transparency with explainable decisions, and visualization techniques such as Grad-CAM can be used in tandem with the ANUNNAKI framework.

8.5 Summary

When autonomous AI systems are deployed in uncertain environments, we need to prevent system failures that would result from the inappropriate use of LECs in adverse contexts. In contrast to existing monolithic techniques to address adversarial detection and robustness, the ANUNNAKI framework provides a more modular microservice approach. As an aggregation of microservices, the ANUNNAKI framework can detect adverse run-time contexts for LECs, monitor and control the use of LECs with respect to user-defined goal models, and leverage robust alternative learning models for targeted adverse phenomena. The composability of ANUNNAKI microservices enables developers to use a modular approach to add, remove, or update behavior oracles and goal models to address new or changing assurance concerns without retraining or rebuilding LECs. Furthermore, the loose-coupling of ANUNNAKI microservices enables the ANUNNAKI framework to run in parallel and independently of managed AI systems. This chapter has demonstrated how the ANUNNAKI framework can be deployed with an autonomous rover to prevent pedestrian collisions resulting from the use of a vision-based object detector in poor lighting conditions. As described, the ANUNNAKI framework requires user-defined goal models and adaptation tactics.

CHAPTER 9

CONCLUSIONS AND FUTURE INVESTIGATIONS

This chapter summarizes the contributions of this doctoral work and outlines directions for future investigation.

This dissertation focuses on managing the capability of an LES in uncertain operating environments. Operational uncertainty at run time poses a challenge for the development of any software system. However, additional challenges arise when developing an LES that must operate safely in uncontrolled environments. Learning models for an LES rely on internal logic that is inferred from limited data, which is typically difficult to interpret by humans. Software assurance for an LES must include arguments to show learning models have been adequately trained and validated to manage uncertain conditions. Furthermore, for safety-critical applications, assurance is required to show that steps are taken to mitigate hazards that arise when learning models are insufficiently trained. This dissertation investigates how automated techniques can improve the robustness and resiliency of an LES to uncertain operational conditions.

Motivated by software assurance practices, this dissertation investigates automated methods to mitigate the impact of uncertain operational conditions on LESs. The effectiveness of a data-driven learning model is limited by training experience, and any gap in training coverage can lead to unexpected and failing behaviors. This doctoral work introduces Enki as an enabling technology to improve the robustness of learning models to phenomena missing from training data. Additionally, this doctoral work introduces Enlil and *behavior oracles* as an enabling technology to predict the impact of *known unknown* phenomena on an LES at run time. Furthermore, this doctoral work explores how LESs can leverage self-adaptive techniques to manage the run-time behavior of autonomous systems, introducing AC-ROS, MoDALAS, and Utu as model-driven MAPE-K controllers. Finally, this doctoral work introduces the Anunnaki framework to coordinate Enki, Enlil, and Utu as Trusted AI services to manage the safe operation of LESs in uncertain environmental conditions, driven by GSN assurance cases and KAOS goal models to mitigate the

impact of failures that would otherwise arise when using learning models with gaps in training coverage.

The contributions and resulting technologies introduced in this dissertation have been demonstrated on multiple learning models and autonomous platforms. ENKI and ENLIL have been validated with image recognition and object detection DNNs, using CIFAR-10, KITTI, and Waymo benchmark datasets. ENKI, AC-ROS, and MODALAS have been validated and demonstrated on the EVORALLY autonomous platform. Finally, the ANUNNAKI framework, including ENKI, ENLIL, and UTU have been validated for real-world environmental uncertainties on an NVIDIA Jetson-controlled autonomous rover with run-time deep learning components for computer vision. As demonstrated in this dissertation, the introduced technologies have been shown to improve the performance of LESs to *known unknowns* and actively mitigate the impact of *known unknowns* on LESs at run-time, thereby making LESs more robust and resilient to environmental uncertainties.

9.1 Summary of Contributions

This doctoral work realized two overarching objectives:

1. Investigate automated methods to improve the *robustness* of LESs operating in uncertain environments, thereby resulting in LESs that can more adequately process data that deviates from training experience.

2. Investigate automated methods to improve the *resiliency* of LESs operating in uncertain environments, thereby resulting in LESs that can adapt and compensate for inadequately trained learning models at run time.

Through these investigations, this doctoral work makes the following research contributions:

- An automated method to assess the behavior of an LES in response to uncertain operating conditions [128].

- An automated method to augment the training and validation of learning models for operating conditions that are inadequately covered by training at design time [126, 130].

- An automated method to detect when learning models have encountered operating conditions for which they have been inadequately trained [127].
- A run-time framework to monitor and control the robustness and resiliency of learning models in response to run-time operating conditions, in order to mitigate failure resulting from the use of inadequately trained learning models [34, 129].

9.2 Future Investigations

Several research areas can be explored based on the results presented in this dissertation. This section describes each of these future investigations in relation to this doctoral work.

Additional Forms of Environmental Uncertainty One direction for future investigation is to extend this doctoral work to address additional forms of uncertainty, including both *known unknowns* and *unknown unknowns* with respect to unexpected system behaviors. The methods and techniques introduced in this dissertation focus on *known unknown* uncertainties, where a risk to an LES can be described (i.e., modeled) but the resulting impact to the LES is unknown. Future work can explore the use of automated methods to harden LESs to *unknown unknowns* by combining and and modifying *known* environmental phenomena in order to discover completely novel phenomena. Additional research can also explore the use of GANs or EAs to help simulate novel phenomena to increase exposure of an LES to unexpected run-time conditions and better prepare an LES to unknown risks.

System State Uncertainty. In addition to addressing environmental uncertainties, another future direction for this doctoral work is to explore uncertainties about the internal system state of autonomous LESs (e.g., system *state estimation* [194]). While an autonomous LES must deal with environmental phenomena that fall outside of its training experience, it must also manage uncertainties about its own system state, especially when state estimates are necessary to inform LECs. One problem when implementing state estimation is uncertainty in the *state observer* due

to unknown parameters, external disturbances, and measurement noise [107]. The methods and techniques introduced by this doctoral work (e.g., the application of ENKI and ENLIL to *digital twins* [76]) can be extended to augment state observers to manage uncertainties with respect to system state estimates.

Additional Trusted AI Assurance Concerns. Future work can extend the ANUNNAKI framework to address new Trusted AI concerns in addition to the robustness and resiliency of autonomous LESs. As described in this dissertation, the ANUNNAKI framework comprises loosely-coupled microservices that reconfigure LECs of an autonomous LES in response to Trusted AI assurance concerns (e.g., robustness). The ANUNNAKI framework can be extended to include additional microservices to address Trusted AI assurance concerns not explicitly covered by this doctoral work, such as the *interpretability* and *fairness* of LECs [200]. Future work can explore methods to assess these additional Trusted AI concerns and integrate them with the ANUNNAKI framework to prevent the use of LECs when such additional assurance concerns are not satisfied.

Security Assurance Cases. Although the ANUNNAKI framework, as demonstrated in this dissertation, monitors components of an LES with respect to assurance cases for functional concerns, steps can be taken to augment the ANUNNAKI framework to additionally support assurance cases for cybersecurity concerns. Possible future investigations can explore the integration of security assurance cases into the ANUNNAKI framework by instantiating an additional autonomic MAPE-SAC [97] feedback control loop service. The new MAPE-SAC service could run in parallel to the UTU service, with additional mechanisms to handle the two interacting MAPE-K and MAPE-SAC controllers [98]. Investigations would require research into possible methods for conflict resolution when two separate autonomic controllers issue competing adaptation tactics. As an extension to this doctoral work, developers would be able to separate cybersecurity concerns from functional concerns in the assurance cases and adaptation tactics provided to the ANUNNAKI framework, enabling the reuse of common cybersecurity assurance cases across platforms with different functional assurance cases.

APPENDIX

APPENDIX

MANUSCRIPTS AND PUBLICATIONS

The following is a list of peer-reviewed publications and manusripts that describe this doctoral work.

