NASA/TM–20240006865

Recommendations on Evidence and Process for Certification of Learning-enabled Components in Aerospace Systems

Adrian Agogino, Guillaume Brat, Yuning He, Daniel Hulse,
Rory Lipkis, and Thomas Pressburger
Ames Research Center, Moffett Field, California

Divya Gopinath, Lukman Irshad, Andreas Katis, Anastasia Mavridou,
Ganesh Pai, Corina Păsăreanu, Ivan Perez, and Johann Schumann
KBR, Inc.
Ames Research Center, Moffett Field, California

May 2024

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collection of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include creating custom thesauri, building customized databases, and organizing and publishing research results.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at ***http://www.sti.nasa.gov***
- E-mail your question via the Internet to help@sti.nasa.gov
- Fax your question to the NASA STI Help Desk at 443-757-5803
- Phone the NASA STI Help Desk at 443-757-5802
- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076–1320

NASA/TM–2024–20240006865

Recommendations on Evidence and Process for Certification of Learning-enabled Components in Aerospace Systems

Adrian Agogino, Guillaume Brat, Yuning He, Daniel Hulse,
Rory Lipkis, and Thomas Pressburger
Ames Research Center, Moffett Field, California

Divya Gopinath, Lukman Irshad, Andreas Katis, Anastasia Mavridou,
Ganesh Pai, Corina Păsăreanu, Ivan Perez, and Johann Schumann
KBR, Inc.
Ames Research Center, Moffett Field, California

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

May 2024

Acknowledgments

This work has been performed for the System-wide Safety (SWS) Project of the Airspace Operations and Safety Program (AOSP) within the Aeronautics Research Mission Directorate (ARMD). Errors in this report are those of the authors.

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an offical endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Summary

This report primarily identifies a collection of relevant and necessary evidence for assurance of machine learnt components (MLCs)—also known as learning-enabled components—integrated into aircraft systems, and gives preliminary suggestions on the elements of a certification process that invoke the identified evidence. The main focus is on feedforward neural networks that are static and trained offline through supervised learning. A brief background on the generic elements of the lifecycle of an MLC is given to contextualize the assurance considerations and, consequently, the evidence that is relevant and necessary to support certification.

At the level of an MLC, those considerations relate to: (i) the consistency and correctness of MLC contributions to system functions in the context of a validated functional intent; and (ii) the absence of MLC contributions to aircraft-level failure conditions. At an ML model level, confidence in model and data properties contribute to assurance of the containing MLC, in particular: (a) generalizability and robustness of models, in the presence of inputs not previously seen during training, disturbances to inputs, and unexpected inputs; and (b) valid data, i.e., data that are at least representative, relevant, complete, and accurate.

Evidence for the above spans the elements of the ML lifecycle, and includes, at a minimum, lifecycle artifacts that pertain to: (1) properties of requirements capturing functional intent, safety constraints, and aspects of the intended use and operating environment; (2) model performance, model complexity and design, and algorithm choice; (3) achievement of required performance at the levels of a trained model during model development, a trained model after model development is complete, and a trained model that is transformed into an executable equivalent; (4) model implementation aspects necessary for transforming a trained model into the executable equivalent; (5) integration of the executable trained model into the containing MLC, and eventually the larger system; and, (6) lastly, the verification and validation (V&V) of each of the above. Such V&V lifecycle artifacts themselves include: aspects of coverage, e.g., of various levels of requirements by the input space of the model and the data; traceability (where applicable); application of formal methods for property specification, analysis, and checking. Examples of evidence generation methods and tools further ground the discussion on what constitutes evidence, and the contribution to assurance during certification.

The identified assurance considerations and supporting evidence is not a comprehensive set. Additionally, neither what should be considered as sufficient evidence relative to the assigned criticality of an MLC, nor how criticality ought to be determined and adjusted, have been considered in this report. However, recommendations and suggestions are made for potential activities of the ML lifecycle that are aimed at providing confidence that an MLC can be relied upon when integrated into its containing (aircraft) system. Those activities are proposed as candidate elements of a certification process for MLCs. The main purpose of this report to inform regulatory guidance and consensus standards that may be used to meet the safety intent of the applicable regulations.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose and Scope

This report aims to inform and support both: (i) the US civil aviation regulator—the Federal Aviation Administration (FAA)—in their development of the appropriate guidance and regulations, and (ii) the wider aviation industry in their development of consensus guidelines and standards, for certification of aircraft containing systems implementing machine learning (ML). Towards those objectives, this report discusses what evidence may be relevant and necessary for assurance. Additionally it discusses where that evidence may be invoked in a lifecycle for ML, towards relating the evidence to a certification process.

The main application focus is airborne systems that integrate machine learnt functionality to deliver aircraft level functions. The main technical focus is on assurance of the machine learnt functionality—in particular *feedforward artificial neural networks* (ANNs) that are *static* and that have been developed using *supervised*, *offline* learning algorithms—from the standpoint of the artifacts developed using ML and their associated properties (Section 2.1 clarifies the italicized terms). This report does not discuss what constitutes sufficient assurance, commensurate with the safety criticality of an application, and the extent of evidence necessary for the same (though it is intended to be addressed as part of a future revision of the report).

1.2 Outline

This report is structured as follows: Section 2 introduces the terms used (Section 2.1), and gives a brief background on the general elements of an ML lifecycle (Section 2.2). Section 3 presents recommendations on evidence and a process in a narrative form: first, the aspects of assurance of ML functionality that need to be considered, at a minimum, are elaborated in Section 3.1; then Section 3.2 presents the kinds of relevant and necessary evidence for assurance of ML functionality, also summarizing the assurance considerations they address, referencing candidate evidence generation methods and tools (elaborated in Appendix A); lastly, in Section 3.3, the lifecycle aspects of ML functionality are related to the identified evidence, towards formulating the elements of a certification process. Section 4 concludes with an outline for further maturing the recommendations of this report.

2 Background

2.1 Terminology and Concepts

The following terms and concepts are relevant for this report: **assurance** is the provision of justified confidence that a product or service can be relied upon. **Evidence** refers to one or more *lifecycle artifacts* (the byproducts of a process, method, or tool) accompanied by verifiable *evidence assertions* (qualitative or quantitative statements of fact), which directly or indirectly provide confirmation that a product or service meets its requirements (and, therefore, can be relied upon). An **item** is a type of lifecycle artifact that is "a defined and bounded set of either (one or more) hardware elements or (one or more) software elements that are treated as a single entity for analytical purposes" [1].

A **machine learnt model** (MLM) is a mathematical formula constructed by applying learning algorithms to data. A **machine learnt component** (MLC) is a specialized item comprising one or more hardware and/or software implementations of one or more MLMs and their supporting functionality—usually *pre-processing* routines applied to the data that will be received by the models[1], and *post-processing* routines applied to the responses the models produce for the data they receive. This concept of MLC is compatible with similar concepts defined elsewhere [2,3], and is synonymous with *learning enabled component* (LEC) [4].

[1]Henceforth, unless otherwise indicated, this report will use *model* and *component* to mean an MLM and an MLC respectively.

An **MLC architecture** is a structure relating the models and supporting functionality of an MLC, which is selected to implement the allocated MLC requirements. An ML **model architecture** is a computational structure (such as a graph) selected to express a model. Usually, a model architecture is selected together with a learning algorithm.

An **artificial neural network** (ANN) is a specific type of model inspired by biological networks of neurons. The model architecture of an ANN, also known as a *neural architecture*, comprises *nodes* (artificial neurons) organized in *layers* (a collection of nodes that are siblings in a graph structured arrangement of nodes) interconnected by *edges* (links between nodes). Nodes and edges are associated with *biases* and *weights* respectively that, together with *activation functions*, transform *inputs* (numerical values from one layer) into *outputs* (numerical values at the next connected layer). *Deep neural networks* (DNNs) are a type of ANN with multiple *hidden* layers between the *input layer* (the first layer of an ANN) and the *output layer* (the last layer of an ANN). A **feedforward ANN** is a neural architecture (specifically, a directed acyclic graph) in which the information flow is unidirectional from its input layers through the hidden layers, if any, to the output layers.

Model parameters are the variables of an MLM established from data through the learning process, e.g., the weights and biases of an ANN. **Model hyperparameters** are those variables of the learning process (e.g., the number of training iterations) and the model (e.g., the number of hidden layers and activation functions) that are *not* learnt from data, but that are necessary to control how learning occurs and how a model is constructed respectively.

Learning algorithms used to build models may be broadly characterized as facilitating *supervised*, *unsupervised*, and *reinforcement* learning. **Supervised ML**, which is the focus of this report, is concerned with building models that infer a mathematical function from **labeled data**, i.e., data constituting correct examples of the function to be inferred, given in terms of its inputs and outputs. Usually, the intended use of such models is for prediction on inputs that were not amongst the examples used to build those models. Unsupervised and reinforcement learning are not further considered in this report.

Training data, also referred to as a *training set*, comprises those examples to which (supervised) learning algorithms will be applied to *train* (i.e., build) an ML model. **Validation data** comprises additional examples that are used to *tune* (i.e., configure) the model parameters of the MLM initially built using the training set. **Test data**, also known as a *test set*, consists of examples that are used to evaluate a trained and tuned MLM, against various *performance metrics* or criteria. When training, tuning, and testing have concluded, the result is a *completely specified* **inference model**, representing what will be implemented using hardware and/or software in an MLC. Section 3.1 gives additional terms and concepts related to data and ML models.

Offline learning, also termed as *batch learning*, occurs when a learning algorithm considers all the examples in a training set at once. **Online learning** occurs when the training examples are presented to a learning algorithm sequentially, and the training is incremental. A **static model** is an inference model whose parameter values and architecture do not change after offline learning, or after its subsequent implementation in an MLC and eventual use in a wider system, i.e., when it has been *deployed* into service.[2]

An **operational design domain** (ODD) for an MLM or an MLC is the (abstract) description of its respective input space, as captured in its requirements, for which its behavior is (to be) designed. More generally, an ODD is the allocated portion of a specification of the intended operating environment(s) of a given system—that includes all foreseeable operating conditions—for which that system is designed and in which it is expected to fulfill its missions [5]. An ODD can be characterized in terms of *ODD parameters*,

[2]In other words, a static model *does not continue to learn in use*. In principle, when developing a system including MLCs, either offline or online learning may be employed. Typically, it is convenient to perform offline learning first when a system is being developed, and then subsequently when it has been deemed necessary to modify the models/components based on data gathered in service, after a defined period of use, or a defined number of operations. In such cases, an initial static model will be replaced with another, new static model, developed through offline learning.

i.e., variables associated with the input space of an MLM/MLC.

2.2 General Elements of a Machine Learning Lifecycle

Various efforts have investigated the safety, development, assurance, and certification aspects of integrating MLCs into safety-critical systems [6,7], including in aviation [2,3,8–13]. Each of those efforts have put forth potential development lifecycles for ML, detailing to varying degrees the corresponding process objectives, inputs, outputs, and the underlying activities and flows. Based on and abstracting from those efforts, a lifecycle for an MLC comprises at least the following elements:

- operating context definition and requirements development (Section 2.2.1);
- data management (Section 2.2.2);
- model development (Section 2.2.3);
- model implementation and deployment (Section 2.2.4);
- verification and validation (V&V) activities (Section 2.2.5).

The elements above may also be seen as *stages* or, equivalently, *phases* of an ML lifecycle, with the exception of V&V activities, which are cross-cutting. That is, each element involves specific V&V activities applicable to one or more lifecycle artifacts related to that element (see Section 2.2.5 for more details).

The definition of a process and flow based on the above elements—for example, as the so-called *W process* [2,3,8], or a modified *V process* [12]—is not in scope for this report. Although, typically, such process flows involve a combination of sequential, concurrent, and/or iterative application of the generic elements. Note that the descriptions that follow are not comprehensive, and the intent is to give a background for: (i) the assurance considerations emerging from the use of ML, and (ii) the evidence relevant and necessary for addressing (some of) those concerns, together with suggestions for a certification process where that evidence is applicable.

2.2.1 Operating Context and Requirements Development

An established and well-understood practice in the development of an aircraft system is to describe the intended operating environment(s) for which that system is to be designed and in which it is expected to fulfill its missions. The system requirements, which capture that description, are developed following the conventional processes for (aircraft) system development [1] and safety assessment [14]. Those processes guide how the identified requirements are decomposed, refined, and allocated across the different levels of a system hierarchy, i.e., from (aircraft) *function* to *system*, *subsystem*, and eventually *item*.

As such, the development of an MLC commences from the (item level) requirements that it is allocated. MLC requirements specify the functionality needed, including the desired performance, behaviors, and the associated tolerances [9]. Additionally, MLC requirements include requirements on the intended operating environment (i.e., its ODD), the data necessary for learning, data quality [2, 3, 8, 11], characteristics of the underlying ML model, and the learning process (including model hyperparameters). Safety requirements allocated to an MLC reflect what is necessary to mitigate or otherwise control the contribution of an MLC to system-level failure conditions.

As with conventional items, the requirements for an MLC are analyzed for (internal) completeness, consistency, traceability to higher-level requirements, and validity. The analysis of MLC requirements, in conjunction with the constraints on its design, and operation can induce *derived requirements*, i.e., additional MLC requirements not directly traceable to higher-level parent requirements that result from (MLC) design choices [15]. Such requirements are analyzed with respect to their safety impact. Requirements development for an MLC together with defining its operating context inform the subsequent elements of an ML lifecycle.

2.2.2 Data Management

The capacity of an MLC to meet its requirements is in part critically dependent on the data used to develop it. Indeed, when an MLC must exhibit certain behaviors (as captured in its allocated requirements), the data used to build the underlying models must contain sufficient and valid examples of those behaviors. Conversely, when learning algorithms are applied to data, the behaviors that an MLC will exhibit will reflect what the training data encodes. Data management is thus broadly concerned with *curating* [9] data: that is, *acquisition*, *preparation*, and *maintenance* of valid and sufficient data for applying ML algorithms. Each of those in turn can involve specific additional activities.

For example, requirements are to be first defined on both the data to be collected [3, 9], and the quality attributes those data must exhibit [2, 8]. Subsequently, data collection can involve sampling of ODD parameters, as well as *augmentation* with data from synthetic sources, e.g., high-fidelity simulators. Following those, data may undergo various *preprocessing* steps including *cleaning* (e.g., removing or fixing values that are missing, duplicates, irrelevant, noisy, and inconsistent), and *labeling* (i.e., annotating the ground truth output), a prerequisite for supervised learning.

Thereafter, various transformations (e.g., normalization, encoding) may be employed to make data suitable for ML. When multiple data sources are involved, additional steps may be necessary for ensuring data consistency, and reduced complexity [6]. Data *segregation* allocates preprocessed data into training, validation, and test datasets. A so-called *verification dataset* may be considered additionally to confirm the behavior of the implementation of a learnt model, e.g., as executable code [9].

2.2.3 Model Development

In general, model development starts with model *selection* [6], also known as *paradigm selection* [9], *learning process management* [2, 8], and *model design* [3]. Effectively, a model architecture is to be chosen that best meets the allocated requirements and suits the type of problem being addressed (e.g., regression versus classification).

Where ANNs are the focus, model selection is concerned with the choice and refinement of specific neural architectures. Amongst the activities involved in this refinement are: selecting the training algorithm(s) that will be used for learning; determining the hyperparameters of the model and the chosen algorithm, including establishing the criteria to determine when training is complete; and defining so-called *loss functions* that are the basis to quantitatively evaluate how the model performs during training.