- Langford, Michael A., Glen A. Simon, Philip K. McKinley & Betty H.C. Cheng. 2019. Applying Evolution and Novelty Search to Enhance the Resilience of Autonomous Systems. In *Proc. 14th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems* (SEAMS 2019), 63–69. ACM. doi: 10.1109/SEAMS.2019.00017

- Langford, Michael A. & Betty H.C. Cheng. 2019. Enhancing Learning-Enabled Soft-ware Systems to Address Environmental Uncertainty. In *Proc. 16th IEEE Int. Conf. on Autonomic Computing* (ICAC 2019), 115–124. IEEE. doi: 10.1109/ICAC.2019.00023

- Cheng, Betty H.C., Robert J. Clark, Jonathon E. Fleck, Michael A. Langford & Philip K.McKinley. 2020. AC-ROS: Assurance Case Driven Adaptation for the Robot Operating System. In *Proc. 23rd Int. Conf. on Model Driven Engineering Languages and Systems* (MODELS 2020), ACM. doi: 10.1145/3365438.3410952

- Langford, Michael A. & Betty H.C. Cheng. 2021. 'Know What You Know': Predicting Behavior for Learning-Enabled Systems When Facing Uncertainty. In *Proc. 16th Int. ACM/IEEE Symp. on Software Engineering for Adaptive and Self-Managing Systems* (SEAMS 2021), 78-89, IEEE. doi: 10.1109/SEAMS51251.2021.00020

- Langford, Michael A. & Betty H.C. Cheng. 2021. Enki: A Diversity-Driven Approach to Test and Train Robust Learning-Enabled Systems. *ACM Trans. Auton. Adapt. Syst.* (15)2, Article 5, June, ACM, 32 pages. doi: 10.1145/3460959

- Langford, Michael A., Kenneth H. Chan, Jonathon E. Fleck, Philip K. McKinley & Betty H.C. Cheng. 2021. MoDALAS: Model-Driven Assurance for Learning-Enabled Autonomous Systems. In *Proc. 24th Int. ACM/IEEE Conf. on Model Driven Engineering Languages and Systems* (MODELS 2021), 182-193, ACM. doi: 10.1109/MODELS50736.2021.00027
  Note: journal extension submitted to Softw Syst Model

- Langford, Michael A. & Betty H.C. Cheng. 2022. A Modular and Composable Approach to Develop Trusted Artificial Intelligence. *Note: submitted for publication.*

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Abadi, Martin, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu & Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*, 265–283. Berkeley: USENIX. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[2] ACWG. 2018. Goal Structuring Notation Community Standard (Version 2). Tech. rep. Assurance Case Working Group, Safety-Critical Systems Club. https://scsc.uk/r141B:1.

[3] ACWG. 2021. Goal Structuring Notation Community Standard (Version 3). Tech. rep. Assurance Case Working Group, Safety-Critical Systems Club. https://scsc.uk/SCSC-141C. Accessed: Aug 2021.

[4] Aitken, Jonathan M., Sandor M. Veres & Mark Judge. 2014. Adaptation of System Configuration under the Robot Operating System. *IFAC Proceedings Volumes* 47(3). 4484–4492. doi:10.3182/20140824-6-ZA-1003.02531. 19th IFAC World Congress.

[5] Asaadi, Erfan, Ewen Denney & Ganesh Pai. 2020. Quantifying Assurance in Learning-Enabled Systems. doi:10.1007/978-3-030-54549-9_18.

[6] Balalaie, Armin, Abbas Heydarnoori & Pooyan Jamshidi. 2016. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software* 33(3). 42–52. doi:10.1109/MS.2016.64.

[7] Bardaro, Gianluca, Andrea Semprebon & Matteo Matteucci. 2018. A Use Case in Model-Based Robot Development Using AADL and ROS. In *Proc. 1st Int. Workshop on Robotics Software Engineering (RoSE 2018)*, 9–16. New York, NY: ACM. doi:10.1145/3196558.3196560.

[8] Baresi, Luciano, Liliana Pasquale & Paola Spoletini. 2010. Fuzzy Goals for Requirements-Driven Adaptation. In *Proc. 18th IEEE Int. Requirements Engineering Conf. (RE 2010)*, 125–134. Washington, DC: IEEE Computer Society. doi:10.1109/RE.2010.25.

[9] Barr, E. T., M. Harman, P. McMinn, M. Shahbaz & S. Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *Transactions on Software Engineering* 41(5). 507–525. doi:10.1109/TSE.2014.2372785.

[10] Barredo Arrieta, Alejandro, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador Garcia, Sergio Gil-Lopez, Daniel Molina, Richard Ben-

jamins, Raja Chatila & Francisco Herrera. 2020. Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges Toward Responsible AI. *Information Fusion* 58. 82–115. doi:10.1016/j.inffus.2019.12.012.

[11] Bass, Len, Paul Clements & Rick Kazman. 2012. *Software Architecture in Practice*. Upper Saddle River, NJ: Addison-Wesley 3rd edn.

[12] Bau, David, Bolei Zhou, Aditya Khosla, Aude Oliva & Antonio Torralba. 2017. Network Dissection: Quantifying Interpretability of Deep Visual Representations. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR 2017)*, 3319–3327. USA: IEEE. doi:10.1109/CVPR.2017.354.

[13] Bellman, Richard. 2015. *Adaptive Control Processes: A Guided Tour*. Princeton University Press.

[14] Bencomo, Nelly & Amel Belaggoun. 2013. Supporting Decision-Making For Self-Adaptive Systems: From Goal Models to Dynamic Decision Networks. In *Proc. Int. Working Conf. on Requirements Engineering: Foundation for Software Quality (REFSQ 2013)*, 221–236. Berlin, Heidelberg: Springer. doi:10.1007/978-3-642-37422-7_16.

[15] Bencomo, Nelly, Sebastian Götz & Hui Song. 2019. Models@run.time: A Guided Tour of the State of the Art and Research Challenges. *Software and Systems Modeling* 18(5). 3049–3082. doi:10.1007/s10270-018-00712-x.

[16] Bengio, Yoshua. 2020. Priors For Deep Learning of Semantic Representations. Keynote at ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MODELS).

[17] Bergstra, James & Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *J. Mach. Learn. Res.* 13. 281–305.

[18] Blair, Gordon, Nelly Bencomo & Robert B. France. 2009. Models@ Run.Time. *Computer* 42(10). 22–27. doi:10.1109/MC.2009.326.

[19] Borde, Etienne, Grégory Haïk & Laurent Pautet. 2009. Mode-Based Reconfiguration of Critical Software Component Architectures. In *Proc. Conf. on Design, Automation and Test in Europe (DATE 2009)*, 1160–1165. Leuven: European Design and Automation Association.

[20] Borg, Markus, Cristofer Englund, Krzysztof Wnuk, Boris Duran, Christoffer Levandowski, Shenjian Gao, Yanwen Tan, Henrik Kaijser, Henrik Lönn & Jonas Törnqvist. 2019. Safely Entering the Deep: A Review of Verification and Validation for Machine Learning and a Challenge Elicitation in the Automotive Industry. *Journal of Automotive Software Engineering* 1. 1–19. doi:10.2991/jase.d.190131.001.

[21] Boursinos, Dimitrios & Xenofon Koutsoukos. 2021. Assurance Monitoring of Learning-Enabled Cyber-Physical Systems Using Inductive Conformal Prediction Based on Distance

Learning. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 35(2). 251–264. doi:10.1017/S089006042100010X.

[22] Boussaa, Mohamed, Olivier Barais, Gerson Sunyé & Benoît Baudry. 2015. A Novelty Search Approach for Automatic Test Data Generation. In *Proc. 8th Int. Workshop on Search-Based Software Testing (SBST 2015)*, 40–43. USA: IEEE. doi:10.1109/SBST.2015.17.

[23] Boussaa, Mohamed, Olivier Barais, Gerson Sunyé & Benoît Baudry. 2015. A Novelty Search-Based Test Data Generator for Object-Oriented Programs. In *Proc. Genetic and Evolutionary Computation Conf. Companion (GECCO 2015)*, 1359–1360. New York, NY: ACM. doi:10.1145/2739482.2764716.

[24] Briand, Lionel, Shiva Nejati, Mehrdad Sabetzadeh & Domenico Bianculli. 2016. Testing the Untestable: Model Testing of Complex Software-Intensive Systems. In *Proc. 38th Int. Conf. on Software Engineering Companion (ICSE 2016)*, 789–792. New York, NY: ACM. doi:10.1145/2889160.2889212.

[25] Brodley, Carla E. & Mark A. Friedl. 1999. Identifying Mislabeled Training Data. *J. Artif. Int. Res.* 11(1). 131–167.

[26] Burton, Simon, Lydia Gauerhof & Christian Heinzemann. 2017. Making the Case for Safety of Machine Learning in Highly Automated Driving. In *Computer Safety, Reliability, and Security. Lecture Notes in Computer Science (SAFECOMP 2017)*, vol. 10489, Cham: Springer. doi:10.1007/978-3-319-66284-8_1.

[27] Calikli, Gul & Ayse Bener. 2010. Empirical Analyses of the Factors Affecting Confirmation Bias and the Effects of Confirmation Bias on Software Developer/Tester Performance. In *Proc. 6th Int. Conf. on Predictive Models in Software Engineering (PROMISE 2010)*, New York, NY: ACM. doi:10.1145/1868328.1868344.

[28] Calinescu, Radu, Carlo Ghezzi, Marta Kwiatkowska & Raffaela Mirandola. 2012. Self-Adaptive Software Needs Quantitative Verification at Runtime. *Commun. ACM* 55(9). 69–77. doi:10.1145/2330667.2330686.

[29] Calinescu, Radu, Danny Weyns, Simos Gerasimou, Muhammad Usman Iftikhar, Ibrahim Habli & Tim Kelly. 2018. Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases. *IEEE Transactions on Software Engineering* 44(11). 1039–1069.

[30] Cámara, Javier, Rogério de Lemos, Carlo Ghezzi & Antónia Lopes (eds.). 2013. *Assurances for Self-Adaptive Systems: Principles, Models, and Techniques*. Berlin, Heidelberg: Springer.