Model training applies the chosen learning algorithm to the training data, through which model parameters are determined. The validation data serves to confirm that a trained model performs as required. Multiple training and validation iterations are typical for tuning both model parameters and hyperparameters towards achieving the required model performance as specified in the allocated requirements. A specific aspect of tuning is *model optimization* [2, 3] that can involve changes to the model architecture. A specific form of model training is *transfer learning* [6], in which a previously trained model, possibly from a different application domain, is reused as a starting point to reduce the duration of training. When model training has concluded, the result is a fully specified model, also termed as an inference model [2, 8, 12]. It is this inference model that is subsequently implemented in software and/or hardware. Note that model training typically does not conclude without evaluation of a trained and tuned model on the test data, which is described later (Section 2.2.5) as part of the V&V activities of an ML lifecycle.

2.2.4 Model Implementation and Deployment

Model implementation alludes to the transformation of an inference model into a form that executes in its *deployment environment*, i.e., as software and/or target hardware of the airborne system [2, 3, 12, 13]. For the rest of the report, the resulting lifecycle artifact is termed as an *executable inference model*. Model deployment then refers to the integration of the executable inference model into the containing subsystem/system. It is worth noting that the inference model resulting from the conclusion of model training is also executable, albeit on the hardware and environment meant for the learning algorithms, which often differ from those of airborne systems.

Developing the executable inference model may involve additional optimizations and modifications to the model architecture, e.g., through *model pruning* to remove unused nodes, edges, and parameters, if any; and replacement of variables representing model parameters with the learnt constant values. Subsequent to these transformations, implementation in hardware and/or software follows conventional processes [15]. Likewise, integration of executable inference models into an MLC, and the subsequent integration of that MLC into its containing subsystem also follows the conventional systems development process [1].

2.2.5 Verification and Validation Activities

As mentioned earlier, V&V activities cut across all the elements of an ML lifecycle. Conventional processes are leveraged to confirm that the system requirements have been correctly allocated to MLCs, that the allocated MLC requirements are valid, i.e., reflect the functional intent, and that the requirements exhibit the usual properties of *completeness*, *consistency*, *unambiguity*, and *verifiability* [1, 2]. When considering ODD aspects of the allocated requirements, the *relevance*, *completeness*, and *accuracy* of the identified ODD parameters (including their respective admissible values) are amongst the foci of requirements validation [2, 3, 8, 10].

Given the central role of data for ML, key attributes of the data acquired are to be confirmed through the V&V activities related to data management. For example, confirmation is sought that data are relevant to and *representative* of both the intended operating environment as captured by the ODD, and the intended behavior as defined in the allocated requirements [2,3,9]. Additional data attributes that are assessed through data management related V&V include correctness or accuracy, completeness, *sufficiency*, and *balance* [2, 3, 6, 9]. Besides data attributes, V&V activities also aim to confirm that the activities performed during data management yield data that meet the applicable data requirements, including those on data format, resolution, timeliness, and integrity. V&V activities also apply based on the constraints emerging from a proper application of ML, e.g., ensuring that the data are correctly labeled, that the training and validation data are independent from the test data, and that the training process did not access the test data. Special emphasis is also given to verifying that the training, validation, and test data can be traced to the unsegregated data.

Model selection induces *model architecture validation*, also termed as *paradigm validation* [9] to confirm that a chosen model architecture can indeed meet the allocated requirements, and that it is appropriate for the type of problem it is meant to solve. Model verification, also called *learning process verification* [2, 8, 13] is an integral step in the training process, but distinct from model development. Specifically, it closes the training and tuning loop to assess whether or not a trained model meets model performance requirements based on the test data set. Those requirements usually encode desired properties of models such as *generalizability*, and *robustness* (see Section 3.1). Additional verification can involve *requirements-based test* to assess the extent to which the behavior of the trained model as exercised by test cases constructed from test data *cover* the allocated requirements.

The conclusion of model verification also concludes model development, producing an inference model that is ready for model implementation (as described in Section 2.2.4). Verification of the resulting executable inference model is aimed at confirming that (executable inference) model performance on the test data, and the associated properties of generalization and robustness have been preserved after model implementation activities. Verification of the integration of executable inference models in an MLC, and its subsequent integration into the containing subsystem follows conventional verification processes

2.3 Machine Learning Lifecycle Interactions

The progress of an MLC through its lifecycle stages occurs within the context of system and item level processes, e.g., those in [1, 14, 15]. The interactions between those processes and the ML lifecycle can impact the safety, development, assurance, and certification aspects of integrating MLCs into an aircraft system.

For example, the definition of MLC requirements and its ODD straddles the interface between system level development and safety assessment, and the data management element of an ML lifecycle (Section 2.2.2). Inadequate definition of the intended operating environment at a system level can affect ODD characterization of the MLC, which can subsequently impact data collection and, in turn, model performance. Conversely, model performance insufficiencies that result from the inherent limitations of ML algorithms—e.g., inferring bounded approximations from data samples, rather than the true required input-output relationship, and the infeasibility of perfect and complete sampling—can necessitate modifying the operating environment definition at the system level. That can potentially impact system requirements, safety assessment, and the system architecture.

Although such interactions are not inherently unique to ML, the reliance upon data introduces additional considerations. For instance, item requirements can be traced to system requirements for conventionally developed items. Such traceability contributes to assurance of, for example, requirements coverage, and also of item functionality not contributing to aircraft-level safety effects, in accordance with the safety objectives established for the containing function. When transitioning from the general, higher level requirements on the intended behavior of an MLC, to the data that encodes the lower level requirements on the behavior expected from the underlying ML models, traceability cannot be readily established in a similar way for certain kinds of functionality, e.g., perception. Moreover, model performance metrics that establish whether or not a model meets its allocated requirements (see Sections 2.2.3 through 2.2.5) also do not readily relate to safety objectives.

Similarly, model implementation and deployment (Section 2.2.4) is the interface to conventional hardware and software implementation and assurance processes. Insufficiencies in executable inference models in the target environments that are revealed when evaluating their performance on the associated test data used during model development and model verification necessitates re-entry of the ML lifecycle. Again, traceability from implementation to various levels of requirements cannot be leveraged for assurance of MLCs in the same way as it can for conventionally developed items. Moreover, V&V of the performance of executable inference models integrated into an aircraft system requires test data at various levels of the system to confirm that the functional intent has been preserved.

Configuration management (CM) and quality assurance (QA) processes are additionally paramount to ensure that the attributes and properties of MLCs are maintained through the ML lifecycle and the lifecycle of the containing system [3, 9, 16]. As such, those processes are tightly coupled with the ML lifecycle; though, this report does not further consider CM and QA aspects.

3 Suggestions and Recommendations

This section formulates suggestions and recommendations on evidence and process for certification of aircraft systems integrating ML, as an extended narrative that addresses the following:

- at a minimum, what aspects of assurance of an MLC need to be considered (Section 3.1);
- at a minimum, what constitutes evidence for assurance of an MLC, including from V&V activities, and what can be concluded regarding their contribution to assurance of an MLC (Section 3.2);
- in the context of the generic elements of an ML lifecycle (described earlier in Section 2.2), where is the evidence for assurance required or invoked, and its relation to a certification process (Section 3.3).

3.1 Assurance Considerations

Integrating and using an MLC in an airborne system that will be certified, necessitates assurance of its *fitness for purpose*. That is: (i) the contributions of an MLC to system/aircraft functions are correct and consistent with respect to a validated functional intent, and (ii) the system safety contributions are benign, i.e., an MLC does not cause or contribute to aircraft-level (failure) conditions that can result in harm. Those, in turn, rely upon assurance of the ML model(s) and the supporting functionalities that compose an MLC, the data used to develop the model(s), as well as the MLC architecture. This section focuses on and highlights the minimum assurance considerations from the perspective of desired ML model properties and the properties of the data used in their development.

Various prior efforts have identified additional assurance considerations beyond those given here, from the standpoints of certification, safety, attributes and properties of ML lifecycle artifacts as well as the processes producing those artifacts, and the applicability of the extant assurance guidelines for aircraft systems, software, and hardware [2, 6, 9, 10, 17, 18]. Assurance of both the supporting functionality in an MLC, and the MLC architecture leverage conventional processes [15] and are not considered further in this report.

3.1.1 Model Properties

Key properties of the inference model related to MLC fitness for purpose include generalizability—also called generalization capability [2, 3, 8]—and robustness.

Generalizability The capacity of an ML model to perform as required in use, on inputs from its ODD not previously encountered during its development is known as **generalizability**. This suggests the need for (i) defining and selecting the data that will be used in an ML lifecycle, and (ii) defining and assessing required model performance.

Due to the data-driven nature of ML and the inherent infeasibility of completely sampling a potentially infinite input space, model performance and consequently generalizability can only be empirically determined. Model performance is defined and assessed on the training, validation, and test data, using *performance metrics* appropriate for the type of problem being addressed (i.e., classification or regression). For supervised ML, model performance metrics usually quantify the *average error* on a given data set. Here *error* is a quantitative characterization of the disagreement between the response of the model to an input—often an approximation of the required response—and the true required response for that input. For example, a performance metric for regression problems is *mean squared error* (MSE), while for classification problems, *misclassification rate* is another performance metric [19]. What constitutes required performance is (to be) established based on a decomposition and refinement of higher-level requirements (see Section 3.2.4).

Generalizability is assessed in terms of the so-called **generalization gap**, a measure of the difference between model performance on the true distribution of its input space in use, and its performance on (the inputs in) the training data. Since the former is unknown, the test data is usually chosen as a surrogate. Both model performance and the generalization gap are then computed using the training and test data, first for a trained model during model development, then for an inference model during model verification, and subsequently for an executable inference model during model implementation and deployment. Here, an additional aspect of defining required model performance is to establish suitable bounds or tolerances on the generalization gap.

In the combined context of using test data as a surrogate of the intended operating environment, and bounding the generalization gap, a model whose performance does not violate those bounds as it progresses through the ML lifecycle (i.e., from trained model to inference model to an executable inference model) is expected to generalize. In other words, the model performance on the test data may approximate its performance in use. That, in turn, may be confirmed by gathering additional data in use, and re-assessing model performance.

As such, additional assurance considerations are induced on the data required for model development, in particular its properties relative to the intended operating environment (see Section 3.1.2). A key requirement for the test data that it is **independent** (I) from and **identically distributed** (ID) as the training data (i.e., so-called IID data).

Robustness This concept applies to a collection of related but subtly different notions, as considered in the context of software assurance, machine learning, and for integration of ML into aviation systems.

For example, in [9], robustness (for an MLC) is "bounded input-output sensitivity" for regression problems; for classification problems it is instead the constancy of the model response to a defined and bounded change to its expected inputs as captured by the data used during model development. Those two notions are similar to what is considered to be robustness (and also termed as *model stability*) in [8]: that is, "maintaining the input-output relations of a trained model", so that "small variations in the input yield small variations in the output".

In turn, the concept of robustness (equivalently, stability) from [8] is considered as *robustness related to generalizability* in [2]. Together with *robustness to adverse conditions*—i.e., when inputs are so-called *boundary cases*, *edge cases*, and/or *adversarial*—robustness in [2] is defined as "the ability of a system to maintain its level of performance under all foreseeable operating conditions". It is worth noting that this definition of robustness is well-aligned with robustness of conventional (non machine learnt) software [15], which is "the extent to which software can continue to operate correctly despite abnormal inputs and conditions". Robustness related to adverse conditions (as earlier), is the concept of model robustness adopted in [3], and it instead defines *model stability* as "the capacity of a model to preserve the intended responses, within specified tolerances under well-characterized and bounded perturbations to its inputs and operating conditions in its ODD".

Each of the preceding notions of robustness represent desiderata for ML models. As such, this report proposes the following notion of robustness: an ML model is **robust** when its intended behavior is maintained under bounded perturbation or variation in its input. This notion of robustness is compatible with the analogous concepts in in [2,3,8,9]. It extends generalizability on (previously unseen) inputs from the model ODD by admitting uncertainty or variability in both the model inputs and its responses. Thus, the distribution of the inputs need not be identical to that of the training data, e.g., due to *distribution shift*. For inputs not from the model ODD, a robust model response is one that does not violate the allocated requirement on intended behavior.

One aspect of defining required model performance for robustness—as with generalizability—is to define the bounds or tolerances on model input and response variability. An additional consideration is defining those regions of the input space where robustness is required. For instance, for a classification model, robustness is required for all inputs except at so-called decision boundaries (or surfaces for higher-dimensional input spaces) where it is (required to be) different.

A concept closely related to the robustness property of inference models is **out of distribution (OOD) performance**. As mentioned earlier, generalizability requires that the datasets for model development, implementation, and verification are IID. If the inputs to a model in use emerge from a distribution different from that of the training data, model performance is typically worse than what was measured in development. This is a common situation since the usage conditions are rarely exactly identical to those during development. Since an ODD specifies the input space for a model, and since the data for model development are to be sampled (or generated) according to that ODD, by definition, all model inputs consistent with the ODD are *in-distribution*. A *distribution shift* occurs when the usage conditions are sufficiently different from the ODD for the model, and what is subsequently represented in the training and validation data, e.g., when there is a difference in the time of day and weather conditions in use, from what they were when the data for training was collected. Thus, *out-of-distribution* inputs are those that the model encounters in use that it has encountered neither in the training data nor the test data. A critical data property related to this situation is *representativeness* (discussed in Section 3.1.2).

3.1.2 Data Properties

Data is central to developing an ML model, and it encodes the behaviors that are to be inferred through the application of learning algorithms. In effect, data can be seen as instances of the requirements on the behavior that a trained model must exhibit. As such, the characteristics of *valid*, *complete*, *consistent* requirements may be translated into properties of the data—including (but not limited to) *representativeness*, *relevance*, *completeness*, *accuracy*, *balance*, *bias*, and *validity*—which are described next. Beyond these, extant standards for processing aeronautical data [20] refer to data quality attributes such as *resolution*, *assurance level*, *traceability*, and *format*. Data properties related to those attributes are not further considered in this report.

Representativeness Data representativeness has been defined in various different (but compatible) ways: for example, in terms of: (i) the similarity of the distribution of the characteristics of the data and the model input state space [2]; (ii) *coverage* of ODD parameters and their respective ranges of values, together with the matching of the distributions of the characteristics of the data and ODD parameters [3]; and (iii) coverage of the foreseeable usage scenarios of the system integrating an MLC [9]. For this report, the following definition is considered: given a domain of interest—in this context, an operating environment, ODD (or input space), space of operating scenarios, or behaviors—that can be described as a statistical population, a dataset **representative** of that population is a subset that accurately reflects the characteristics of that population and the relative proportions of those characteristics.

Relevance Data used to develop an ML model are **relevant** for model development when the characteristics of the data relate to both the functional intent of the model (equivalently, its intended use) and the intended operating environment. In the context of ML model development, functional intent is reflected in the requirements allocated to the models, while their respective ODDs represent the intended operating environment. In [6], relevance of data is characterized in terms of the intersection of the dataset and the desired behaviors of a model in its specified operating environment. A related concept in [3] is *suitability*, which relates to whether or not data are appropriate for a specific purpose, and can satisfy so-called *data quality attributes* invoked in the data requirements formulated during the data management lifecycle stage (Section 2.2.2).

Completeness One notion of completeness of datasets used for model development is *sufficient coverage* of the model ODD [2, 3]. Thus, a key aspect of **completeness** for a dataset for model development is for the datapoints to map to the state space (the valid combinations of every independent characteristic) of the domain of interest (that is, as before, the operating environment, ODD, operating scenarios, or space of intended behaviors). The number of data points for each characteristic or combination thereof is determined by the joint distribution of those characteristics, e.g., for an ODD, the joint distribution of the combination of ODD parameters, each of which must be independent of every other parameter. As such, completeness of a dataset closely relates to its representativeness. This concept of completeness also may be considered as *internal*, since it assumes that the domain characteristics have themselves been completely identified. Additionally, some combinations of the identified domain characteristics may render data collection infeasible. Thus, considerations for data requirements include (i) in-depth domain modeling towards a comprehensive identification of the relevant domain characteristics and their interrelations; and (ii) specifying what constitutes *sufficient* data for a dataset to be considered complete and representative.