[31] Carleton, Anita D., Erin Harper, Tim Menzies, Tao Xie, Sigrid Eldh & Michael R. Lyu. 2020. The AI Effect: Working at the Intersection of AI and SE. *IEEE Software* 37(4). 26–35. doi:10.1109/MS.2020.2987666.

[32] Cashmore, Michael, Maria Fox, Derek Long, Daniele Magazzeni, Bram Ridder, Arnau Carreraa, Narcís Palomeras, Natàlia Hurtós & Marc Carrerasa. 2015. ROSPlan: Planning in the Robot Operating System. In *Proc. 25th Int. Conf. on Automated Planning and Scheduling (ICAPS 2015)*, 333–341. Palo Alto, CA: AAAI Press.

[33] Chechik, Marsha & Arie Gurfinkel. 2007. A Framework for Counterexample Generation and Exploration. *Int. J. Softw. Tools Technol. Transf.* 9(5–6). 429–445. doi:10.5555/3220881.3220980.

[34] Cheng, Betty H. C., Robert J. Clark, Jonathon E. Fleck, Michael A. Langford & Philip K. McKinley. 2020. AC-ROS: Assurance Case Driven Adaptation for the Robot Operating System. In *Proc. 23rd Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2020)*, New York, NY: ACM.

[35] Cheng, Betty H. C., Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns & Jon Whittle. 2009. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems [Outcome of a Dagstuhl Seminar]*, vol. 5525, 1–26. Cham: Springer. doi:10.1007/978-3-642-02161-9_1.

[36] Cheng, Shang-Wen. 2008. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*: Carnegie Mellon University dissertation.

[37] Cheng, Shang-Wen, David Garlan & Bradley Schmerl. 2006. Architecture-Based Self-Adaptation in the Presence of Multiple Objectives. In *Proc. Int. Workshop on Self-Adaptation and Self-Managing Systems (SEAMS 2006)*, 2–8. New York, NY: ACM. doi:10.1145/1137677.1137679.

[38] Ciupa, Ilinca, Bertrand Meyer, Manuel Oriol & Alexander Pretschner. 2008. Finding Faults: Manual Testing vs. Random+ Testing vs. User Reports. In *Proc. 19th Int. Symposium on Software Reliability Engineering (ISSRE 2008)*, 157–166. USA: IEEE Computer Society. doi:10.1109/ISSRE.2008.18.

[39] Clark, Anthony, Byron DeVries, Jared Moore, Betty H. C. Cheng & Philip McKinley. 2016. An Evolutionary Approach to Discovering Execution Mode Boundaries for Adaptive Controllers. In *Proc. IEEE Symposium Series on Computational Intelligence (SSCI 2016)*, 1–8. New York, NY: IEEE. doi:10.1109/SSCI.2016.7850178.

[40] Clarke, Edmund M., William Klieber, Miloš Nováček & Paolo Zuliani. 2011. Model Checking and the State Explosion Problem. In *Tools for Practical Software Verification. Lecture Notes in Computer Science (LASER 2011)*, vol. 7682, Cham: Springer. doi:10.1007/978-3-642-35746-6_1.

[41] Clarke, Jr., Edmund M., Orna Grumberg, Daniel Kroening, Doron Peled & Helmut Veith. 2018. *Model Checking*. MIT Press 2nd edn.

[42] Cortés-Ciriano, Isidro & Andreas Bender. 2018. Deep Confidence: A Computationally Efficient Framework for Calculating Reliable Prediction Errors for Deep Neural Networks. *J. Chem Inf. Model.* 59(3). 1269–1281. doi:10.1021/acs.jcim.8b00542.

[43] Dalpiaz, Fabiano, Alexander Borgida, Jennifer Horkoff & John Mylopoulos. 2013. Runtime Goal Models: Keynote. In *Proc. 7th IEEE Int. Conf. on Research Challenges in Information Science (RCIS 2013)*, 1–11. New York, NY: IEEE. doi:10.1109/RCIS.2013.6577674.

[44] D'Angelo, Mirko, Simos Gerasimou, Sona Ghahremani, Johannes Grohmann, Ingrid Nunes, Evangelos Pournaras & Sven Tomforde. 2019. On Learning in Collective Self-Adaptive Systems: State of Practice and a 3D Framework. In *Proc. 14th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2019)*, 13–24. USA: IEEE. doi:10.1109/SEAMS.2019.00012.

[45] deGrandis, Paul & Giuseppe Valetto. 2009. Elicitation and Utilization of Application-Level Utility Functions. In *Proc. 6th Int. Conf. on Autonomic Computing (ICAC 2009)*, 107–116. New York, NY: ACM. doi:10.1145/1555228.1555259.

[46] Denney, Ewen, Ganesh Pai & Josef Pohl. 2012. AdvoCATE: An Assurance Case Automation Toolset. In *Computer Safety, Reliability, and Security (SAFECOMP 2012)*, vol. 7613, 8–21. Berlin, Heidelberg: Springer. doi:10.1007/978-3-642-33675-1_2.

[47] Denney, Ewen, Ganesh Pai & Iain Whiteside. 2017. Model-Driven Development of Safety Architectures. In *Proc. ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2017)*, 156–166. USA: IEEE. doi:10.1109/MODELS.2017.27.

[48] Depeweg, Stefan, José Miguel Hernández-Lobato, Steffen Udluft & Thomas Runkler. 2017. Sensitivity Analysis for Predictive Uncertainty in Bayesian Neural Networks. *CoRR* abs/1712.03605.

[49] Dersten, Sara, Peter Wallin, Joakim Fröberg & Jakob Axelsson. 2016. Analysis of the Information Needs of an Autonomous Hauler in a Quarry Site. In *Proc. 11th System of Systems Engineering Conf. (SoSE 2016)*, 1–6. doi:10.1109/SYSOSE.2016.7542936.

[50] Ding, Junhua, Xiaojun Kang & Xin-Hua Hu. 2017. Validating a Deep Learning Framework by Metamorphic Testing. In *Proc. 2nd Int. Workshop on Metamorphic Testing (MET 2017)*, 28–34. USA: IEEE. doi:10.1109/MET.2017.2.

[51] Dosovitskiy, Alexey & Thomas Brox. 2016. Inverting Visual Representations with Convolutional Networks. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR 2016)*, 4829–4837. USA: IEEE. doi:10.1109/CVPR.2016.522.

[52] Eiben, Agoston E. & James E. Smith. 2015. *Introduction to Evolutionary Computing*. Berlin, Heidelberg: Springer-Verlag 2nd edn.

[53] Elkady, Ayssam & Tarek Sobh. 2012. Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography. *Journal of Robotics* 2012(Article ID 959013). doi:10.1155/2012/959013.

[54] Eykholt, Kevin, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno & Dawn Song. 2018. Robust Physical-World Attacks on Deep Learning Visual Classification. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR 2018)*, 1625–1634. USA: IEEE. doi:10.1109/CVPR.2018.00175.

[55] Fan, Chen-Yuan & Shang-Pin Ma. 2017. Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report. In *Proc. Int Conf. on AI Mobile Services (AIMS 2017)*, 109–112. USA: IEEE. doi:10.1109/AIMS.2017.23.

[56] Feng, Yang, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang & Zhenyu Chen. 2020. DeepGini: Prioritizing Massive Tests to Enhance the Robustness of Deep Neural Networks. In *Proc. 29th ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA 2020)*, 177–188. USA: Association for Computing Machinery. doi:10.1145/3395363.3397357.

[57] Filieri, Antonio, Carlo Ghezzi & Giordano Tamburrelli. 2012. A Formal Approach to Adaptive Software: Continuous Assurance of Non-Functional Requirements. *Formal Asp. Comput.* 24(2). 163–186. doi:10.1007/s00165-011-0207-2.

[58] Floreano, Dario, Phil Husbands & Stefano Nolfi. 2008. Evolutionary Robotics. In *Springer Handbook of Robotics*, 1423–1451. Berlin, Heidelberg: Springer. doi:10.1007/978-3-540-30301-5_62.

[59] Franklin, Dustin. 2019. Jetson Nano Brings AI Computing to Everyone. https://developer.nvidia.com/blog/jetson-nano-ai-computing/.

[60] Fraser, Gordon & Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proc. 19th ACM SIGSOFT Symposium and the 13th European Conf. on Foundations of Software Engineering (ESEC/FSE 2011)*, 416–419. New York, NY: ACM. doi:10.1145/2025113.2025179.

[61] Fraser, Gordon & Neil Walkinshaw. 2015. Assessing and Generating Test Sets in Terms of Behavioural Adequacy. *Softw. Test. Verif. Reliab.* 25(8). 749–780. doi:10.1002/stvr.1575.

[62] Fredericks, Erik M. & Betty H. C. Cheng. 2014. Automated Generation of Adaptive Test Plans for Self-Adaptive Systems. In *Proc. 10th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015)*, 157–168. New York, NY: ACM.

[63] Fredericks, Erik M., Byron DeVries & Betty H. C. Cheng. 2014. Towards Run-time Adap-

tation of Test Cases for Self-Adaptive Systems in the Face of Uncertainty. In *Proc. 9th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2014)*, 17–26. New York, NY: ACM.

[64] Fritzsch, Jonas, Justus Bogner, Alfred Zimmermann & Stefan Wagner. 2018. From Monolith to Microservices: A Classification of Refactoring Approaches. In *Proc. 1st Int. Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, 128–141. Cham: Springer. doi:10.1007/978-3-030-06019-0_10.

[65] Gaiser, Hans, Maarten de Vries, Valeriu Lacatusu, vcarpani, Ashley Williamson, Enrico Liscio, Andr/'as, Yann Henon, jjiun, Cristian Gratie, Mihai Morariu, Charles Ye, Martin Zlocha, Ben Weinstein, Rodrigo Meira de Andrade, Pedro Conceição, Alexander Pacha, hannesedvartsen, Daniyal Shahrokhian, Wudi Fang, Mike Clark, meagerYak, Iver Jordal, Max Van Sande, Jin, Etienne-Meunier, Andrew Grigorev, Guillaume Erhard, Eduardo Ramos & Denis Dowling. 2019. fizyr/keras-retinanet 0.5.1. doi:10.5281/zenodo.3250670.

[66] Gal, Yarin & Zoubin Ghahramani. 2016. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. In *Proc. 33rd Int. Conf. on Machine Learning (ICML 2016)*, vol. 48, 1050–1059. JMLR.org.

[67] Garg, Kshitiz & Shree K. Nayar. 2004. Photometric Model of a Rain Drop. Tech. rep. Columbia University. https://www1.cs.columbia.edu/CAVE/publications/pdfs/Garg_TR04.pdf.

[68] Gastaldi, Xavier. 2017. Shake-Shake Regularization. *CoRR* abs/1705.07485. Appeared in ICLR 2017.

[69] Gawlikowski, Jakob, Cedrique Rovile Njieutcheu Tassi, Mohsin Ali, Jongseok Lee, Matthias Humt, Jianxiang Feng, Anna Kruspe, Rudolph Triebel, Peter Jung, Ribana Roscher, Muhammad Shahzad, Wen Yang, Richard Bamler & Xiao Xiang Zhu. 2021. A Survey of Uncertainty in Deep Neural Networks. *CoRR* abs/2107.03342.

[70] Geiger, Andreas, Philip Lenz & Raquel Urtasun. 2012. Are We Ready For Autonomous Driving? The KITTI Vision Benchmark Suite. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR 2012)*, 3354–3361. USA: IEEE. doi:10.1109/CVPR.2012.6248074.

[71] Goldfain, Brian, Paul Drews, Changxi You, Matthew Barulic, Orlin Velev, Panagiotis Tsiotras & James M Rehg. 2019. AutoRally: An Open Platform for Aggressive Autonomous Driving. *IEEE Control Systems Magazine* 39(1). 26–55. doi:10.1109/MCS.2018.2876958.

[72] Goodenough, John, Charles Weinstock & Ari Klein. 2012. Toward a Theory of Assurance Case Confidence. Tech. Rep. CMU/SEI-2012-TR-002 Software Engineering Institute, Carnegie Mellon University Pittsburgh, PA. https://resources.sei.cmu.edu/library/asset-

view.cfm?AssetID=28067.

[73] Goodfellow, Ian, Yoshua Bengio & Aaron Courville. 2016. *Deep Learning*. Cambridge, MA: MIT.

[74] Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville & Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems*, vol. 27, 2672–2680. Red Hook, NY: Curran Associates.

[75] Goodfellow, Ian J., Jonathons Shlens & Christian Szegedy. 2014. Explaining and Harnessing Adversarial Examples. *CoRR* abs/1412.6572. Appeared in ICLR 2015.

[76] Grieves, Michael & John Vickers. 2017. Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems. In *Transdisciplinary Perspectives on Complex Systems: New Findings and Approaches*, 85–113. Cham: Springer. doi:10.1007/978-3-319-38756-7_4.

[77] Gu, Rong, Raluca Marinescu, Cristina Seceleanu & Kristina Lundqvist. 2018. Formal Verification of an Autonomous Wheel Loader by Model Checking. In *Proc. 6th Conf. on Formal Methods in Software Engineering (FormaliSE 2018)*, 74–83. New York, NY: ACM. doi:10.1145/3193992.3193999.

[78] Gutjahr, Walter J. 1999. Partition Testing vs. Random Testing: The Influence of Uncertainty. *IEEE Trans. Softw. Eng.* 25(5). 661–674. doi:10.1109/32.815325.

[79] Hájek, Petr. 2013. *Metamathematics of Fuzzy Logic*. Dordrecht: Springer.

[80] Hamon, Ronan, Henrik Junklewitz & Ignacio Sanchez. 2020. Robustness and Explainability of Artificial Intelligence. Tech. rep. European Commission / Joint Research Centre (JRC).

[81] Hao, Karen. 2019. This Is How AI Bias Really Happens–And Why It's So Hard to Fix. *MIT Technology Review* Tech policy/AI Ethics. https://www.technologyreview.com/2019/02/04/137602/this-is-how-ai-bias-really-happensand-why-its-so-hard-to-fix/.

[82] Harman, Mark, S. Afshin Mansouri & Yuanyuan Zhang. 2012. Search-Based Software Engineering: Trends, Techniques and Applications. *ACM Comput. Surv.* 45(1). doi:10.1145/2379776.2379787.

[83] Hartley, Richard I. & Peter Sturm. 1997. Triangulation. *Comput. Vis. Image Underst.* 68(2). 146–157. doi:10.1006/cviu.1997.0547.

[84] Hartsell, Charles, Nagabhushan Mahadevan, Shreyas Ramakrishna, Abhishek Dubey, Theodore Bapty, Taylor Johnson, Xenofon Koutsoukos, Janos Sztipanovits & Gabor Karsai. 2019. Model-Based Design For CPS With Learning-Enabled Components. In *Proc.*

Workshop on Design Automation for CPS and IoT (DESTION 2019), 1–9. ACM. doi: 10.1145/3313151.3313166.

[85] Hatcher, William Grant & Wei Yu. 2018. A Survey of Deep Learning: Platforms, Applications and Emerging Research Trends. *IEEE Access* 6. 24411–24432. doi: 10.1109/ACCESS.2018.2830661.

[86] He, Haibo & Edwardo A. Garcia. 2009. Learning from Imbalanced Data. *IEEE Transactions on Knowledge and Data Engineering* 21(9). 1263–1284. doi:10.1109/TKDE.2008.239.

[87] He, Kaiming, Xiangyu Zhang, Shaoqing Ren & Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR 2016)*, 770–778. USA: IEEE. doi:10.1109/CVPR.2016.90.

[88] Hochgeschwender, Nico, Luca Gherardi, Azamat Shakhirmardanov, Gerhard K. Kraetzschmar, Davide Brugali & Herman Bruyninckx. 2013. A Model-Based Approach to Software Deployment in Robotics. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2004)*, 3907–3914. USA: IEEE.

[89] Hornik, Kurt, Maxwell Stinchcombe & Halbert White. 1989. Multilayer feedforward networks are universal approximators. *Neural Networks* 2(5). 359–366. doi:10.1016/0893-6080(89)90020-8.

[90] Huang, Wei, Yifan Zhou, Youcheng Sun, Alec Banks, Jie Meng, James Sharp, Simon Maskell & Xiaowei Huang. 2020. Formal Verification of Robustness and Resilience of Learning-Enabled State Estimation Systems for Robotics. *CoRR* abs/2010.08311.

[91] Hughes, John F., Andries van Dam, Morgan McGuire, David F. Sklar, James D. Foley, Steven Feiner & Kurt Akeley. 2013. *Computer Graphics: Principles and Practice*. Upper Saddle River, NJ: Addison-Wesley.

[92] IBM. 2021. Trusted AI. Website. Accessed: Aug 2021. https://www.research.ibm.com/teams/trusted-ai.

[93] Iftikhar, M. Usman & Danny Weyns. 2014. ActivFORMS: Active Formal Models For Self-Adaptation. In *Proc. 9th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2009)*, 125–134. New York, NY: ACM. doi:10.1145/2593929.2593944.

[94] Institute, Deloitte AI. 2020. Deloitte Introduces Trustworthy AI Framework to Guide Organizations in Ethical Application of Technology in the Age of With. Press Release. Accessed: Aug 2021. https://www2.deloitte.com/us/en/pages/about-deloitte/articles/press-releases/deloitte-introduces-trustworthy-ai-framework.html.

[95] Ioffe, Sergey & Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network

Training by Reducing Internal Covariate Shift. In *Proc. 32nd Int. Conf. on Machine Learning (ICML 2015)*, vol. 37, 448–456. Brookline, MA: Microtome Publishing.