Accuracy The **accuracy** of data is usually given in terms of how close the (measured, sampled, or estimated) values of the characteristics of a domain are to their true values. Accuracy also relates to the retention of the trueness of data that has been transformed, processed, or augmented, e.g., by inclusion of data labels. The preceding aspects of data accuracy together with the degree to which the distribution of the data conforms to the distributions of the characteristics of the domain thus impact data representativeness. In other words, data that is representative of a population is also accurate and complete. An additional aspect of accuracy, which relates to data relevance, is fidelity with the intended use and functional intent. That is, data ought not to inadvertently encode unintended and undesired behaviors, e.g., through the use of a single source [8].

Balance and Bias For classification problems, a **balanced** dataset is one which contains equal or almost equal proportions of the classes comprising a valid response of a model. A **biased** dataset is one whose characteristics have distributions that are skewed, imbalanced, or otherwise have a systematic error, relative to those of the population from which the dataset was sampled. Thus, by definition, a biased dataset is not representative of the population/domain of interest. In general, the characteristics of a domain need not be balanced, so that some characteristics are rare (*minority* classes), while others are more prevalent (*majority* classes). Thus, when a dataset that samples from such a domain is balanced, it is biased; however such *data bias* is desirable for classification problems to mitigate the introduction of *model bias*, i.e., the systematic error in the model response, where a majority class is consistently and erroneously selected despite an input that should have resulted in a minority class as a response.

Validity Data that are representative and relevant, both of which, in turn, rely upon data being accurate and complete, are **valid**. Considering that the datasets used for model development embody the requirements on the intended behavior of a model, this intuitively aligns with the notion of validation of requirements [1], which is “the determination that the requirements for a product are correct and complete.”

3.2 Evidence

Recalling from Section 2.1, here *evidence* refers to ML lifecycle artifacts accompanied by qualitative or quantitative statements of fact (evidence assertions), which can be based on measurements, observations, and test/verification that can themselves be verified (i.e., verification of verification results). Collectively, they provide direct or indirect confirmation of, and thereby confidence in, the properties of an ML model and the associated data. That, in turn, contributes to assurance of the fitness for purpose of the containing MLC. Evidence concerning at least the following aspects are relevant and necessary for assurance of an MLC in support of approval/certification: system and operating context, requirements for ML (i.e., on the MLCs and associated models), data relevant for model development, ML performance, MLC and model architecture, and the artifacts produced from the associated V&V activities. Each of these aspects—which were inspired by FAA policy development efforts for ML [21]—is described next, elaborating on the minimum information that is likely to be necessary for certification of systems integrating ML components.

3.2.1 System and Operating Context

The system context for MLC development—i.e., the aircraft or system function to which it contributes, together with the characterization of its intended operating environment—are embodied in its allocated requirements. This section focuses on the operating environment as described by an operational design domain (ODD). The evidence from and related to MLC requirements is discussed subsequently (Section 3.2.2).

As indicated earlier (Section 2.1), an ODD gives an abstract description of the input space for an MLC and its underlying ML model(s) in terms of domain-specific concepts associated with ODD parameters. Defining and collecting the data necessary for ML relies upon a valid and complete ODD.

Establishing ODD coverage contributes to assurance of ODD validity and completeness, for example by showing that ODD parameters at a model level trace to those at the MLC level, and additionally to operating environment concepts referenced in the system requirements. Besides influencing component and model design, an ODD additionally serves as a reference against which to confirm the data properties, and the operating context when verifying an MLC implementation or validating the function inferred from learning. Additionally, systematic exploration of the ODD—analogous to hazard analysis conducted in conventional aircraft safety assessment processes [14]—can provide assurance of ODD validity and completeness.

Data-centric ODD Characterization A *data-centric ODD characterization* [5] contributes to determining the requirements necessary to drive ML model design, identifying the potential effects of MLC misbehavior on the (sub)system that contains it, whilst informing both architectural choices, and the assurance activities necessary for confidence that the learning process itself did not introduce errors. This characterization comprises the following data categories: *nominal*, *edge case*, *corner case*, *inlier*, *outlier*, *novelty*, and *singularity*. It also gives the following kinds of data: *in sample*, *out of sample*, *in-ODD*, and *out-of-ODD*, with the latter two qualified in terms of whether the data is relevant for the model, or its containing MLC.

Though the detailed definitions of the data categories and kinds of data is not in scope for this report, the following can give an intuitive understanding: a model should receive *in-ODD* inputs, which are data from the nominal, edge case, and corner case categories. The latter two are special types of inputs that represent the limiting conditions of the environment, described by the extrema of the ODD parameters values and their valid combinations. Additionally they may (but need not) be rare in terms of their occurrence frequency relative to the remainder of the *in-ODD* inputs. Nominal data are thus all *in-ODD* inputs that are not edge or corner cases. All *out-of-ODD* inputs are outliers. Novelty data are outlier inputs miscategorized as *in-ODD*, or vice versa. Thus, they are deficiencies in the ODD specification, while inliers are special cases of novelties resulting from errors in data management. Singularities are special *in-ODD* data where model behavior may not be (or is not required to be) robust. By construction, the data categories are complete, i.e., they address the full model input space.

Data that meets its properties, gathered from the data categories of a valid and complete ODD is thus necessary for and contributes to assurance of model robustness and generalizability. The combination of the categories and kinds of real data in operation, facilitates *partitioning* an ODD, and each such partition may be analyzed from a safety standpoint for the contribution of the model and its containing MLC to system hazards in terms of the effects produced in response to inputs drawn from that partition. Subsequently, the (high-level) requirements that an MLC should fulfill can be established, which can include, for instance, restrictions on model behavior, constraints on data processing, limitations of use, as well as requirements necessary to manage the safety impact of the identified effects. The latter, in turn, also informs the selection of the mitigation measures appropriate for sufficient safety assurance. Such mitigations include the application of learning assurance processes, architectural mechanisms, as well as traditional development assurance processes as appropriate.

Scenario-based Assessment Scenario-based assessment [22,23] gives an approach to explore and validate requirements that invoke ODD parameters, and thereby the ODD itself. A collection of scenarios describes the intended use of the system in the ODD. Conventional hazard analysis techniques are leveraged to construct the scenarios, and to establish failure conditions, their effects, as well as the impacts of variations in ODD parameters and events tied to ODD parameters. A distinction from conventional functional hazard assessment (FHA) and preliminary system safety assessment (PSSA) is the focus on operational use, which resides at a higher level than aircraft functions.[3]

Required performance at a system level can be decomposed and allocated to various elements of the scenarios which then can be confirmed through, first, an exploratory analysis in the early stages, and subsequently in system development and verification. Evidence from scenario-based assessment are safety perfor-

[3]It is noted that operational safety assessment generally resides at a higher level of analysis than what is considered in the prevailing aircraft safety assessment and system development standard practices [1, 14].

mance metrics tied to operational scenarios describing system use, through which performance thresholds tied to safety objectives can be confirmed, e.g., by aggregating metrics for each scenario and checking that in aggregate, they meet the set targets. This is inspired by aviation safety management systems (SMS) practice [24]. Although, such analysis relies upon additional confirmation that the assumptions made during scenario modeling are themselves valid relative to the functional intent and intended use. That, in turn, may rely upon simulation techniques providing evidence (and thereby assurance) of validity.

3.2.2 Machine Learning Requirements

This section explores the evidence that can be derived from activities related to the specification and analysis of requirements. The characteristic and common attributes of requirements for systems/components integrating ML has been explored in [25], and suggests that requirements of various kinds can be formulated; for example, requirements on intended functionality that may or may not include probabilities and confidence levels, and requirements invoking model and data properties (Sections 3.1.1, 3.1.2). Amongst those, some requirements are amenable to mathematically-based specification using structured natural language that have an underlying formal semantics [26, 27].

Requirements Formalization and Properties Formal specifications can provide assurance of fitness for purpose, when they are:

- *Complete*: all necessary and relevant system behaviors are specified;
- *Correct* (Valid): each requirement captures the right functional intent, i.e., the intended meaning;
- *Consistent*: the set of requirements contains no logical contradictions, i.e., the set of requirements are *satisfiable* for some set of inputs from the environment;
- *Realizable*: a strengthening of consistency, such that the set of requirements describes a system that is satisfiable for all assumed environment behavior. In other words, given any input from an environment that an MLC may receive, a system behavior or output exists such that all the requirements are met;
- *Verifiable* (or formally analyzable): unambiguously stated such that they can be verified, e.g., using (formal) static or dynamic analysis tools such as model checkers.

Evidence supporting claims that formally stated MLC requirements meet the preceding properties include the results of: (i) requirements verification activities using techniques for requirements-based testing, and formal verification, e.g., of requirements consistency, realizability, and model robustness; (ii) requirements validation activities, such as inspection, and simulation.

Formal Requirements Analysis *Consistency and realizability analysis* [28, 29] involves using formal methods to provide a mathematically sound confirmation as to whether or not a set of requirements is realizable (and, therefore, also consistent). The result of such an analysis is a `TRUE` or `FALSE` result, with the latter returning a counterexample. In general, realizability checking is a hard problem for MLCs including non-linearities, and compositional techniques are necessary to decompose a realizability check into smaller, more tractable analyses. For an example of compositional realizability analysis, see Appendix A.1.

Satisfiability modulo theory (SMT), and techniques built around SMT can be used to ensure *local robustness* of a model by providing mathematically-based analysis that, for a suitably-sized region around an input space point, the same model response will always be returned [6, 30–32]. Local robustness here refers to both so-called *adversarial* variations to inputs, and *natural perturbations*, representing stochastically occurring changes to the inputs, e.g., due to changes in the environment, and noise. *Global robustness* refers to network robustness for all inputs, which extends the concept of local robustness. SMT can be used for global robustness verification together with approaches that decompose global robustness as a property on an ML model into a collection of local robustness checks [33]. For additional details on such robustness verification techniques, see Appendix A.5

Testing and Coverage Testing is amongst the key steps to gaining confidence that a trained model is robust and generalizable. Indeed, the purpose of the test set is exactly to test a trained model (as well as the inference model and executable inference model) to gauge how it performs on data not previously seen during model training.

This notion of testing is different from *requirements-based testing* that uses a formal requirements specification attached to the code, which rather gives a coverage assessment of test cases. For instance, the so-called FLIP coverage metric [34] is a formally defined criterion to assess the extent to which a given test suite covers a requirement specified formally in temporal logic: for each Boolean constraint in a formally specified requirement with no logical operators, the metric generates an *obligation* to show how that constraint contributes to meeting the requirement. Then a test suite adhering to the set of obligations is devised. Intuitively, *full coverage* means that for each obligation, at least one test can be generated that demonstrates how that obligation is met and, in turn, how the requirement is met. Section 3.2.3 discusses the use of the FLIP metric for analysis of data quality and for data augmentation.

Additional notions of coverage relate requirements and ODD parameters: each MLC requirement references at least one ODD parameter, and all ODD parameters are referenced by at least one MLC requirement.

Requirements Validation Inspection and simulation of formal specifications of MLC behavior provides confidence that requirements are valid, i.e., encode the functional intent, according to the underlying semantics of the formalisms used. The extent of confidence in validity is dependent on the amount and diversity of simulation scenarios. For an example, see Appendix A.1.

Additionally, statistically based surrogate models can be used for validation of the safety envelopes and the thresholds stated in MLC requirements; for example, by identifying regions in the system state space where failures impact performance.

Formal verification of large inference models—both individually, and when integrated into its containing system—is constrained by model complexity and scale [35]. However, probabilistic abstractions built on trained model performance metrics, e.g., confusion matrices for classification problems, can be readily integrated into a system level model for abstraction-based *closed loop analysis* at a system level [36]. Such models can leverage conventional model-checking techniques [37], together with *assume-guarantee reasoning*. The results of such analyses can serve as evidence providing assurance that system-level requirements (in particular those that can be formalized in formal logic, such as probabilistic temporal logic) can be met with the inference model integrated into the system.

However, validation evidence of this form requires additional evidence, e.g., empirically gathered data, that confirm that the assumptions made in the surrogate statistical models are themselves valid and appropriate. For instance, that the data used to build the abstractions are indeed representative of the intended operating environment and usage, or that the abstractions are a faithful approximation of the learned models.

For more details, see Appendices A.2 and A.5.

3.2.3 Data for Machine Learning

Evidence that the datasets gathered for model development have the properties listed in Section 3.1.2 contributes to assurance of the properties of the learned model (Section 3.1.1) and that the containing MLC is fit for purpose.

For example, assurance of data validity can leverage, in part, verification that preprocessing has produced accurately labelled data, i.e., data labels that highlight what a learning algorithm should consider as ground truth during model training and testing, in fact tag the true instance of the domain concept; for example, if the data comprises still image frames each showing a *scene* of the landing environment, then region(s) labeled in each such frame as a runway indeed contain an instance of a runway image.

Coverage of the ODD by the data is evidence providing partial support for the data completeness and representativeness properties. That is, the data represents all ODD parameters and the full range of their admissible values, e.g., according to a metric such as a *goodness of fit* of the data distribution and the distributions specified in the ODD. In [13], for instance, a quantitative characterization of such coverage is formulated in terms of a *coverage ratio* which measures how much of the data in a dataset belongs to the input state space of a model as described by the combinations of ODD parameters.

Assurance in data completeness can be strengthened using the results of requirements-based testing through evidence assertions about coverage of requirements and through enabling data augmentation (i.e., adding programmatically generated data to the training, validation, and test data). Recall that the FLIP coverage metric (see Section 3.2.2) evaluates a test suite for its coverage of formal requirements, i.e., the extent to which a test suite covers all possible ways to satisfy a formal requirement.

In the context of data used for model development, each element of the dataset (i.e., a pair of inputs and outputs) can be considered as analogous to an execution trace, and then assessed against the obligations generated by the FLIP metric on formalized MLC (and system) requirements. Full coverage then means that the data set contains at least one element satisfying each generated obligation of the respective requirements.

For data augmentation, test cases can be considered as a sequence of input-output values defining, as before, an execution trace on which MLC behavior can be observed and evaluated. By model checking the obligations generated (on a formalized requirement) using the FLIP metric against a simplified system representation, e.g., a black box with only inputs and outputs specified, the counterexamples generated are the test cases that satisfy the obligations. In other words, including those test cases in the test suite achieves full requirements coverage. Since datasets for model development are effectively pairs of inputs and outputs, the test cases generated represent the data augmentations for the ML datasets.

3.2.4 Machine Learning Performance

The response/output of an ML model to an input is usually an approximation of the required response; the difference between the output produced and the required response is its *error*. By bounding the error in the requirements, model output can be evaluated as either being the required response (i.e., the error is within the stated bounds, and the response is correct), or not being the required response (i.e., the error exceeds the bounds, and the response is incorrect). Whenever the model output is not correct, it can be said to have *failed* (on that input).[4] Equivalently, when a model meets its allocated, validated requirements, it is considered as correct. When a model does not meet its allocated, validated requirements, it is considered as not correct. ML **model performance** is a quantification of the long term behavior of a given model, in terms of its error, according to a metric (or a set of metrics) relative to its requirements, e.g., as an *error distribution* for regression problems, or a *confusion matrix* for classification problems.

Surrogate statistical models (which are similar to ML models but are differentiated in that they are another mathematical abstraction of the functionality to be learned) can be used to explore both generalizability and robustness of a trained model (and subsequently the inference model and the executable inference model). For example, a white-box analysis of robustness in terms of the changes to model responses to defined changes to the model inputs can be visualized as a heat map, giving a visualization of robustness. Similarly, a white-box analysis of model generalizability can be examined by comparing the distributions of the errors in responses produced to those of the test data, and to other models. Appendix A.2 gives a detailed description of a specific framework for surrogate modeling through which assurance of model performance can be during model development.