[96] Jacobi, Nick, Phil Husbands & Inman Harvey. 1995. Noise and the Reality Gap: The Use of Simulation in Evolutionary Robotics. In *Proc. 3rd European Conf. on Advances in Artificial Life*, 704–720. Berlin, Heidelberg: Springer-Verlag.

[97] Jahan, Sharmin, Allen Marshall & Rose F. Gamble. 2019. Evaluating Security Assurance Case Adaptation. In *Proc. 52nd Hawaii Int. Conf. on System Sciences (HICSS 2019)*, 1–10. Manoa, HI: ScholarSpace.

[98] Jahan, Sharmin, Ian Riley, Charles Walter, Rose F. Gamble, Matt Pasco, Philip K. McKinley & Betty H.C. Cheng. 2020. MAPE-K/MAPE-SAC: An Interaction Framework for Adaptive Systems with Security Assurance Cases. *Future Generation Computer Systems* 109. 197–209. doi:10.1016/j.future.2020.03.031.

[99] Janai, Joel, Fatma Güney, Aseem Behl & Andreas Geiger. 2017. Computer Vision for Autonomous Vehicles: Problems, Datasets and State-of-the-Art. *CoRR* abs/1704.05519.

[100] Jiao, Licheng, Fan Zhang, Fang Liu, Shuyuan Yang, Lingling Li, Zhixi Feng & Rong Qu. 2019. A Survey of Deep Learning-Based Object Detection. *IEEE Access* 7. 128837–128868. doi:10.1109/ACCESS.2019.2939201.

[101] Jo, Jason & Yoshua Bengio. 2017. Measuring the Tendency of CNNs to Learn Surface Statistical Regularities. *CoRR* abs/1711.11561.

[102] Jospin, Laurent Valentin, Wray Buntine, Farid Boussaid, Hamid Laga & Mohammed Bennamoun. 2020. Hands-on Bayesian Neural Networks–a Tutorial for Deep Learning Users. *CoRR* abs/2007.06823.

[103] Kawaguchi, Kenji, Leslie Pack Kaelbling & Yoshua Bengio. 2018. Generalization in Deep Learning. Tech. rep. MIT. https://lis.csail.mit.edu/pubs/kawaguchi-techreport18.pdf.

[104] Kelly, Tim & Rob Weaver. 2004. The Goal Structuring Notation–A Safety Argument Notation. In *Proc. Dependable Systems and Networks 2004 Workshop on Assurance Cases*, CiteseerX.

[105] Kephart, Jeffrey O. & David M. Chess. 2003. The Vision Of Autonomic Computing. *Computer* 36(1). 41–50. doi:10.1109/MC.2003.1160055.

[106] Kephart, Jeffrey O. & Rajarshi Das. 2007. Achieving Self-Management via Utility Functions. *IEEE Internet Computing* 11(1). 40–48. doi:10.1109/MIC.2007.2.

[107] Khan, Awais, Wei Xie, Langwen Zhang & Long-Wen Liu. 2020. Design and applications of interval observers for uncertain dynamical systems. *IET Circuits, Devices & Systems* 14(6).

721–740.

[108] Kheradpisheh, Saeed Reza, Masoud Ghodrati, Mohammad Ganjtabesh & Timothée Masquelier. 2016. Deep Networks Can Resemble Human Feed-forward Vision in Invariant Object Recognition. *Scientific Reports* 6(32672). doi:10.1038/srep32672.

[109] Kingma, Diederik P. & Jimmy Ba. 2014. Adam: a Method for Stochastic Optimization. *CoRR* abs/1412.6980. Appeared in ICLR 2015.

[110] Kirch, Wilhelm (ed.). 2008. *Pearson's Correlation Coefficient* 1090–1091. Dordrecht: Springer Netherlands. doi:10.1007/978-1-4020-5614-7_2569.

[111] Klir, George J. & Bo Yuan (eds.). 1996. *Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems: Selected Papers by Lotfi A. Zadeh*. USA: World Scientific.

[112] Knight, Will. 2017. The Dark Secret at the Heart of AI. *MIT Technology Review* Artificial intelligence/Machine learning. https://www.technologyreview.com/2017/04/11/5113/the-dark-secret-at-the-heart-of-ai/.

[113] Koenig, Nathan & Andrew Howard. 2004. Design and Use Paradigms for Gazebo, an Open-source Multi-robot Simulator. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, vol. 3, 2149–2154.

[114] Kolak, Sophia, Afsoon Afzal, Claire Le Goues, Michael Hilton & Christopher Steven Timperley. 2020. It Takes a Village to Build a Robot: An Empirical Study of The ROS Ecosystem. In *Proc. IEEE Int. Conf. on Software Maintenance and Evolution (ICSME 2020)*, 430–440. USA: IEEE. doi:10.1109/ICSME46990.2020.00048.

[115] Koulamas, Christos & Athanasios Kalogeras. 2018. Cyber-Physical Systems and Digital Twins in the Industrial Internet of Things [Cyber-Physical Systems]. *Computer* 51(11). 95–98. doi:10.1109/MC.2018.2876181.

[116] Kozma, Robert, Roman Ilin & Hava T. Siegelmann. 2018. Evolution of Abstraction Across Layers in Deep Learning Neural Networks. *Procedia Computer Science* 144. 203–213. doi:10.1016/j.procs.2018.10.520. Appeared in the Conf. on Big Data and Deep Learning (INNS).

[117] Kramer, Jeff & Jeff Magee. 1990. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. Softw. Eng.* 16(11). 1293–1306. doi:10.1109/32.60317.

[118] Kramer, Samuel Noah. 1963. *The Sumerians: Their History, Culture, and Character*. The University of Chicago Press.

[119] Krizhevsky, Alex. 2009. Learning Multiple Layers of Features from Tiny Images. Tech. rep. University of Toronto.

[120] Krizhevsky, Alex, Ilya Sutskever & Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60(6). 84–90. doi:10.1145/3065386.

[121] Krupitzer, Christian, Felix Maximilian Roth, Martin Pfannemüller & Christian Becker. 2016. Comparison of Approaches for Self-Improvement in Self-Adaptive Systems. In *Proc. 13th IEEE Int. Conf. on Autonomic Computing (ICAC 2016)*, 308–314. USA: IEEE. doi:10.1109/ICAC.2016.18.

[122] Kurakin, Alexey, Ian J. Goodfellow & Sami Bengio. 2016. Adversarial Machine Learning at Scale. *CoRR* abs/1611.01236. Appeared in ECAI 2016.

[123] Lakshminarayanan, Balaji, Alexander Pritzel & Charles Blundell. 2017. Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles. In *Proc. 31st Int. Conf. on Neural Information Processing Systems (NIPS 2017)*, 6405–6416. Red Hook, NY: Curran Associates Inc.

[124] van Lamsweerde, Axel & Emmanuel Letier. 2004. From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering. In *Radical Innovations of Software and Systems Engineering in the Future. Lecture Notes in Computer Science (RISSEF 2002)*, vol. 2941, Berlin, Heidelberg: Springer-Verlag. doi:10.1007/978-3-540-24626-8_23.

[125] Langari, Zarrin & Tom Maibaum. 2013. Safety Cases: A Review of Challenges. In *Proc. 1st Int. Workshop on Assurance Cases for Software-Intensive Systems (ASSURE 2013)*, 1–6. USA: IEEE.

[126] Langford, Michael A. & Betty H. C. Cheng. 2019. Enhancing Learning-Enabled Software Systems to Address Environmental Uncertainty. In *Proc. 16th IEEE Int. Conf. on Autonomic Computing (ICAC 2019)*, 115–124. USA: IEEE. doi:10.1109/ICAC.2019.00023.

[127] Langford, Michael A. & Betty H.C. Cheng. 2021. "Know What You Know": Predicting Behavior for Learning-Enabled Systems When Facing Uncertainty. In *Proc. 16th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2021)*, 78–89. New York, NY: ACM.

[128] Langford, Michael A., Glen A. Simon, Philip K. McKinley & Betty H. C. Cheng. 2019. Applying Evolution and Novelty Search to Enhance the Resilience of Autonomous Systems. In *Proc. 14th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2019)*, 63–69. New York, NY: ACM. doi:10.1109/SEAMS.2019.00017.

[129] Langford, Michael Austin, Kenneth H. Chan, Jonathon Emil Fleck, Philip K. McKinley & Betty H.C. Cheng. 2021. MoDALAS: Model-Driven Assurance for Learning-Enabled Autonomous Systems. In *Proc. 24th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2021)*, New York, NY: ACM.

[130] Langford, Michael Austin & Betty H.C. Cheng. 2021. Enki: A Diversity-Driven Approach

to Test and Train Robust Learning-Enabled Systems. *ACM Trans. Auton. Adapt. Syst.* 15(2). doi:10.1145/3460959.

[131] Lapouchnian, Alexei. 2005. Goal-Oriented Requirements Engineering: An Overview of the Current Research. Tech. rep. University of Toronto. http://www.cs.utoronto.ca/~alexei/pub/Lapouchnian-Depth.pdf.

[132] Legay, Axel, Benoît Delahaye & Saddek Bensalem. 2010. Statistical Model Checking: An Overview. In *Runtime Verification. Lecture Notes in Computer Science (RV 2010)*, vol. 6418, 122–135. Berlin, Heidelberg: Springer. doi:10.1007/978-3-642-16612-9_11.

[133] Lehman, Joel. 2012. *Evolution Through the Search for Novelty*: University of Central Florida dissertation.

[134] Lehman, Joel & Kenneth O. Stanley. 2011. Abandoning Objectives: Evolution Through the Search for Novelty Alone. *Evolutionary Computation* 19(2). 189–223. doi:10.1162/EVCO_a_00025.

[135] Letier, Emmanuel & Axel van Lamsweerde. 2004. Reasoning About Partial Goal Satisfaction for Requirements and Design Engineering. In *Proc. 12th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (SIGSOFT 2004/FSE-12)*, 53–62. New York: ACM. doi:10.1145/1029894.1029905.

[136] Letier, Emmanuel, David Stefan & Earl T. Barr. 2014. Uncertainty, Risk, and Information Value in Software Requirements and Architecture. In *Proc. 36th int. conf. on software engineering (icse 2014)*, 883–894. New York: ACM. doi:10.1145/2568225.2568239.

[137] Li, Zenan, Xiaoxing Ma, Chang Xu & Chun Cao. 2019. Structural Coverage Criteria for Neural Networks Could Be Misleading. In *Proc. 41st Int. Conf. on Software Engineering (ICSE-NIER 2019)*, 89–92. USA: IEEE. doi:10.1109/ICSE-NIER.2019.00031.

[138] Lin, Chung-Ling, Wuwei Shen, Steven Drager & Betty Cheng. 2018. Measure Confidence of Assurance Cases in Safety-Critical Domains. In *Proc. 40th Int. Conf. on Software Engineering: New Ideas and Emerging Results (ICSE-NIER 2018)*, 13–16. New York, NY: ACM. doi:10.1145/3183399.3183419.

[139] Lin, Tsung-Yi, Priya Goyal, Ross Girshick, Kaiming He & Piotr Dollár. 2017. Focal Loss for Dense Object Detection. In *Proc. IEEE Int. Conf. on Computer Vision (ICCV 2017)*, 2999–3007. USA: IEEE. doi:10.1109/ICCV.2017.324.

[140] Lindqvist, Jakob, Amanda Olmin, Fredrik Lindsten & Lennart Svensson. 2020. A General Framework for Ensemble Distribution Distillation. In *Proc. IEEE 30th Int. Workshop on Machine Learning for Signal Processing (MLSP 2020)*, 1–6. USA: IEEE. doi:10.1109/MLSP49062.2020.9231703.

[141] Liu, Li, Wanli Ouyang, Xiaogang Wang, Paul Fieguth, Jie Chen, Xinwang Liu & Matti Pietikäinen. 2018. Deep Learning for Generic Object Detection: A Survey. *Int J Comput Vis* 128. 261–318. doi:10.1007/s11263-019-01247-4.

[142] Liu, Ninghao, Mengnan Du, Ruocheng Guo, Huan Liu & Xia Hu. 2021. Adversarial Attacks and Defenses: An Interpretation Perspective. *SIGKDD Explor. Newsl.* 23(1). 86–99. doi:10.1145/3468507.3468519.

[143] Lundh, Fredrik, Alex Clark & Contributors. 2015. *Pillow (PIL Fork) 3.0x*. Read the Docs. https://pillow.readthedocs.io/en/3.0.x/about.html. (Revision 7729d889).

[144] Ma, Lei, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li & Yang Liu. 2018. DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems. In *Proc. 33rd ACM/IEEE Int. Conf. on Automated Software Engineering (ASE 2018)*, 120–131. New York, NY: ACM. doi:10.1145/3238147.3238202.

[145] Maksimov, Mike, Nick L. S. Fung, Sahar Kokaly & Marsha Chechik. 2018. Two Decades of Assurance Case Tools: A Survey. In *Proc. Computer Safety, Reliability, and Security (SAFECOMP 2018)*, 49–59. Cham: Springer.

[146] Malone, David. 2019. Rovers Set to Invade Construction Jobsites. https://www.bdcnetwork.com/rovers-set-invade-construction-jobsites.

[147] Mao, Ke, Mark Harman & Yue Jia. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proc. 25th Int. Symposium on Software Testing and Analysis (ISSTA 2016)*, 94–105. New York, NY: ACM. doi:10.1145/2931037.2931054.

[148] McMinn, Phil. 2011. Search-Based Software Testing: Past, Present and Future. In *Proc. 4th Int. Conf. on Software Testing, Verification and Validation Workshops (ICST 2011)*, 153–163. USA: IEEE. doi:10.1109/ICSTW.2011.100.

[149] Melenbrink, Nathan, Justin Werfel & Achim Menges. 2020. On-Site Autonomous Construction Robots: Towards Unsupervised Building. *Automation in Construction* 119. doi:10.1016/j.autcon.2020.103312.

[150] Menzies, Tim. 2020. The Five Laws of SE for AI. *IEEE Software* 37(1). 81–85. doi:10.1109/MS.2019.2954841.

[151] Microsoft. 2021. Responsible AI Principles from Microsoft. Website. Accessed: Aug 2021. https://www.microsoft.com/en-us/ai/responsible-ai.

[152] Moreno, Gabriel A., Javier Cámara, David Garlan & Bradley R. Schmerl. 2015. Proactive Self-Adaptation Under Uncertainty: A Probabilistic Model Checking Approach. In *Proc. 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*, 1–12. New York, NY: ACM. doi:10.1145/2786805.2786853.

[153] Morin, Brice, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey & Arnor Solberg. 2009. Models@ Run.Time to Support Dynamic Adaptation. *Computer* 42(10). 44–51. doi:10.1109/MC.2009.327.

[154] Muthusamy, Vinod, Aleksander Slominski & Vatche Ishakian. 2018. Towards Enterprise-Ready AI Deployments: Minimizing the Risk of Consuming AI Models in Business Applications. In *Proc. 1st Int. Conf. on Artificial Intelligence for Industries (AI4I 2018)*, 108–109. USA: IEEE.

[155] Nadareishvili, Irakli, Ronnie Mitra, Matt McLarty & Mike Amundsen. 2016. *Microservice Architecture: Aligning Principles, Practices, and Culture*. Sebastopol: O'Reilly Media.

[156] Naeem Irfan, Muhammad, Catherine Oriat & Roland Groz. 2013. Model Inference and Testing. *Advances in Computers* 89. 89–139. doi:10.1016/B978-0-12-408094-2.00003-5.

[157] Nair, Vinod & Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proc. 27th Int. Conf. on Machine Learning (ICML 2010)*, 807–814. Madison, WI: Omnipress.

[158] National Research Council. 2007. *Software for Dependable Systems: Sufficient Evidence?* Washington, DC: The National Academies Press. doi:10.17226/11923.

[159] Nekovee, Maziar, Sachin Sharma, Navdeep Uniyal, Avishek Nag, Reza Nejabati & Dimitra Simeonidou. 2020. Towards AI-enabled Microservice Architecture for Network Function Virtualization. In *Proc. 8th Int. Conf. on Communications and Networking (ComNet 2020)*, 1–8. USA: IEEE. doi:10.1109/ComNet47917.2020.9306098.

[160] Nguyen, Anh, Jason Yosinski & Jeff Clune. 2015. Deep Neural Networks Are Easily Fooled: High Confidence Predictions For Unrecognizable Images. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR 2015)*, 427–436. USA: IEEE. doi:10.1109/CVPR.2015.7298640.

[161] NIST. 2019. U.S. Leadership In AI: A Plan for Federal Engagement in Developing Technical Standards and Related Tools. Tech. rep. U.S. National Institute of Standards and Technology. https://www.nist.gov/system/files/documents/2019/08/10/ai_standards_fedengagement_plan_9aug2019.pdf. Accessed: Aug 2021.

[162] NTSB. 2017. Accident Report, NTSB/HAR-17/02. Tech. Rep. PB2017-102600 U.S. National Transportation Safety Board.

[163] NTSB. 2018. Preliminary Report, Highway. Tech. Rep. HWY18FH011 U.S. National Transportation Safety Board.

[164] NTSB. 2019. Highway Accident Report, Collision Between Vehicle Controlled by Developmental Automated Driving System and Pedestrian. Tech. Rep. NTSB/HAR-19/03 U.S. Na-

tional Transportation Safety Board. https://www.ntsb.gov/investigations/AccidentReports/Reports/HAR1903.pdf. Accessed: Aug 2021.

[165] NTSB. 2021. U.S. National Transportation Safety Board. Letter to Tesla. NEF-104. PE21-020. Accessed: Aug 2021. https://static.nhtsa.gov/odi/inv/2021/INIM-PE21020-84913P.pdf.

[166] NVIDIA Corporation. 2017. prepare_kitti_data.py. https://github.com/NVIDIA/DIGITS/blob/master/examples/object-detection/prepare_kitti_data.py.