Defining valid performance metrics is both problem and context dependent. It ties model properties, via requirements, to the aircraft/system function to which the model ultimately contributes; for example, clarifying how model performance is derived from, or contributes to system safety objectives. As such,

[4]This report adopts the definitions in [38], which provides a precise, unambiguous, and internally consistent taxonomy of dependability concepts, wherein a *failure* is a transition from correct service to incorrect service.

evidence supporting assurance of model performance includes, at a minimum, (i) analyses showing that model performance metrics (a) are relevant (i.e., suitable for the type of problem), (b) are related to the requirements of the containing MLC, as well as the higher-level system or function, (c) account for the MLC architecture, and (d) are valid (retains the intent of the higher-level requirements to which they are related); (ii) verification that the stated performance metrics are met by a trained model in model development, and then subsequently by the inference model and executable inference model in model implementation and deployment.

3.2.5 Architecture

Recalling (Section 2.1) that the concept of model architecture is different from that of MLC architecture, assurance from an architecture standpoint considers different levels of the system hierarchy.

Assurance of an ML model may require mechanisms such as *monitoring*, and the definition of a suitable internal configuration of the MLC that contains the model (i.e., MLC architecture). For instance, when a model cannot deliver its required performance on certain inputs (such as inputs that are out of its specified ODD), then reliance is placed on the MLC architecture that such out-of-ODD inputs will be filtered and never supplied to the model.

On the other hand, assurance of the ML model architecture relies upon verification that a trained model (also, subsequently, both the inference model and the executable inference model) meets not only the allocated requirements but also the constraints of the various activities in model development, i.e., selection and tuning of model and learning algorithm hyperparameters.

(Runtime) monitors can detect and respond during the execution/use of an MLC as to whether or not an applicable *property* (used here in the sense of a requirement or condition that evaluates to `TRUE` or `FALSE`) is valid. Effectively, monitoring provides a layer of protection against the propagation of MLC requirements violations. Monitors for MLCs need to address temporal and probabilistic properties, for instance when an MLC is used in responding to a sequence of inputs, or when an ML model response is given as a probability (for more details on methods and tools for monitoring see Appendices A.3 and A.4).

A common architectural pattern involving monitors is the **runtime assurance** (RTA) pattern [39] that defines a class of configurations (specifically a simplex architecture and its variations) involving monitors and switches organized around an MLC (or any other system/item for which there is insufficient assurance in design), and a so-called *assured fallback* or *backup function*. Though an RTA architecture employing runtime monitoring is mainly relevant in use (i.e., during deployment), runtime monitoring can also be used during MLC development, e.g., as oracles for testing. During unit or system testing, monitoring can aid in checking the validity of multiple requirements, and provide insightful information based on observing the internals of an inference model, i.e., the model parameter values and the results of activation functions within a neural network.

A key requirement for using an RTA architecture is for the monitors to be *unobtrusive*, i.e., there must be no impact or effect on the state of the system being monitored due to the monitor. Other characteristics for an RTA architecture have been elaborated in [39].

Formalized requirements (see Section 3.2.2 and Appendix A.1) can be used to generate *runtime monitors* during the deployment of an MLC. More generally, monitors can apply at all levels of the system hierarchy depending on the property being checked. For instance, (i) from requirements on inputs monitors can be generated to check whether inputs are within acceptable bounds before they are provided to the executable inference model; (ii) from requirements that describe assumptions on the environment, monitors can be generated that check whether or not the observed environment of the system matches the assumptions of an MLC and executable inference model; and (iii) monitors can be generated from requirements on model output to check that an MLC produces the correct reponses.

3.2.6 Verification

Model and Component Verification Section 3.2.1 has described coverage of ODD parameters against the higher-level description of the intended operating environment as one means of assuring ODD validity and completeness. Likewise, Section 3.2.2 has discussed various verification approaches for requirements stated at the system, MLC, and ML model levels, including formal techniques, testing, coverage, and simulation-based validation. Similarly, some approaches to verifying properties of the data used during ML have been discussed in Section 3.2.3, while Section 3.2.5 has discussed monitoring as an approach to architecture-centered assurance.

Additional verification approaches applicable to an inference model are described next, whose results contribute to assurance of model generalizability and robustness.

Neuron Pattern Based Property Checking Formal analyses based on *neuron patterns* define rules based on pre-conditions and post-conditions of a trained model. For example, in [30] a trained model is decomposed into a set of compact rules, which makes it more amenable for analysis. Specifically, given a model and a set of inputs, rules of the form `pre` $\Rightarrow$ `post` are mined.

Here, `post` is a desirable property of the output, such as a label being a certain class (for a classification problem), and `pre` represents a condition characterizing inputs on which the model displays desirable behavior with respect to `post`. The condition is in terms of neuron patterns, potentially capturing the logic of the model. Each rule has the form of an *input-output specification* of the functionality of the model, which enables verifying the model behavior beyond concrete inputs. Such analysis contributes to assurance of model generalizability.

Probabilistic Verification *Model counting based verification* can provide precise estimates of the long-run probabilities of output state occupancy. This approach provides an alternative to statistical methods that estimate model performance based on test data, which relies upon representativeness with respect to the intended operating environment and being IID as the training data. The idea is to perform the verification on the executable inference model using model checking together with model counting.

An example of this is [40], where model counting has been applied to assess correctness properties that have a bearing on safety, as well as generalizability and robustness. A trained model is translated into a C program that is analyzed with the C Bounded Model Checker (CBMC) [41] to produce a formula in Conjunctive Normal Form (CNF), which in turn is analyzed with state-of-the-art model counters to efficiently obtain precise counts with respect to different outputs, i.e., the relative frequencies of different outputs, providing a probabilistic characterization of state occupancy.

The verification results produced include: (i) an assessment of generalizability from a comparison of the probability of model outputs against the ground truth outputs, expressed as logical predicates (if available); (ii) a comparison of the performance of different ML models, possibly built using different algorithms; and (iii) a quantification of correctness and robustness of the trained models in terms of a probability that the analyzed property holds assuming an input distribution. These, in turn, contribute to assurance of model generalizability and robustness. For additional methods and tools supporting a probabilistic analysis of trained models, see Appendix A.5.

Traceability by Bridging the Semantic Gap from Model to Requirements Establishing traceability from system requirements (which reference high-level domain-specific concepts) to the implementation of an ML component (where the input space is higher dimensional and the inputs are raw data) is a challenge. Traceability of this form can contribute to assurance that an executable inference model completely implements the allocated requirements.

Extracting *task-specific features* from the internal representation of a neural network can potentially serve as a link to the system-level requirements [42]. Extracted feature representations enable verification of models with respect to system-level requirements. Specifically, the pre-condition in the requirements, expressed in terms of high-level features, can be translated into a constraint, pre', expressed over neuron values, by substituting the features with their corresponding representations. The modified requirement $\mathtt{pre}' \Rightarrow \mathtt{post}$ can be checked automatically using off-the-shelf verification tools.

Rule-based testing [43] contributes to assurance that an inference model completely implements its requirements. Using rules that abstract input-output behavior to drive testing the model can translate into adequate coverage of the rules, which have clear semantic meaning with respect to network behavior as compared to purely structural entities such as neurons. This approach supports fundamental testing tasks, such as: (i) automatically generating test oracles or ground truths for new (unlabelled) test inputs, and (ii) providing a meaningful coverage metric which can evaluate test suites for functional diversity, defect defection capabilities, and coverage of different input scenarios.

Verification in Integration Verification of a system integrating MLCs is challenging, since the integration and interactions with the intended environment create dynamics that are hard to model. Though, in some situations (e.g., where the criticality of a failure condition is low), it may be sufficient to use sample-based testing to statistically validate system behavior. Random testing is insufficient when an MLC may be well-trained. Testing that aims to probe the extreme and corner cases in an ODD, as well as other situations, e.g., edge cases and singularities (see Section 3.2.1), is necessary but can be incomplete. Domain expertise can help to bias tests towards certain system failure conditions, but a lack of independence between testing and development can lead to an undesired bias in the test results.

Adaptive stress testing (AST) [44] is a framework that uses ML for automatic discovery of weaknesses in MLCs for sequential decision making applications in simulation. AST assumes that the MLC runs deterministically within a system that may have stochastic environmental inputs, and it perturbs those inputs to elicit failures in the system under test. The results of AST serve as an *independent* black-box based alternative to complement other conventional testing techniques. Independence here is gained through an exploration of the system behavior based on a higher-level requirements specification rather than developer/tester knowledge to explore regions of the ODD of an MLC where there may be inadequate model performance.

3.3 Lifecycle Aspects

The model and data properties described in the preceding narrative (Section 3.1.1 and 3.1.2), as well as the evidence that provides assurance that those properties hold (Sections 3.2.1 – 3.2.6), can be associated with activities related to the general elements of an ML lifecycle (Section 2.2). Those activities, together with other processes, have been referred to collectively as *learning assurance* [2, 3, 8, 12, 13]. Learning assurance is the analog of so-called *development assurance* [45] but applied to the elements of an ML lifecycle.[5] That, in turn, can serve to inform what elements may constitute a certification process for ML components, analogous to how development assurance has been considered in current certification processes [46].

Operating Context and Requirements Development As discussed earlier (Section 3.1.1), model properties and their contribution to MLC fitness for purpose rely upon validated requirements that capture functional intent (of the aircraft/system function that invokes the MLC). ML model requirements include a description of the input space for a model, or its ODD (Section 2.2.1). That in turn is an allocation from a decomposition and subsequent refinement of a higher-level description of the intended operating environment of the containing system. Moreover, it is the ODD from which data are to be collected in accordance with the defined (and validated) data requirements (Section 2.2.2).

From the standpoint of an ODD used for characterizing the operating context of an MLC, it is thus im-

[5]Development assurance has also been termed as design assurance in the context of airborne software [15].

perative to have, at least: (i) a detailed characterization of the ODDs of both an ML model and its containing MLC in terms of ODD parameters, and their admissible values; (ii) Confirmation of sufficiency, completeness, and validity of the ODDs of the models and the MLCs, e.g., through a mapping from the model ODD to its input space, and traceability from the ODD to the data sampled from the intended operating environment.

From a requirements development standpoint, it is likewise necessary to at least: (i) formulate MLC and model performance requirements related to both functional and safety objectives; (ii) identify the bounds of acceptable MLC and model performance such that they correctly reflect the higher-level requirements, e.g., the bounds are such that failing to meet the allocated MLC requirements is less frequent than the safety target allocated to an MLC; (iii) have confirmation (i.e., V&V) of requirements validity, completeness, and consistency for model, data, and MLC requirements such that there is assurance that the intent of the functions or services being provided are met. Examples of techniques that may be leveraged here include formal methods, e.g., for realizability checking, simulation-based validation, or requirements-based test.

Data Management A core focus of data management is (as mentioned in Section 2.2.2) is ensuring that the data collected are representative, relevant, complete, balanced, and accurate (see Section 3.1.2). A deficiency of these data properties invariably contributes to a model whose performance is insufficient, i.e., it cannot meet its requirements, is not robust, and does not generalize. The various activities inherent to data management—including acquisition, preparation, maintenance, preprocessing, and allocation (or segregation) into training, validation, and test data—thus inherit the obligation to preserve the aforementioned data properties.

These obligations may be discharged through V&V activities that confirm the properties at each step in the data management. For example, assessment of various forms of coverage (e.g., of the data against the ODD, and MLC requirements) serve to provide confirmation of completeness and representativeness, as does traceability from data characteristics to ODD parameters. While for higher-dimensional data traceability is a known gap, for lower-dimensional input spaces traceability can be readily established from inputs to ODD parameters (e.g., for measurable physical and environmental parameters such as air temperature, airspeed, and altitude).

Similarly reducing data acquisition errors, such as those in measurement, sampling, and preprocessing, contributes to data accuracy, data completeness, and thereby also data validity. A critical related activity is ensuring that that the training, validation and test data individually meet the requirement of being IID with respect to each other *and* the intended operating environment.

Model Development The earlier elements of context formulation, requirements development, and data management are critical initial activities that set the foundation for developing performant ML models that, in turn, contribute to a fit for purpose MLC.

Model development builds on that foundation to iteratively select, train, and optimize a model that infers the intended behavior from the data to meet the allocated requirements. This overall process may be considered as a form of *systematized and partially mechanized trial and error*, where domain and subject matter expertise is usually leveraged to first *select* an initial model form (or architecture), learning algorithm[6], and to define the appropriate model performance metrics for evaluating the results of training.

Important considerations for model section include model complexity, model design, and out of distribution (OOD) performance.

Model Complexity This relates to choosing a model form and algorithm whose *degrees of freedom* are appropriate to the underlying statistics of the problem. Using an algorithm that is too simple for the target domain results in poor model performance. In contrast, an overly complex algorithm can model the data well but may not be able to generalize as required (i.e., *overfitting*).

[6]This reflects the state of the practice, although the state of the art has employed so-called *neural architecture search*, and *meta-learning* which attempts to *learn how to learn*. Those topics are not in scope for this report.

The so-called *bias-variance tradeoff* relates model complexity, predictive accuracy (i.e., the estimated total error of the model response to a future input not seen during its training), and its generalizability. It represents the situation where a model being built through supervised ML cannot simultaneously minimize both bias and variance. The *total error* in the response of a model is characterized in terms of model **bias** (a systematic error that measures the difference between the expected response of the model and the true response for a given input), and its **variance** (a component of error that measures the variation in the expected response of a model and its true response for minor variations of a given input). Thus, critical activities for model complexity management, and in turn, for model development include those targeted at reducing the total error (to a jointly optimal point).

Model Design and Interpretability Models that generate rules and those that make decisions directly from exemplars are two classes of algorithms that have enhanced *interpretability*, i.e., the logic of rule generation and decision making in the learnt model can be readily understood via human inspection. That, in turn, provides confidence that the models reflect the functional intent as captured in their allocated requirements, analogous to systematic inspection of a software architecture that embodies a design.

A popular algorithm in the first category is *decision trees* where the rules in the nodes of the tree can be inspected to see how they are processing data. In the second category, so-called *nearest neighbor algorithms* can tie a decision directly to a set of examples in the training set that were used. While these more interpretable algorithms can usually only handle problems of limited capacity, more complex algorithms can use portions of these more interpretable algorithms in key parts of their system. For instance a convolutional neural network can use a convolutional deep neural network to process inputs into a representation state, but can then use a nearest neighbor network to convert these representations into a decision.

Thus, developing a model architecture that leverages interpretable model forms and algorithms can additionally contribute to assurance that a model architecture is suitable to implement the allocated requirements.

Out of Distribution (OOD) Performance As previously discussed (Section 3.1.1), OOD performance impacts both model robustness and generalizability. In general, trained models may perform well on tests for data closely related to the training data, but perform poorly on data that is different (i.e., out of distribution). The contributors to OOD inputs to a model include:

- incomplete or underspecified ODD(s), i.e., corresponding to ODD parameters that were not included in an ODD, but should have been;
- deficient data acquisition (including errors in measurement, sampling, and preprocessing), i.e., corresponding to ODD parameters that were specified but were not sampled, inaccurately sampled, or sampled with measurement errors; and
- inadequacies or deficiencies in the MLC architecture, i.e., failure to filter inputs that should have been excluded (because they are not part of the functional intent, i.e., *out of (model) ODD* inputs.

Of these, the former two activities are within the scope of the earlier elements of the ML lifecycle (namely operating context development, and data management, respectively), whereas the third requires an architectural mechanism not in the scope of the model (also see Section 3.2), i.e., by defining the appropriate MLC architecture. As such, model development activities need to confirm that the input space for the chosen model form and algorithm are consistent with what is specified for that model in the MLC architecture.

Model training, tuning, and subsequent optimization to build a trained model is iterative after the selection of a model form and algorithm. Each iteration of training yields a model that is assessed against the validation set(s) for achieving the required performance in terms of generalizability and robustness. Additional aspects of this stage of model development include confirmation that iterative training and tuning never access the test data until after training has been deemed complete.