[167] Object Management Group. 2020. Structured Assurance Case Metamodel (SACM) Version 2.1. Tech. rep. OMG. https://www.omg.org/spec/SACM.

[168] Odena, Augustus, Catherine Olsson, David Andersen & Ian Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proc. 36th Int. Conf. on Machine Learning (PMLR 2019)*, 4901–4911.

[169] Palmerino, J., Q. Yu, T. Desell & D. Krutz. 2019. Improving the Decision-Making Process of Self-Adaptive Systems by Accounting for Tactic Volatility. In *Proc. 34th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2019)*, 949–961. doi:10.1109/ASE.2019.00092.

[170] Papadopoulos, Petros & Neil Walkinshaw. 2015. Black-Box Test Generation from Inferred Models. In *Proc. 4th Int. Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2015)*, 19–24. USA: IEEE.

[171] Pei, Kexin, Yinzhi Cao, Junfeng Yang & Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proc. 26th Symposium on Operating Systems Principles (SOSP 2017)*, 1–18. New York, NY: ACM. doi:10.1145/3132747.3132785.

[172] Pontes, Felipe & Edward Curry. 2021. Cloud-Edge Microservice Architecture for DNN-based Distributed Multimedia Event Processing. In *Advances in Service-Oriented and Cloud Computing. ESOCC 2020. Communications in Computer and Information Science*, vol. 1360, Cham: Springer. doi:10.1007/978-3-030-71906-7_6.

[173] Python.org. 2022. Python 3.6.15 Documentation. Accessed: Jan 2022. https://docs.python.org/3.6.

[174] PyTorch.org. 2022. PyTorch Documentation. Website. Accessed: Jan 2022. https://pytorch.org/docs/stable/index.html.

[175] Qinbao Song, M. Shepperd, M. Cartwright & C. Mair. 2006. Software Defect Association Mining and Defect Correction Effort Prediction. *IEEE Trans. Softw. Eng.* 32(2). 69–82. doi:10.1109/TSE.2006.1599417.

[176] Quigley, Morgan, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler & Andrew Ng. 2009. ROS: An Open-Source Robot Operating System. In *Proc. Int. Conf. on Robotics and Automation Workshop on Open Source Software (ICRA Workshop 2009)*, USA: IEEE.

[177] Ramirez, Andres J. & Betty H. C. Cheng. 2011. Automatic Derivation of Utility Functions for Monitoring Software Requirements. In *Proc. 14th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2011)*, 501–516. Berlin, Heidelberg: Springer-Verlag.

[178] Ramirez, Andres J., Adam C. Jensen, Betty H. C. Cheng & David B. Knoester. 2011. Automatically Exploring How Uncertainty Impacts Behavior of Dynamically Adaptive Systems. In *Proc. 26th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2011)*, 568–571. USA: IEEE. doi:10.1109/ASE.2011.6100127.

[179] Ramos, Leonardo, Gabriel Divino, Guilherme Lopes, Breno de França, Leonardo Montecchi & Esther Colombini. 2019. The RoCS Framework to Support the Development of Autonomous Robots. *Journal of Software Engineering Research and Development* 7. 10:1–10:14. doi:10.5753/jserd.2019.470.

[180] Reeves, Colin R. 2000. Fitness Landscapes and Evolutionary Algorithms. In *Proc. Int. Conf. on Computer Safety, Reliability, and Security (SAFECOMP 2000)*, 3–20. Berlin, Heidelberg: Springer.

[181] Richter, Charles & Nicholas Roy. 2017. Safe Visual Navigation via Deep Learning and Novelty Detection. In *Proc. Robotics: Science and Systems*, 64–73. Cambridge, MA: MIT. doi:10.15607/RSS.2017.XIII.064.

[182] Rodriguez, Daniel, Roberto Ruiz, Jose C. Riquelme & Rachel Harrison. 2013. A Study of Subgroup Discovery Approaches For Defect Prediction. *Information and Software Technology* 55(10). 1810–1822. doi:10.1016/j.infsof.2013.05.002.

[183] ROS.org. 2019. Obstacle Avoidance and Robot Footprint Model. http://wiki.ros.org/teb_local_planner/Tutorials/Obstacle%20Avoidance%20and%20Robot%20Footprint%20Model.

[184] ROS.org. 2022. ROS Kinetic Kame Documentation. Accessed: Jan 2022. http://wiki.ros.org/kinetic.

[185] ROS.org. 2022. ROS Melodic Morenia Documentation. Accessed: Jan 2022. http://wiki.ros.org/melodic.

[186] Rushby, John. 2015. The Interpretation and Evaluation of Assurance Cases. Tech. Rep. SRI-CSL-15-01 Computer Science Laboratory, SRI International Menlo Park, CA. https://www.csl.sri.com/users/rushby/papers/sri-csl-15-1-assurance-cases.pdf.

[187] Russell, Stuart J. & Peter Norvig. 2010. *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ: Pearson Education 3rd edn.

[188] Rusu, Radu Bogdan & Steve Cousins. 2011. 3D is here: Point Cloud Library (PCL). In *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA 2011)*, 1–4. USA: IEEE. doi: 10.1109/ICRA.2011.5980567.

[189] Sabol, Patrik & Peter Sincak. 2018. AI Bricks: A Microservices-Based Software for a Usage in the Cloud Robotics. In *Proc. World Symposium on Digital Intelligence for Systems and Machines (DISA 2018)*, 207–212. USA: IEEE. doi:10.1109/DISA.2018.8490532.

[190] Saif, Irfan & Beena Ammanath. 2020. 'Trustworthy AI' Is A Framework To Help Manage Unique Risk. *MIT Technology Review* Artificial intelligence. https://www.technologyreview.com/2020/03/25/950291/trustworthy-ai-is-a-framework-to-help-manage-unique-risk/.

[191] Sallab, Ahmad, Mohammed Abdou, Etienne Perot & Senthil Yogamani. 2017. Deep Reinforcement Learning Framework for Autonomous Driving. In *Proc. IS&T Int. Symposium on Electronic Imaging, Autonomous Vehicles and Machines (AVM 2017)*, 70–76. Springfield, VA: IS&T. doi:10.2352/ISSN.2470-1173.2017.19.AVM-023.

[192] Sasaki, Yutaka. 2007. The Truth of the F-measure. Tech. rep. University of Manchester. https://www.cs.odu.edu/~mukka/cs795sum09dm/Lecturenotes/Day3/F-measure-YS-26Oct07.pdf.

[193] Schumann, Johann, Pramod Gupta & Yan Liu. 2010. *Application of Neural Networks in High Assurance Systems: A Survey*, vol. 268 Studies in Computational Intelligence (SCI). Berlin, Heidelberg: Springer.

[194] Segal, Shai, Avishy Carmi & Pini Gurfil. 2011. Vision-Based Relative State Estimation of Non-Cooperative Spacecraft Under Modeling Uncertainty. In *Aerospace Conference*, 1–8. doi:10.1109/AERO.2011.5747479.

[195] Sekhon, Jasmine & Cody Fleming. 2019. Towards Improved Testing for Deep Learning. In *Proc. 41st Int. Conf. on Software Engineering (ICSE-NIER 2019)*, 85–88. USA: IEEE. doi:10.1109/ICSE-NIER.2019.00030.

[196] Selvaraju, Ramprasaath R., Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh & Dhruv Batra. 2017. Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization. In *Proc. IEEE Int. Conf. on Computer Vision (ICCV 2017)*, 618–626. USA: IEEE. doi:10.1109/ICCV.2017.74.

[197] Sessions, Valerie & Marco Valtorta. 2006. The Effects of Data Quality on Machine Learning Algorithms. In *Proc. The Int. Conf. on Information Quality (ICIQ 2006)*, 485–498. Cambridge, MA: MIT.

[198] Shorten, Connor & Taghi M. Khoshgoftaar. 2019. A Survey on Image Data Augmentation For Deep Learning. *Journal of Big Data* 6(1). 1–48.

[199] Simon, Glen A., Jared M. Moore, Anthony J. Clark & Philip K. McKinley. 2018. Evo-ROS: Integrating Evolution and the Robot Operating System. In *Proc. Genetic and Evolutionary Computation Conf. Companion (GECCO 2018)*, 1386–1393. New York, NY: ACM. doi: 10.1145/3205651.3208269.

[200] Slingerland, Philip C. & Lauren H. Perry. 2021. A Framework for Trusted Artificial Intelligence in High-Consequence Environments. Tech. Rep. ATR-2021-01456 The Aerospace Corporation.

[201] Smith, Colin, Ewen Denney & Ganesh Pai. 2020. Hazard Contribution Modes of Machine Learning Components. Tech. rep. OSTI. https://www.osti.gov/biblio/1606667. (AAAI Workshop: SafeAI 2020).

[202] Sokolova, Marina & Guy Lapalme. 2009. A Systematic Analysis of Performance Measures for Classification Tasks. *Information Processing & Management* 45. 427–437. doi:10.1016/j.ipm.2009.03.002.