Model Implementation and Deployment Amongst the main objectives of model implementation is to ensure that the inference model that results from model development maintains its performance when transformed into an executable form. The learning process can mask certain flaws during that transformation into, for example, conventional software. For instance, the stochasticity in learning often yields models whose performance is not predictable (though it may nevertheless meet the requirements). These variations in performance may mask certain errors that can be difficult to reproduce.

Simulation-based testing is particularly susceptible to situations where learning algorithms may exploit modeling errors (affecting simulation fidelity), to yield performant models in test that may be insufficiently performant in use, especially when the datasets for model development contain augmented data produced in an algorithmic way. Thus it may be necessary for the test data—used to re-confirm the performance of an executable inference model—to contain samples from the intended real world environment to the extent possible. Additional verification steps are also likely necessary to gain assurance that both the functional intent of the allocated requirements, and the allocated safety objectives continue to be met. As mentioned in Section 3.2.6, confirmation activities can involve property checking (e.g., using neuron patterns), probabilistic approaches and surrogate models that verify integrated behavior, rule-based testing, and stress testing.

Verification and Validation Section 2.2 has indicated that V&V activites span the entire ML lifecycle. Indeed, the earlier discussions in this section have described various V&V activities (albeit not a comprehensive nor sufficient set) that serve to provide evidence for assurance of model properties, data properties, and in turn of MLC fitness for purpose. The verification of an executable inference model that results from model implementation, as well as verifying its integration both into the containing MLC and then into the higher-level system, leverage conventional assurance processes at the item-level and the system-level respectively [1, 14, 15].

4 Concluding Remarks

This report presents assurance considerations for machine learnt components (MLCs) and their underlying models, with suggestions for the minimum necessary and relevant evidence. Based on those, and based on the general elements of a lifecycle for MLCs, this report also suggests potential activities in that lifecycle aimed at providing confidence that an MLC can be relied upon when integrated into its containing (aircraft) system. Those activities are proposed as candidate elements of a certification process for MLCs.

The methodology underpinning the core outcomes, above, is based on a synthesis of: primarily, the collective research of the report authors, the literature available on contemporary research and practical efforts at integrating MLCs into aircraft systems; and—to a lesser but nevertheless appreciable extent—knowledge gained from technical discussions with some of the key stakeholders for this report: the FAA, and aviation industry consensus standardization bodies. As such, the suggestions and recommendations in this report can be further expanded, beyond its current content:

(i) The identified assurance considerations and supporting evidence is not a comprehensive set. Additionally, neither what should be considered as sufficient evidence relative to the assigned criticality of an MLC, nor how criticality ought to be determined and adjusted, were in scope for this report, though they are under consideration for a future revision of this work.

(ii) The rationale linking MLC and ML model assurance considerations, associated evidence, and associated activities is presently implicit and can be made explicit; attempts at crafting such a rationale do exist in other contemporary efforts [2, 3] though as part of the working knowledge of those efforts. A similar effort here can provide a common, concise basis for comparison, gap analysis, and harmonization.

(iii) Lastly, the suggestions and recommendations made can be adjusted, and refined based on an end-to-end application to concrete functionality that relies upon ML and is deployed into a real (but low criticality) aircraft system. Although the research that has informed this work has been applied to real examples, applying all the suggested techniques, tools, and suggestions to a common application will further enhance their credibility.

References

[1] S-18, Aircraft And System Development And Safety Assessment Committee, "Guidelines for Development of Civil Aircraft and Systems." Aerospace Recommended Practice ARP4754 Rev. B, Oct. 2023.

[2] European Union Aviation Safety Agency (EASA), "Guidance for Level 1 and 2 Machine Learning Applications." EASA Concept Paper Issue 02, March 2024.

[3] SAE G-34 Committee for AI in Aviation and EUROCAE WG-114 for AI, "Process Guidelines for Development and Certification/Approval of Aeronautical Safety-Related Products Implementing AI." Aerospace Recommended Practice ARP6983 / ED-324 - Work in Progress.

[4] E. Denney, R. Lee, G. Pai, and I. Šljivo, "QUASAR: Quantifiable Assurance Cases for Trusted Autonomy," Technical Report AFRL-RI-RS-TR-2023-162, Air Force Research Laboratory (AFRL), Sep. 2023.

[5] F. Kaakai, S. Adibhatla, G. Pai, and E. Escorihuela, "Data-centric operational design domain characterization for machine learning-based aeronautical products," in *Computer Safety, Reliability, and Security* (J. Guiochet, S. Tonetta, and F. Bitsch, eds.), (Cham), pp. 227–242, Springer Nature Switzerland, 2023.

[6] R. Ashmore, R. Calinescu, and C. Paterson, "Assuring the Machine Learning Lifecycle: Desiderata, Methods, and Challenges," *ACM Computing Surveys*, vol. 54, May 2021.

[7] R. Hawkins, C. Paterson, C. Picardi, Y. Jia, R. Calinescu, and I. Habli, "Guidance on the Assurance of Machine Learning in Autonomous Systems (AMLAS)." Assuring Autonomy International Programme Report version 1.1, March 2021.

[8] J. M. Cluzeau, X. Henriquel, G. Rebender, G. Soudain, L. van Dijk, A. Gronskiy, D. Haber, C. Perret-Gentil, and R. Polak, "Concepts of Design Assurance for Neural Networks (CoDANN)," Public Report Extract Version 1.0, European Union Aviation Safety Agency (EASA) and Daedalean AG, March 2020.

[9] AFE 87 Project Members, "AFE 87 - Machine Learning," Final Report 87-REP-01, Aerospace Vehicles Systems Institute, College Station, TX, June 2020.

[10] SAE G-34 Committee for AI in Aviation, "Artificial Intelligence in Aeronautical Systems: Statement of Concerns." AIR6988, April 2021.

[11] F. Kaakai, K. Dmitriev, S. Adibhatla, E. Baskaya, E. Bezzecchi, R. Bharadwaj, B. Brown, G. Gentile, C. Gingins, S. Grihon, and C. Travers, "Toward a Machine Learning Development Lifecycle for Product Certification and Approval in Aviation," *SAE International Journal of Aerospace*, vol. 15, no. 2, pp. 127–143, 2022.

[12] M. Gariel, B. Shimanuki, R. Timpe, and E. Wilson, "Framework for Certification of AI-Based Systems." arXiv: 2302.11049 [cs.LG], Feb 2023.

[13] G. Balduzzi, M. F. Bravo, A. Chernova, C. Cruceru, L. van Dijk, P. de Lange, J. Jerez, N. Koehler, M. Koerner, C. Perret-Gentil, Z. Pillio, R. Polak, H. Silva, R. Valentin, I. Whittington, and G. Yakushev, "Neural Network Based Runway Landing Guidance for General Aviation Autoland," Technical Report DOT/FAA/TC-21/48, Federal Aviation Administration, William J. Hughes Technical Center, New Jersey, November 2021.

[14] S-18, Aircraft And System Development And Safety Assessment Committee, “Guidelines and Methods for Conducting the Safety Assessment Process on Civil Aircraft, Systems, and Equipment.” Aerospace Recommended Practice ARP4761 Rev. A, Oct. 2023.

[15] RTCA SC-205 and EUROCAE WG-71, “Software Considerations in Airborne Systems and Equipment Certification.” DO-178C / ED-12C, Dec. 2011.

[16] SAE G-33 Configuration Management Committee, “Configuration Management Standard.” SAE ANSI/EIA-649C.

[17] G. Brat, H. Yu, E. Atkins, P. Sharma, D. Cofer, M. Durling, B. Meng, C. Alexander, S. Borgyos, C. Fan, and K. Garg, “Autonomy Verification & Validation Roadmap and Vision 2045,” Technical Memorandum NASA/TM-20230003734, NASA Ames Research Center, January 2023.

[18] Safety of Autonomous Systems Working Group (SASWG), “Safety Assurance Objectives for Autonomous Systems.” SCSC-153C Version 4.0, February 2024.

[19] K. P. Murphy, *Probabilistic Machine Learning: An Introduction.* MIT Press, 2022.

[20] RTCA SC-217 and EUROCAE , “Standards for Processing Aeronautical Data.” DO-200B / ED-76A, June 2015.

[21] J. Pastore, “AI and Machine Learning Policy Development,” in *FAA Artificial Intelligence Safety Assurance: Roadmap and Technical Exchange Meetings*, MITRE, March 2024.

[22] E. Denney and G. Pai, “Reconciling Safety Measurement and Dynamic Assurance,” in *Proceedings of the 43rd International Conference on Computer Safety, Reliability and Security (SafeComp) (to Appear)*, September 2024.

[23] L. Irshad and D. Hulse, “Resilience modeling in complex engineered systems with human-machine interactions,” in *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, vol. 86212, p. V002T02A024, American Society of Mechanical Engineers, 2022.

[24] Federal Aviation Administration (FAA), “Safety Management System.” Order 8000.369C, June 2020.

[25] M. Farrell, A. Mavridou, and J. Schumann, “Exploring requirements for software that learns: A research preview,” in *Requirements Engineering: Foundation for Software Quality* (A. Ferrari and B. Penzenstadler, eds.), (Cham), pp. 179–188, Springer Nature Switzerland, 2023.

[26] D. Giannakopoulou, T. Pressburger, A. Mavridou, J. Rhein, J. Schumann, and N. Shi, “Formal requirements elicitation with FRET,” in *Joint Proceedings of 26th International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2020) Workshops, Doctoral Symposium, Live Studies Track, and Poster Track* (M. Sabetzadeh, A. Vogelsang, S. Abualhaija, M. Borg, F. Dalpiaz, M. Daneva, N. Condori-Fernández, X. Franch, D. Fucci, V. Gervasi, E. C. Groen, R. S. S. Guizzardi, A. Herrmann, J. Horkoff, L. Mich, A. Perini, and A. Susi, eds.), vol. 2584 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2020.

[27] D. Giannakopoulou, T. Pressburger, A. Mavridou, and J. Schumann, “Automated formalization of structured natural language requirements,” *Information and Software Technology*, vol. 137, p. 106590, 2021.

[28] A. Katis, A. Mavridou, D. Giannakopoulou, T. Pressburger, and J. Schumann, "Capture, analyze, diagnose: Realizability checking of requirements in FRET," in *34th International Conference on Computer Aided Verification (CAV 2022)* (S. Shoham and Y. Vizel, eds.), vol. 13372 of *Lecture Notes in Computer Science*, pp. 490–504, Springer, August 2022.

[29] A. Mavridou, A. Katis, D. Giannakopoulou, D. Kooi, T. Pressburger, and M. W. Whalen, "From Partial to Global Assume-Guarantee Contracts: Compositional Realizability Analysis in FRET," in *24th International Symposium on Formal Methods (FM 2021)*, pp. 503–523, Springer, November 2021.

[30] D. Gopinath, H. Converse, C. S. Pasareanu, and A. Taly, "Property inference for deep neural networks," in *34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*, pp. 797–809, IEEE, November 2019.

[31] D. Gopinath, M. Zhang, K. Wang, I. B. Kadron, C. S. Pasareanu, and S. Khurshid, "Symbolic execution for importance analysis and adversarial generation in neural networks," in *30th IEEE International Symposium on Software Reliability Engineering (ISSRE 2019)* (K. Wolter, I. Schieferdecker, B. Gallina, M. Cukier, R. Natella, N. R. Ivaki, and N. Laranjeiro, eds.), pp. 313–322, IEEE, October 2019.

[32] C. Paterson, H. Wu, J. Grese, R. Calinescu, C. S. Pasareanu, and C. W. Barrett, "Deepcert: Verification of contextually relevant robustness for neural network image classifiers," in *40th International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2021)* (I. Habli, M. Sujan, and F. Bitsch, eds.), vol. 12852 of *Lecture Notes in Computer Science*, pp. 3–17, Springer, September 2021.

[33] D. Gopinath, G. Katz, C. S. Pasareanu, and C. W. Barrett, "Deepsafe: A data-driven approach for assessing robustness of neural networks," in *16th International Symposium on Automated Technology for Verification and Analysis (ATVA 2018)* (S. K. Lahiri and C. Wang, eds.), vol. 11138 of *Lecture Notes in Computer Science*, pp. 3–19, Springer, October 2018.

[34] C. Pecheur, F. Raimondi, and G. Brat, "A formal analysis of requirements-based testing," in *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, pp. 47–56, 2009.

[35] M. Baleani, A. Clavière, D. Cofer, E. DeWind, L. D. Guglielmo, O. Ferrante, G. Gentile, D. Kirov, D. Larsen, L. Mangeruca, S. F. Rollini, G. Cima, R. Schneider, H. Semde, and G. Soudain, "Formal Methods use for Learning Assurance (ForMuLA)," Technical Report version 1.0, Collins Aerospace and EASA, April 2023.

[36] C. S. Pasareanu, R. Mangal, D. Gopinath, S. G. Yaman, C. Imrie, R. Calinescu, and H. Yu, "Closed-loop analysis of vision-based autonomous systems: A case study," in *35th International Conference on Computer Aided Verification (CAV 2023)* (C. Enea and A. Lal, eds.), vol. 13964 of *Lecture Notes in Computer Science*, pp. 289–303, Springer, July 2023.

[37] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of Probabilistic Real-time Systems," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *LNCS*, pp. 585–591, Springer, 2011.

[38] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11–33, Jan 2004.

[39] G. Brat and G. Pai, "Runtime Assurance of Aeronautical Products: Preliminary Recommendations," Technical Memorandum NASA/TM-20220015734, NASA Ames Research Center, January 2023.

[40] M. Usman, D. Gopinath, and C. S. Pasareanu, “QuantifyML: How Good is my Machine Learning Model?,” in *Proceedings of the 3rd Workshop on Formal Methods for Autonomous Systems, (FMAS 2021)* (M. Farrell and M. Luckcuck, eds.), vol. 348 of *EPTCS*, pp. 92–100, October 2021.

[41] D. Kroening, P. Schrammel, and M. Tautschnig, “CBMC: The C Bounded Model Checker.” arXiv 2302.02384 [cs.SE], February 2023.

[42] D. Gopinath, L. Lungeanu, R. Mangal, C. S. Pasareanu, S. Xie, and H. Yu, “Feature-guided analysis of neural networks,” in *26th International Conference on Fundamental Approaches to Software Engineering (FASE 2023)* (L. Lambers and S. Uchitel, eds.), vol. 13991 of *Lecture Notes in Computer Science*, pp. 133–142, Springer, April 2023.

[43] M. Usman, Y. Sun, D. Gopinath, and C. S. Pasareanu, “Rule-based testing of neural networks,” in *Proceedings of the 1st International Workshop on Dependability and Trustworthiness of Safety-Critical Systems with Machine Learned Components* (M. Chechik, S. G. Elbaum, B. C. Hu, L. Marsso, and M. von Stein, eds.), pp. 1–5, ACM, December 2023.

[44] R. Lipkis, R. Lee, J. Silbermann, and T. Young, “Adaptive Stress Testing of Collision Avoidance Systems for Small UASs with Deep Reinforcement Learning,” in *AIAA SCITECH 2022 Forum*, 2022.

[45] S-18, Aircraft And System Development And Safety Assessment Committee, “Development Assurance Principles for Aerospace Vehicles and Systems.” Aerospace Informartion Report AIR7209, October 2022.

[46] AIR-134, Systems Integration Section, Federal Aviation Administration (FAA), “Airborne Software Development Assurance Using EUROCAE ED-12( ) and RTCA DO-178( ).” Advisory Circular AC 20-115D, July 2017.