[203] Stone, James V. 2015. *Information Theory: A Tutorial Introduction*. Sebtel Press 1st edn.

[204] Strömgren, Oliver. 2018. *Deep Learning for Autonomous Collision Avoidance*. Linköping University MSc thesis.

[205] Sun, Pei, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, Vijay Vasudevan, Wei Han, Jiquan Ngiam, Hang Zhao, Aleksei Timofeev, Scott Ettinger, Maxim Krivokon, Amy Gao, Aditya Joshi, Yu Zhang, Jonathon Shlens, Zhifeng Chen & Dragomir Anguelov. 2020. Scalability in Perception for Autonomous Driving: Waymo Open Dataset. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR 2020)*, 2446–2454. USA: IEEE.

[206] Sun, Shengyang, Guodong Zhang, Jiaxin Shi & Roger Grosse. 2019. Functional Variational Bayesian Neural Networks. *CoRR* abs/1903.05779. Appeared in ICLR 2019.

[207] Sun, Youcheng, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill & Rob Ashmore. 2019. DeepConcolic: Testing and Debugging Deep Neural Networks. In *Proc. 41st Int. Conf. on Software Engineering (ICSE 2019)*, 111–114. USA: IEEE. doi:10.1109/ICSE-Companion.2019.00051.

[208] Sun, Youcheng, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska & Daniel Kroening. 2018. Concolic Testing for Deep Neural Networks. In *Proc. 33rd ACM/IEEE Int. Conf. on Automated Software Engineering (ASE 2018)*, 109–119. New York, NY: ACM. doi:10.1145/3238147.3238172.

[209] Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow & Rob Fergus. 2013. Intriguing Properties of Neural Networks. *CoRR* abs/1312.6199. Appeared in ICLR 2014.

[210] Tagasovska, Natasa & David Lopez-Paz. 2019. Single-Model Uncertainties for Deep Learning. In *Advances in Neural Information Processing Systems (NIPS 2019*, 6417–6428.

[211] Tian, Yuchi, Kexin Pei, Suman Jana & Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *Proc. 40th Int. Conf. on Software Engineering (NeurIPS 2018)*, 303–314. New York, NY: ACM. doi:10.1145/3180155.3180220.

[212] Tuncali, Cumhur Erkan, James Kapinski, Hisahiro Ito & Jyotirmoy V. Deshmukh. 2018. Reasoning about Safety of Learning-Enabled Components in Autonomous Cyber-Physical Systems. In *Proc. 55th Annual Design Automation Conf. (DAC 2018)*, New York, NY: ACM. doi:10.1145/3195970.3199852.

[213] U.S. Department of Defense. 2016. Report of the Defense Science Board Summer Study on Autonomy. Tech. rep. DoD.

[214] Walpole, Ronald E., Raymond H. Myers, Sharon L. Myers & Keying Ye. 2012. *Probability & Statistics For Engineers & Scientists*. Boston: Prentice Hall 9th edn.

[215] Walsh, William E., Gerald Tesauro, Jeffrey O. Kephart & Rajarshi Das. 2004. Utility Functions in Autonomic Systems. In *Proc. Int. Conf. on Autonomic Computing (ICAC 2004)*, 70–77. USA: IEEE.

[216] Weyns, D. & M. U. Iftikhar. 2016. Model-Based Simulation at Runtime for Self-Adaptive Systems. In *Proc. 15th Int. Conf. on Autonomic Computing (ICAC 2016)*, 364–373. doi: 10.1109/ICAC.2016.67.

[217] Weyns, Danny, Tom Holvoet, Kurt Schelfthout & Jan Wielemans. 2008. Decentralized Control of Automatic Guided Vehicles: Applying Multi-Agent Systems in Practice. In *Companion to the 23rd ACM SIGPLAN Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA Companion 2008)*, 663–674. New York, NY: ACM. doi:10.1145/1449814.1449819.

[218] Weyns, Danny, M. Usman Iftikhar, Didac Gil de la Iglesia & Tanvir Ahmad. 2012. A Survey of Formal Methods in Self-Adaptive Systems. In *Proc. 5th Int. C* Conf. on Computer Science and Software Engineering (C3S2E 2012)*, 67–79. New York, NY: ACM. doi: 10.1145/2347583.2347592.

[219] Weyns, Danny, Sam Malek & Jesper Andersson. 2012. FORMS: Unifying Reference Model for Formal Specification of Distributed Self-Adaptive Systems. *ACM Trans. Auton. Adapt. Syst.* 7(1). doi:10.1145/2168260.2168268.

[220] Weyns, Danny, Bradley Schmerl, Masako Kishida, Alberto Leva, Marin Litoiu, Necmiye Ozay, Colin Paterson & Kenji Tei. 2021. Towards Better Adaptive Systems by Combining MAPE, Control Theory, and Machine Learning. doi:10.1109/SEAMS51251.2021.00036.

[221] Whittle, Jon, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng & Jean-Michel Bruel. 2010. RELAX: A Language to Address Uncertainty in Self-Adaptive Systems Requirement. *Requirements Engineering* 15(2). 177–196. doi:10.1007/s00766-010-0101-0.

[222] Whittle, Jon, Peter Sawyer, Nelly Bencomo, Betty H. C. Cheng & Jean-Michel Bruel. 2009. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *Proc. 17th IEEE Int. Requirements Engineering Conf. (RE 2009)*, 79–88. Washington, DC: IEEE Computer Society. doi:10.1109/RE.2009.36.

[223] Williams, Grady, Paul Drews, Brian Goldfain, James M. Rehg & Evangelos A. Theodorou. 2016. Aggressive Driving with Model Predictive Path Integral Control. In *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA 2016)*, 1433–1440. doi:10.1109/ICRA.2016.7487277.

[224] Wing, Jeannette M. 2021. Trustworthy AI. *Commun. ACM* 64(10). 64–71. doi:10.1145/3448248.

[225] World Wide Web Consortium. 2020. Extensible Markup Language (XML) 1.0. Tech. Rep. REC-xml-20081126 W3C. https://www.w3.org/TR/xml/.

[226] Xie, Xiaofei, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin & Simon See. 2019. DeepHunter: A Coverage-Guided Fuzz Testing Framework for Deep Neural Networks. In *Proc. 28th ACM SIGSOFT Int. Symposium on Software Testing and Analysis*, 146–157. New York, NY: ACM. doi:10.1145/3293882.3330579.

[227] Xu, Zongben & Jian Sun. 2017. Model-driven Deep-Learning. *National Science Review* 5(1). 22–24. doi:10.1093/nsr/nwx099.

[228] Yerak, Becky & Tatyana Shumsky. 2019. More Companies Flag a New Risk: Artificial Intelligence. Wall Street Journal. Accessed: Aug 2021. https://www.wsj.com/articles/more-companies-flag-a-new-risk-artificial-intelligence-11547035202.

[229] Yosinski, Jason, Jeff Clune, Anh Nguyen, Thomas Fuchs & Hod Lipson. 2015. Understanding Neural Networks Through Deep Visualization. *CoRR* abs/1506.06579. Appeared in ICML 2015.

[230] Yu, Fuxun, Zhuwei Qin, Chenchen Liu, Liang Zhao, Yanzhi Wang & Xiang Chen. 2019. Interpreting and Evaluating Neural Network Robustness. In *Proc. 28th International Joint Conf. on Artificial Intelligence (IJCAI 2019)*, 4199–4205. ijcai.org. doi:10.24963/ijcai.2019/583.

[231] Zadeh, L.A. 1988. Fuzzy Logic. *Computer* 21(4). 83–93. doi:10.1109/2.53.

[232] Zagoruyko, Sergey & Nikos Komodakis. 2016. Wide Residual Networks. *CoRR* abs/1605.07146. Appeared in BMVC 2016.

[233] Zhang, Ji & Betty H. C. Cheng. 2006. Model-Based Development of Dynamically Adaptive Software. In *Proc. 28th Int. Conf. on Software Engineering (ICSE 2006)*, 371–380. New York, NY: ACM. doi:10.1145/1134285.1134337.

[234] Zhang, Ji, Heather Goldsby & Betty H. C. Cheng. 2009. Modular Verification of Dynamically Adaptive Systems. In *Proc. 8th Int. Conf. on Aspect-Oriented Software Development (ASOD 2009)*, 161–172. New York, NY: ACM. doi:10.1145/1509239.1509262.

[235] Zhang, Mengshi, Yuqun Zhang, Lingming Zhang, Cong Liu & Sarfraz Khurshid. 2018. DeepRoad: GAN-Based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *Proc. 33rd ACM/IEEE Int. Conf. on Automated Software Engineering (ASE 2018)*, 132–142. New York, NY: ACM. doi:10.1145/3238147.3238187.

[236] Zhu, Jun-Yan, Taesung Park, Phillip Isola & Alexei A. Efros. 2017. Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks. In *Proc. IEEE Int. Conf. on Computer Vision (ICCV 2017)*, 2242–2251. USA: IEEE.