[47] C. S. Pasareanu, R. Mangal, D. Gopinath, S. G. Yaman, C. Imrie, R. Calinescu, and H. Yu, “Closed-loop analysis of vision-based autonomous systems: A case study,” in *24th International Symposium on Computer Aided Verification (CAV 2023)* (C. Enea and A. Lal, eds.), vol. 13964 of *Lecture Notes in Computer Science*, pp. 289–303, Springer, July 2023.

[48] A. Mavridou, H. Bourbouh, D. Giannakopoulou, T. Pressburger, M. Hejase, P.-L. Garoche, and J. Schumann, “The Ten Lockheed Martin Cyber-Physical Challenges: Formalized, Analyzed, and Explained,” in *Requirements Engineering*, pp. 300–310, 2020.

[49] C. Elliott, “On example models and challenges ahead for the evaluation of complex cyber-physical systems with state of the art formal methods V&V, Lockheed Martin Skunk Works,” in *Safe & Secure Systems and Software Symposium (S5)*, Air Force Research Laboratory, July 2015.

[50] C. Elliott, “An example set of cyber-physical V&V challenges for S5, Lockheed Martin Skunk Works,” in *Safe & Secure Systems and Software Symposium (S5)*, Air Force Research Laboratory, July 2016.

[51] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, “NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking,” in *Proceedings of the International Conference on Computer-Aided Verification (CAV)*, vol. 2404 of *LNCS*, Springer, July 2002.

[52] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli, “The Kind 2 model checker,” in *International Conference on Computer Aided Verification (CAV)*, pp. 510–517, Springer, 2016.

[53] G. Hamon, B. Dutertre, L. Erkok, J. Matthews, D. Sheridan, D. Cok, J. Rushby, P. Bokor, S. Shukla, A. Pataricza, *et al.*, "Simulink Design Verifier: Applying Automated Formal Methods to Simulink and Stateflow," in *Third Workshop on Automated Formal Methods*, 2008.

[54] Y. He, "Online detection and modeling of safety boundaries for aerospace applications using active learning and bayesian statistics," in *2015 International Joint Conference on Neural Networks, IJCNN 2015, Killarney, Ireland, July 12-17, 2015*, pp. 1–8, 2015.

[55] M. A. Taddy, R. B. Gramacy, and N. G. Polson, "Dynamic trees for learning and design," *Journal of the American Statistical Association*, vol. 106, no. 493, pp. 109–123, 2011.

[56] R. Gramacy and N. Polson, "Particle learning of Gaussian process models for sequential design and optimization," *Journal of Computational and Graphical Statistics*, vol. 20, no. 1, pp. 467–478, 2011.

[57] Y. He, *Variable-length Functional Output Prediction and Boundary Detection for an Adaptive Flight Control Simulator*. PhD thesis, University of California at Santa Cruz, 2012.

[58] Y. He, H. Yu, G. Brat, and M. Davies, "Statistical learning framework for safety and failure analysis of a DNN-based autonomous aircraft system," in *Proc. International Conference on Machine Learning Applications (ICMLA)*, IEEE, 2021.

[59] ASTM International, "Standard Practice for Methods to Safely Bound Behavior of Aircraft Systems Containing Complex Functions Using Run-Time Assurance." ASTM F3269-21, November 2021.

[60] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks." arXiv: 1312.6199 [cs.CV], December 2013.

[61] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks," in *Proceedings of the 29th International Conference on Computer Aided Verification (CAV)* (R. Majumdar and V. Kunčak, eds.), vol. 10426 of *Lecture Notes in Computer Science*, pp. 97–117, Springer, July 2017.

[62] H. Wu, T. Tagomori, A. Robey, F. Yang, N. Matni, G. Pappas, H. Hassani, C. Pasareanu, and C. Barrett, "Toward certified robustness against real-world distribution shifts," in *2023 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, pp. 537–553, IEEE, 2023.

[63] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks." arXiv 1702.01135 [cs.AI], February 2017.

[64] I. B. Kadron, D. Gopinath, C. S. Pasareanu, and H. Yu, "Case study: Analysis of autonomous center line tracking neural networks," in *13th International Conference on Software Verification (VSTTE 2021)* (R. Bloem, R. Dimitrova, C. Fan, and N. Sharygina, eds.), vol. 13124 of *Lecture Notes in Computer Science*, pp. 104–121, Springer, October 2021.

[65] H. Converse, A. Filieri, D. Gopinath, and C. S. Pasareanu, "Probabilistic symbolic analysis of neural networks," in *31st IEEE International Symposium on Software Reliability Engineering (ISSRE 2020)* (M. Vieira, H. Madeira, N. Antunes, and Z. Zheng, eds.), pp. 148–159, IEEE, October 2020.

[66] M. Usman, Y. Sun, D. Gopinath, R. Dange, L. Manolache, and C. S. Pasareanu, "An overview of structural coverage metrics for testing neural networks," *International Journal of Software Tools and Technology Transfer*, vol. 25, no. 3, pp. 393–405, 2023.

[67] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety Verification of Deep Neural Networks," in *Computer Aided Verification (CAV 2017)* (R. Majumdar and V. Kunčak, eds.), vol. 10426 of *Lecture Notes in Computer Science*, Springer, July 2017.

[68] J. Kim, R. Feldt, and S. Yoo, "Guiding Deep Learning System Testing using Surprise Adequacy." arXiv:1808.08444 [cs.SE], August 2018.

[69] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Structural test coverage criteria for deep neural networks," *ACM Transactions on Embedded Computing Systems (TECS)*, 2019.

[70] C. S. Pasareanu, R. Mangal, D. Gopinath, and H. Yu, "Assumption generation for learning-enabled autonomous systems," in *23rd International Conference on Runtime Verification (RV 2023)* (P. Katsaros and L. Nenzi, eds.), vol. 14245 of *Lecture Notes in Computer Science*, pp. 3–22, Springer, October 2023.

[71] M. Usman, D. Gopinath, Y. Sun, and C. S. Pasareanu, "Rule-based runtime mitigation against poison attacks on neural networks," in *Proceedings of the 22nd International Conference on Runtime Verification (RV 2022)* (T. Dang and V. Stolz, eds.), vol. 13498 of *Lecture Notes in Computer Science*, pp. 67–84, Springer, 2022.

A Methods and Tools Supporting Assurance of Machine Learning

A.1 Formal Requirements Elicitation Tool (FRET)

The characteristic and common attributes of requirements on systems and components integrating ML has been explored in [25], and suggests that requirements of at least the following kinds can be formulated:

- conventionally stated requirements that *do not always involve probabilities*, despite ML models displaying behavior and responses which can be described in a probabilistic manner. Extant techniques for requirements development and verification can be employed for such requirements. For example,

  The ML model shall produce as output a sensible angle between $-90°$*and* $+90°$

- requirements that can be formulated as logical statements containing probabilities. For instance,

  The minimum accuracy of the ML model on the training set shall be $X\%$*; and on the test set shall be* $Y\%$

- confidence levels or probabilities on requirements, giving quantitative bounds on the uncertainty in a verification of the requirement or the accuracy of a verification procedure for a requirement. For example,

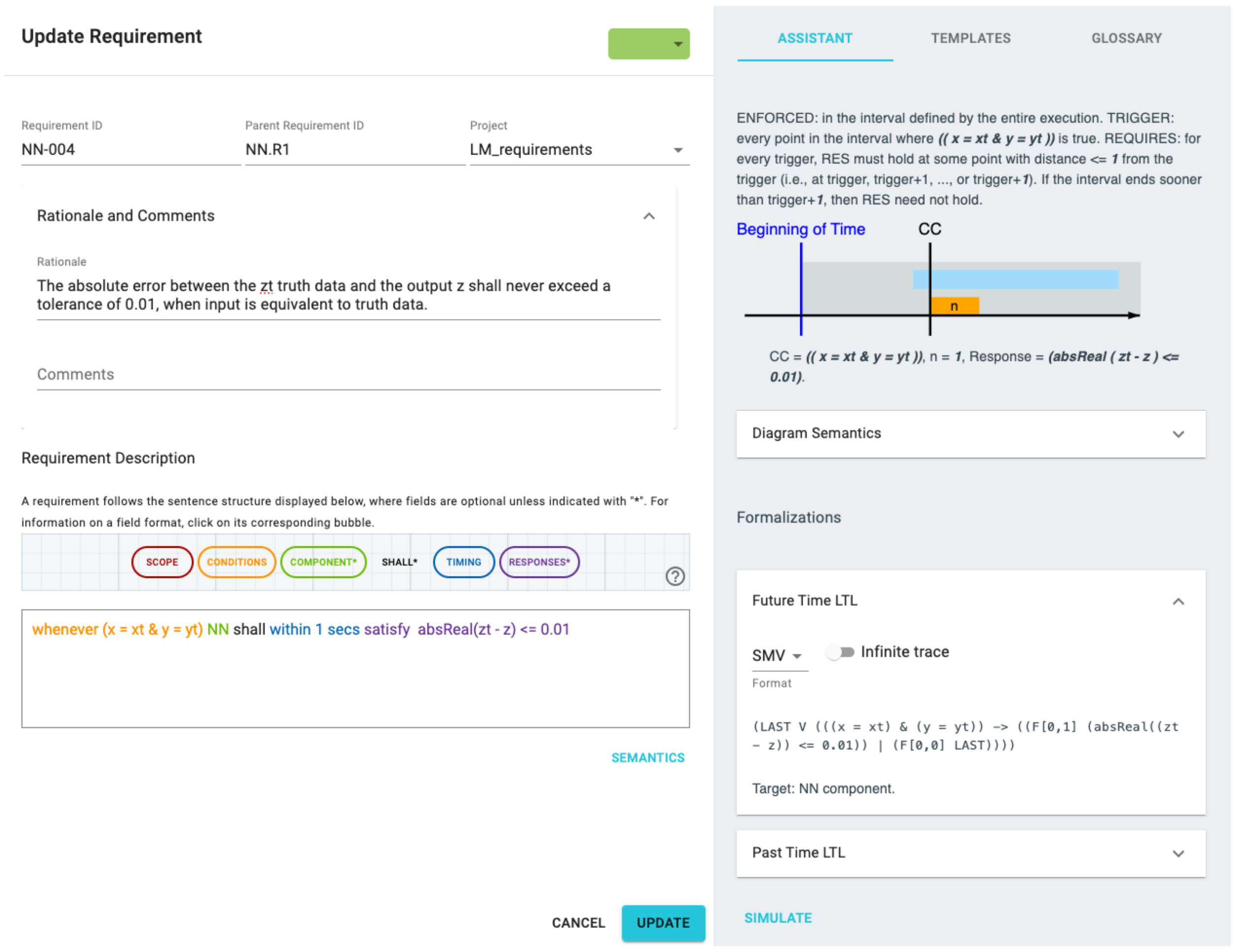
  The cross track error shall not exceed $\pm X$ *with confidence* $Y\%$

ML requirements often rely on probabilities to represent uncertainty. There are two important challenges associated with Requirements Engineering (RE), which are aggravated by the use of probabilities. First, RE is an inherently human-centric activity and so the language chosen to specify requirements is important for usability and interpretation of requirements. To this end, developers typically write requirements in intuitive natural language, which however can be very ambiguous. Adding probabilities only exacerbates the existing ambiguity issue. The second challenge is related to connecting requirements with formal analysis tools. A variety of analysis tools have been developed for specifications written in probabilistic temporal logics, such as model checking and runtime monitoring. However, expecting developers to directly write requirements in such complex specification formalisms is error prone or even unrealistic. The Formal Requirements Elicitation Tool (FRET) aims to tackle these two challenges. FRET is an open source tool for writing, understanding, formalizing, and analyzing requirements [26, 27]

Table 1. Requirements Formats and Characteristics

Requirements Format	Example	Characteristics
Natural Language	The absolute error between the zt truth data and the output z shall never exceed a tolerance of 0.01, when input is equivalent to truth data	Intuitive, ambiguous, not amenable to formal analysis
FRETISH (Restricted Natural Language)	Whenever $(x = xt \& y = yt)$ NN shall within 1 seconds satisfy absReal$(zt - z) \leq 0.01$	Intuitive, unambiguous
Temporal Logic Formula	`(LAST V (((x = xt) & (y = yt)) -> ((F[0,1] (absReal((zt - z)) <= 0.01)) | LAST)))`	Unintuitive, unambiguous, automatically generated from FRETISH, amenable to formal analysis

A.1.1 FRET Capabilities


Update Requirement

Requirement ID: NN-004
Parent Requirement ID: NN.R1
Project: LM_requirements

Rationale and Comments

Rationale: The absolute error between the zt truth data and the output z shall never exceed a tolerance of 0.01, when input is equivalent to truth data.

Comments

Requirement Description

A requirement follows the sentence structure displayed below, where fields are optional unless indicated with "*". For information on a field format, click on its corresponding bubble.

whenever (x = xt & y = yt) NN shall within 1 secs satisfy absReal(zt - z) <= 0.01

ENFORCED: in the interval defined by the entire execution. TRIGGER: every point in the interval where ***((x = xt & y = yt))*** is true. REQUIRES: for every trigger, RES must hold at some point with distance <= ***1*** from the trigger (i.e., at trigger, trigger+1, ..., or trigger+***1***). If the interval ends sooner than trigger+***1***, then RES need not hold.

CC = ***((x = xt & y = yt))***, n = ***1***, Response = ***(absReal (zt - z) <= 0.01)***.

Format: SMV — Infinite trace

```
(LAST V (((x = xt) & (y = yt)) -> ((F[0,1] (absReal((zt
- z)) <= 0.01)) | (F[0,0] LAST))))
```

Target: NN component.

Figure 1. FRET requirements editor

Requirements Elicitation Users of FRET write requirements in an intuitive, restricted natural language, called FRETISH, with precise, unambiguous meaning (see Table 1) using FRET's requirement elicitation interface (Figure 1). As shown in the figure, the FRETISH requirement **[NN-004]**: "*whenever* $(x = xt)$ *&* $(y = yt)$ *NN shall within* 1 *seconds satisfy* $\text{absReal}(zt - z) \leq 0.01$" expresses the natural language description for the neural network (NN) included in the "Rationale and Comments" field.

FRETISH requirements have up to six fields: scope, condition, component, timing and response. To specify and reason about uncertainty, a probability field has been recently added, that specifies a probability bound on the requirement. To this end, FRET can be used to specify non-probabilistic requirements as well as probabilistic requirements.

Tables 2, 3 and 4 contain examples of requirements at a system, MLC, and ML model levels as specified in FRETISH.

Table 2. Examples of System Requirements stated in FRETISH

Natural Language Requirement	FRETISH Requirement
The probability that the aircraft leaves the taxiway, i.e., $\lvert \texttt{cte} \rvert > 8$ meters, shall be extremely low [47]	The aircraft shall with probability $<= 0.001$ eventually satisfy $\mathrm{absReal}(\texttt{cte}) > 8$
The probability that the aircraft turns more than a prescribed degree ($\lvert \texttt{he} \rvert \leq 35°$) shall be extremely low [47]	The aircraft shall with probability ≤ 0.002 eventually satisfy $\mathrm{absReal}(\texttt{he}) \leq 35$
When aircraft is near the right border of the runway, the aircraft shall not move further to the right ($\texttt{he} > 0$) for an extended time.	Whenever $\texttt{he} >$ limit the aircraft shall for T seconds satisfy $dhe\_dt <= 0$
The aircraft shall reach the end of the runway within T seconds.	The aircraft shall within T seconds satisfy endOfRunway

Table 3. Examples of MLC Requirements stated in FRETISH

Natural Language Requirement	FRETISH Requirement
Upon receiving an image, the component shall output the cross track error within T seconds with a high probability	Upon imageReceived the component shall with a probability ≥ 0.98 within T seconds outputCTE
If DNN1 is active and a camera obstruction occurs in a sensitive area, DNN2 shall be activated.	The aircraft shall with probability $<= 0.002$ eventually satisfy $\mathrm{absReal}(\texttt{he}) <= 35$
When aircraft is near the right border of the runway, the aircraft shall not move further to the right ($\texttt{he} > 0$) for an extended time.	whenever active(DNN1) & obstructed component `shall` immediately satisfy active(DNN2) Whenever active($\texttt{DNN1}$) & obstructed the aircraft shall immediately satisfy active$\texttt{DNN2}$
The MLC shall receive an image frame every 0.2 seconds (0.1 second clock rate) and over the duration of 1 second shall correctly classify at least half of the received image frames.	Two FRETISH requirements are necessary for formalization: (1) The MLC shall every 2 ticks satisfy receivedImage; (2) The MLC shall every 5 ticks satisfy $\mathrm{correctlyClassifiedImages} > 2$

Table 4. Examples of ML Model Requirements stated in FRETISH

Natural Language Requirement	FRETISH Requirement
The DNN output should not fluctuate	The DNN shall always satisfy $\mathrm{absReal}(\mathrm{output} - \mathrm{preReal}(\mathrm{output})) <= \mathrm{theta}$

Table 4. Examples of ML Model Requirements stated in FRETISH (Continued)

Natural Language Requirement	FRETISH Requirement
The maximum value of the NN output, z, shall always be less than or equal to 1.1, regardless of the input values [48]	The NN shall always satisfy $z <= 1.1$
The absolute error between the zt truth data and the output z shall never exceed a tolerance of 0.01, when inputs x, y are equivalent to truth data xt, yt [48]	Whenever $(x = xt \,\&\, y = yt)$ NN shall within 1 seconds satisfy $\mathrm{absReal(zt - z)} <= 0.01$

From properties examined so far, data requirements—e.g., whether a training set reflects completeness over a real environment (such as lighting, weather, terrain)—usually are not formal properties. In the future, different types data requirements shall be considered in more detail. If these can be formulated as formal properties, then they can be captured by FRET.

Validation of Semantics Through Explanations and Simulation Getting a requirement with temporal relationships right is a tricky and subtle task. FRET produces natural language and diagrammatic explanations of a requirement's exact meaning (see the ASSISTANT tab in Figure 1).

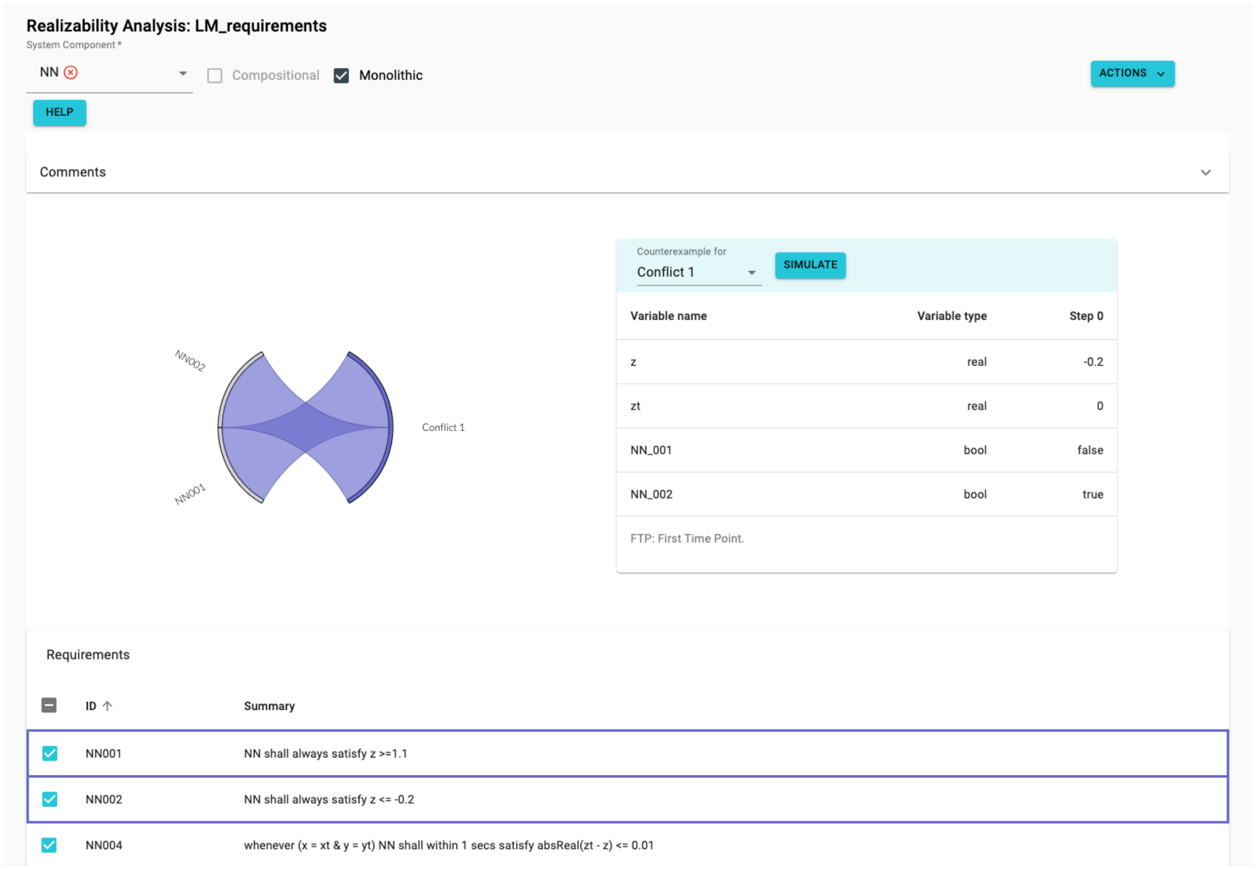
Furthermore, FRET supports interactive simulation of a FRETISH requirement to ensure that it captures the user's intention. Given a FRETISH requirement, the simulator shows temporal traces of each of the signals (variables) involved as well as the valuation of the requirement for each point in time. The user can modify the input signals; the valuation of the requirement is updated automatically and thus makes it possible for the user to visually inspect and validate the temporal behavior of the requirement. FRET allows the simultaneous validation of multiple related requirements through its simulator component.

Automatic Generation of Logic Formalizations FRET formalizes FRETISH requirements in metric temporal logics and in probabilistic temporal logics. Such formalizations are difficult to correctly specify manually, even for formal methods experts. See the examples in Table 1.

Realizability Analysis of Requirements FRET allows a user to check that a set of requirements is *realizable*, i.e., it remains satisfiable given any expected input from the environment [28]. If the set is not realizable, FRET supports *explainability* features: it performs a diagnosis to find minimal sets of conflicting requirements, and shows counter-example traces. Figure 2 shows a screenshot of the FRET analysis portal, giving an example of unrealizable requirements of a neural network based component from [48–50]. This diagnosis process provides critical information on how to repair subsets of conflicting requirements. On the other hand, if the set is realizable then the realizability checking mechanism *guarantees* that there are no conflicts between the requirements for any environment input. Being able to check realizability of a set of requirements can enhance the quality of the requirements set.

Realizability checking is a hard problem especially within the context of ML components since these usually include non-linear behavior. To enhance the scalability, in [29], an approach has been proposed that decomposes realizability checking into smaller, more tractable problems (partitions). There, it is proved that checking whether or not a specification is realizable reduces to checking that each partition is realizable. Currently, realizability can only be checked when the requirements are not probabilistic but in the future extending realizability checking to account for probabilistic requirements has been planned.

Connections to Analysis Tools FRET generates specifications that can be automatically consumed by tools that perform runtime monitor generation, such as Copilot (Section A.3) and R2U2 (Section A.4),



Realizability Analysis: LM_requirements

System Component *: NN

☐ Compositional ☑ Monolithic

Comments

Counterexample for: Conflict 1

Variable name	Variable type	Step 0
z	real	-0.2
zt	real	0
NN_001	bool	false
NN_002	bool	true

FTP: First Time Point.

Requirements

	ID	Summary
✓	NN001	NN shall always satisfy z >=1.1
✓	NN002	NN shall always satisfy z <= -0.2
✓	NN004	whenever (x = xt & y = yt) NN shall within 1 secs satisfy absReal(zt - z) <= 0.01

Figure 2. FRET Analysis Portal

conventional model checking tools (e.g., NuSMV [51], Kind2 [52], Simulink Design Verifier [53]), as well as probabilistic model checking tool, such as PRISM [37].

Automated Requirement-based Test Case Generation Automated test case generation is possible from requirements stated in FRETISH. The automatically generated tests for a requirement provide requirements-based test coverage according to the FLIP test coverage metric [34], which identifies the extent to which each atomic proposition in the requirement uniquely affects its satisfaction. Furthermore, we are currently studying the application of FLIP (1) as a metric on the quality of a dataset, and (2) as a data augmentation approach for a data set.

The FRET test case generation mechanism is shown with an example requirement from [48]. The natural language requirement and the FRETISH version are shown in Table 1. For the sake of simplicity of explanations, we simplify the requirement to use the immediately timing instead of within 1 second. Thus the requirement becomes:

$$\text{Whenever } (x = xt \;\&\; y = yt) \text{ NN shall immediately satisfy } \text{absReal}(zt - z) \leq 0.01$$

In the above formalization, three Boolean constraints exist: $x = xt$, $y = yt$ and $\text{absReal}(zt - z) \leq 0.01$. As such, three obligations need to be generated that show the effect that each of the three constraints may have in the satisfaction of the requirement. FRET translates the FRETISH requirement into the following Linear Temporal Logic (LTL) formula:

$$G((x = xt \;\wedge y = yt) \Rightarrow \text{absReal}(zt - z) \leq 0.01)$$

which formally encapsulates the tolerance constraint on the absolute error between zt and z, specifying that it needs to always ((G)lobally) be met during the NN's operation, if the precondition $(x = xt \wedge y = yt)$ holds. If the precondition is false, the requirement is trivially satisfied, and the threshold constraint does not need to hold. Assume that we want to cover $x = xt$ while satisfying the requirement. FRET generates the following LTL obligation towards this objective:

$$\begin{gathered} G((x = xt\ \wedge y = yt) \Rightarrow \text{absReal}(zt - z) \leq 0.01) \\ \wedge\, F(\neg(x = xt\ \wedge y = yt) \wedge \text{absReal}(zt - z) > 0.01) \end{gathered}$$

The obligation captures the following:

- The original requirement must still be satisfied.
- Eventually, i.e., in the future (F), $\text{absReal}(zt - z) > 0.01$ is observed while either $x \neq xt$, $y \neq yt$, or both. Note that this case still satisfies the natural language requirement.

More interestingly, the obligation above is the same as the obligation that would be generated for $y = yt$. This relays the fact that $x = xt$ cannot be shown to solely affect the satisfaction of the requirement, but rather that it has to be considered in combination with the truth of $y = yt$.

Considering the above, an adequate test case for this obligation needs to 1) satisfy the requirement and 2) demonstrate how the threshold can be exceeded when the precondition $x = xt\ \wedge y = yt$ does not hold. Such a test covers the case where the original requirement is expected to be trivially true when the input constraints on x, xt, y and yt do not hold. Requiring the test to show that the threshold can be exceeded is important in this case, as the resulting test is more indicative of the requirement semantics, when compared to a test where the requirement is satisfied because the precondition is FALSE and $\text{absReal}(zt - z) \leq 0.01$.

The semantics of FLIP coverage can also be understood when compared against the Modified Condition and Decision Coverage criterion (MC/DC) [15]. In MCDC, a test suite is evaluated in terms of how well it covers each decision (i.e. Boolean expression) in the code. Towards that end, for every decision, every condition (i.e., subexpression in decision with no logical operators) needs to be shown to uniquely affect the decision's truth. In the case of FLIP, the Boolean constraints expressed in the formal requirement can be mapped to MCDC's conditions, whereas the requirement itself can be mapped to MCDC's concept of decisions.

A.2 System Analysis using Statistical AI (SYSAI)

SYSAI (System Analysis using Statistical AI) [54] is a flexible statistical learning framework for V&V and the analysis of complex and high-dimensional cyber-physical systems with AI components.

A.2.1 SYSAI Tool Architecture

Figure 3 shows the high-level architecture of SYSAI analysis framework. On the left-hand side, we have the *system under test* (SuT), which is executed given a set of parameters provided by the statistical learning model of SYSAI. The result of each test run is then used to incrementally construct the statistical model.

SYSAI can be used to analyze an entire system in a black-box manner as well as the behavior of an ML Component in both a black-box and a white-box manner. The latter can provide insights into the network behavior and performance when, for example, comparing different DNN architectures or learning algorithms.

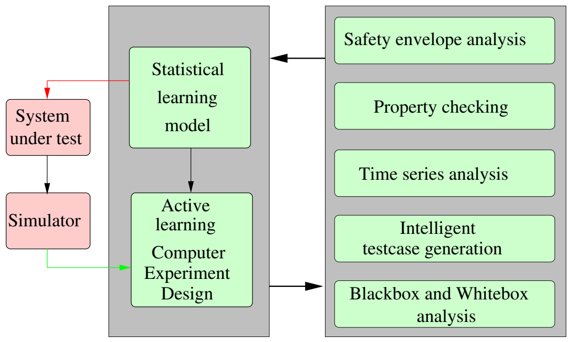

Statistical learning model
System under test
Simulator
Active learning
Computer Experiment Design
Safety envelope analysis
Property checking
Time series analysis
Intelligent testcase generation
Blackbox and Whitebox analysis

Figure 3. SYSAI architecture

In general, the interface between SYSAI and the SuT is designed to be very small and generic, so that systems implemented in various languages such as C/C++, R, Matlab, Java, or Python can be connected easily. For the representation and construction of the statistical model, SYSAI uses *Dynamic Regression Trees* (DynaTrees) [55, 56], a dynamic Gaussian process model based upon Particle Filters. DynaTrees are regression and classification learning models with complicated response surfaces in on-line application settings. DynaTrees create a sequential tree model whose state changes over time with the accumulation of new data, and provide particle learning algorithms that allow for the efficient on-line posterior filtering of tree-states. A major advantage of DynaTrees is that they allow for the use of simple models within each partition. The models also facilitate a natural division in sequential particle-based inference: tree dynamics are defined through a few potential changes that are local to each newly arrived observation, while global uncertainty is captured by the ensemble of particles.

This surrogate model is initialized with available training data and incrementally refined using candidate data points that are produced by an active learning module (see [57] for details). It evaluates the current surrogate model using a customized active-learning heuristics and suggests candidate data points that provide most information for model refinement. For these candidate points, the ground truth is obtained by executing the SuT.

A.2.2 SYSAI Capabilities

SYSAI provides support for V&V around the tasks of finding safety regions, characterizing system performance for ConOps, analyzing system behavior in failure modes, and performing statistical analyses on the training and testing data sets. A detailed description of the SYSAI framework and its capabilities are described in [58], and summarized in brief next:

Safety Envelope Analysis SYSAI can perform an automated analysis of safety envelopes to discriminate between safe and unsafe operational conditions of system behavior. Safety envelopes are not usually rectangular; SYSAI can model geometric shapes to identify and characterize regions of similar behavior, describing those regions in simple geometric terms using typical shapes (e.g., spheres, ellipsoids, cylinders, cubes, etc.)

ConOps and Failure Mode Analysis Safety-critical systems must continue to operate safely under numerous failure conditions. SYSAI can be used to identify regions in the state space where a failure has substantial impact on system performance. Similarly, SYSAI can explore environmental conditions to evaluate the system's performance under those conditions. Examples include location, time-of-the-day, visibility, weather, or dirt spots on the camera in a vision-based system.

Statistical Analysis of Training and Test Data All guidelines for V&V and certification of AI/ML systems put a strong focus on the tasks around collection and acquisition of data, as well as their quality analysis and data management. SYSAI can perform detailed statistical analysis on training and testing data, both for high-dimensional visual data (e.g., camera images) as well as time-series data.

Property Checking SYSAI supports the automatic checking and analysis of safety and performance requirements. SYSAI can effectively find regions and their boundaries where certain safety properties (e.g., the maximum error is always less than 40ft) are violated.

Time-series analysis SYSAI can perform advanced time-series analysis in a high-dimensional parameter and state space. This analysis provides a deeper understanding of the system behavior and its dynamics. The tool also supports event prediction.

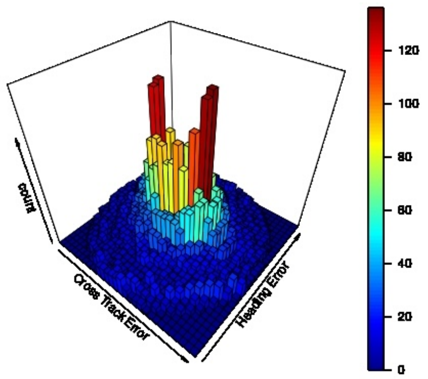
White-box Analysis SYSAI can perform analysis on the AI component in isolation. This allows us to carry out analyses on internal calculations within the AI component, comparison between different AI components, and learning.

A.2.3 Process Integration

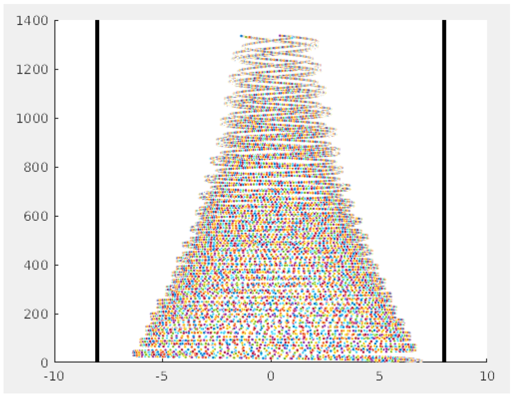
SYSAI can support many elements of an ML Lifecycle (Section 2.2) as shown in Table 5.

Table 5. ML lifecycle elements supported by SYSAI

ML Lifecycle Element	SYSAI Results	SYSAI Support
Requirements development	Feedback to designer	Property checking at an early stage on system and component level; Safety boundary analysis
Data Management	Feedback for data acquisition	Statistical analysis of training and test data providing coverage analysis, time series analysis, and rare event handling
Model Training	Model evaluation	White-box analysis of ML component with capability of model comparison
Model verification	Model evaluation	White-box analysis, Uncertainty quantification, Out-of-distribution analysis



(a) Data distribution



(b) Data coverage

Figure 4. Analysis of training and test data

Table 5. ML lifecycle elements supported by SYSAI (Continued)

ML Lifecycle Element	SYSAI Results	SYSAI Support
System-level validation	Requirements analysis	System-level, high dimensional analysis for verification of safety regions, performance boundaries, and verification of system-level and component-level requirements

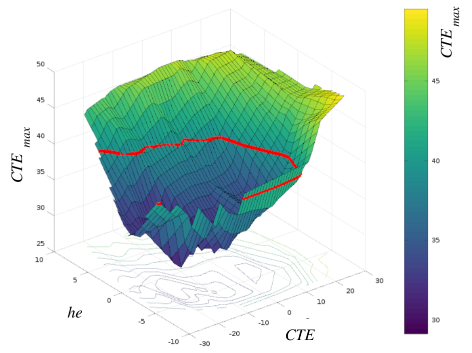
Statistical Analysis of Training and Test Data The probability distributions of training and test data can be obtained with SYSAI, e.g., Figure 4 shows poor coverage of the given training data: in Figure 4a, training data do not exist for areas toward the end of the runway if the aircraft is not close to the runway.

Safety Boundary Analysis for ML requirements SYSAI can perform automatic analysis of safety boundaries for an autonomous centerline tracking (ACT) example with a deep convolutional neural network. Figure 5b shows a projection of the safety-envelope. The surface shows estimated maximal cross track error (CTE) values during a run over initial values of CTE and heading error (HE) values. The safety envelope at a given threshold (here 40 feet) is shown as a red line. Usually, the safety envelope is not rectangular. Therefore, SYSAI can perform geometric shape modeling to not only identify but also characterize regions with similar behavior and describes those regions in easy to understand geometrical terms.

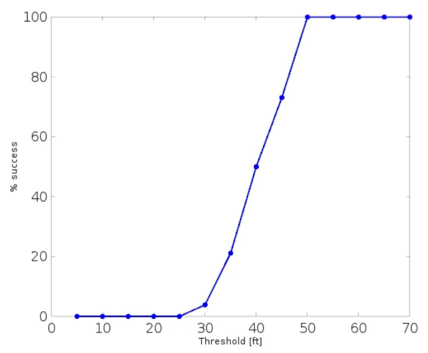
Determining Requirements Thresholds Figure 5b shows the compliance rate for the requirement, *Aircraft shall diverge no more than X feet from the runway centerline* over a range of thresholds. Smaller thresholds (narrow runway) lead to a low compliance rate.

Operational Requirements Analysis SYSAI can explore environmental conditions to evaluate the system's performance. Figure 6 shows ACT performance for different times of the day. The performance is dropping dramatically for times after around 2PM, the reason being different lighting and shadow conditions. In our example, the DNN has only been trained with images taken at 9AM.

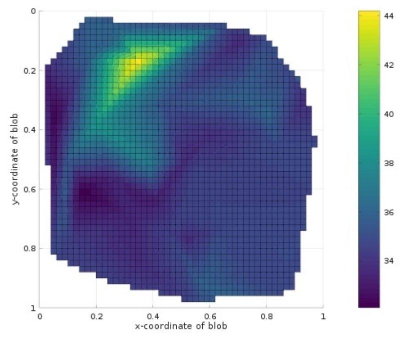
System Validation through Failure Mode Analysis SYSAI can be used to identify regions in the state space where a failure has substantial impact on system performance. For the ACT system, a clear



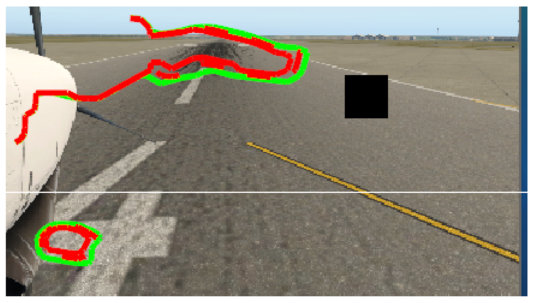
(a) Safety region: Red line corresponds to a threshold of 40 feet



(b) Compliance over threshold



(c) Heat map of sensitive areas of a DNN relative to camera failure (location of a dark spot on an image) Brighter colors correspond to higher sensitivity



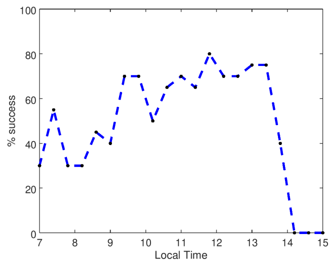
(d) Sensitive areas overlaid on a camera image. The black square represents a typical injected failure

Figure 5. Analysis of requirements thresholds and safety envelope boundaries

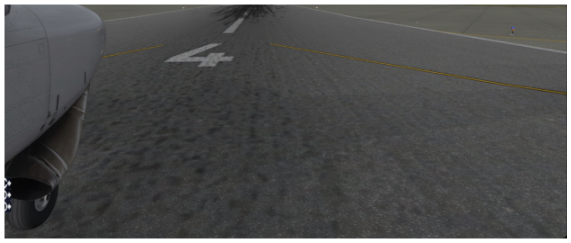
camera image of the runway is important. In practice, however, a partial obstruction of the image, e.g., caused by a piece of dirt on the camera lens, can occur. Figure 5c shows the results of a SYSAI analysis, which investigated how the location of an obstruction (modeled as a black rectangle) influences the ACT system behavior. Regions highlighted in red are particularly sensitive: as expected areas near the center of the runway toward the horizon, but surprisingly also an area close to the AC front wheel. If the obstruction occurs in these areas, the performance of ACT is very diminished and can cause the AC to leave the runway.

Black Box and White Box Analysis of ML Models SYSAI is capable of analyzing the entire system under test. However, SYSAI can be used to specifically analyze an ML component (e.g., a deep neural network) in isolation. The framework supports black-box and white-box analysis (in the latter case, e.g., weights or internal calculations of the DNN are available to SYSAI), and can perform comparisons.

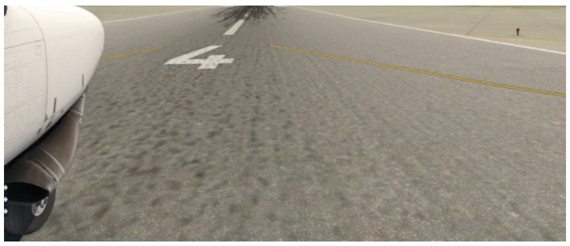
Figure 7 shows the probability distribution of the actual DNN output $\mathtt{cte}$ versus the ground truth $\mathtt{cte}_{\mathtt{gt}}$. Two different deep neural network (DNN) architectures that have been trained for ACT have been considered in this case. The ideal DNN should have sharp peaks along the diagonal (shown in red). Whereas DNN 1 has an overall good behavior but a small bias for negative values of $\mathtt{cte}$, DNN 2 behaves better in that area but has substantial deviations for larger positive values. These results are based upon the DNN only analysis. Incorporated into the ACT system, both DNNs perform satisfactory, due to robustness of the controller. Here, SYSAI caused the execution of an entire run



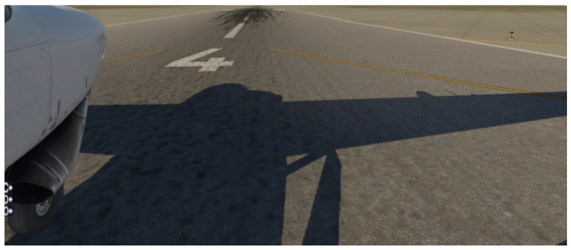
(a) Success rate (in %) for different times of the day. Threshold for CTE is 40 feet



(b) Camera image from a taxiing run (in simulation) at 8AM



(c) Camera image from a taxiing run (in simulation) at 11AM



(d) Camera image from a taxiing run (in simulation) at 3PM

Figure 6. Operational Requirements Analysis with SYSAI

down the runway with a random initial cte.

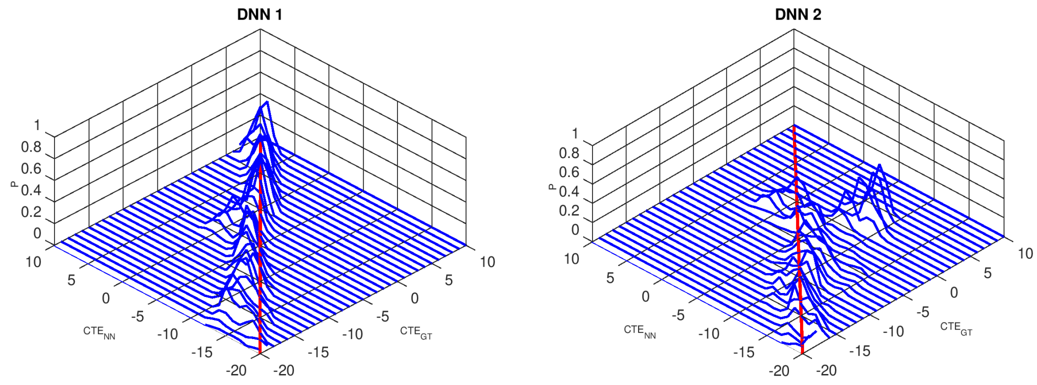


Figure 7. Performance analysis of two different DNN architectures

A.2.4 Support for Assurance Considerations

SYSAI can provide evidence to support and address the considerations elaborated in Section 3.1

Generalizability Analysis SYSAI can efficiently explore high-dimensional parameter spaces enabling generalizability analyses. At an MLC level that includes the characterization of regions with low confidence during generalization, DNN performance analysis, calculation of ROC curves for different ranges of thresholds, and characterizing the behavior of false alarms/false negatives. Impact analysis of generalizability requirements on the system behavior explores, as shown in Figure 6, a characterization of (large) variations in performance of an MLC (here, the ACT example) under different environmental conditions (here different times of the day). This can be directly attributed to poor generalization behavior of the trained deep neural network.

Analysis of Robustness SYSAI can perform system-level robustness analysis by analyzing the system performance in the presence of failures or environmental conditions (e.g., light, fog, wind). Figure 5c indicates sensitive regions, i.e., regions where ACT is not robust with respect to camera failures. SYSAI provides several ways of exploring different failure modes and presentation of results as evidence for robustness or lack of robustness (see Figures 5c and 8b). SYSAI can also perform a robustness analysis of ML components in a white box fashion. Here, robustness of changes in the image (e.g., fog induced diffusion, brightness, or dirt on camera lens) to the output of the DNN is explored using heat maps (Figure 8a).

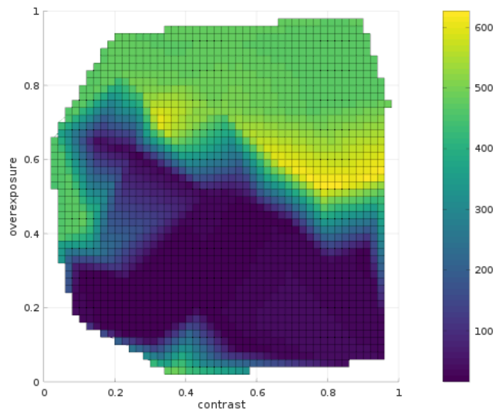
Table 6. Methods for generalization improvement during training and post-training, Analyses for in-distribution versus and out-of-distribution generalization.

Generalizability Techniques	In-distribution	Out of Distribution
Training Augmentations	Data Augmentation, Regularization	Domain adaptation, Distributionally Robust Optimization
Post-training analysis methods	Rademacher Complexity, VC Dimension, Probably approximately correct (PAC) Bayesian	Agreement on the Line, Average Thresholded Confidence, Detection of OOD Inputs

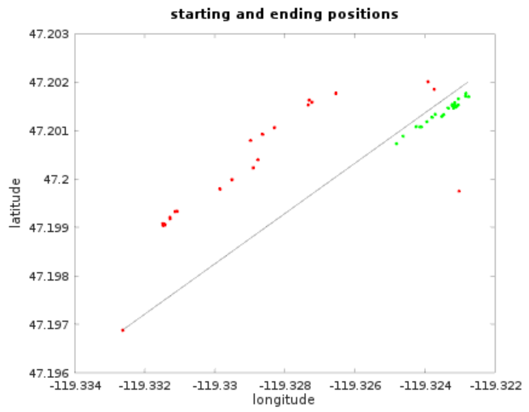
Out of Distribution Performance Analysis There are a number of different techniques for the analysis of out of distribution generalizability as shown Table 6. Data augmentation can be done through the acquisition of additional (real) data or by generating sufficient data according to the requirements of the domain and application. Because acquiring real data is a costly and time-consuming process, the generation of synthetic data is becoming increasingly important. A major tasks, which can be supported by SYSAI is their rigorous statistical analysis and measurements how the distribution of the newly generated data align to the required distribution of the real data. Figure 9 shows a comparison of errors in OOD prediction using different metrics.

Model Development SYSAI can perform analysis over hyper-parameters of learning algorithms and model parameters. Therefore, SYSAI can provide evidence regarding performance and possible flaws of learning algorithms and models.

Explainability During its operation, SYSAI produces a statistical surrogate model of the system or the AI component. These models are usually of a relatively small size and therefore are more amenable to human analysis and explanation than large unstructured DNNs. SYSAI can also create such surrogate models for the entire system (helpful for explaining system behavior) as well as for the AI/ML component.



(a) Heat map for robustness of a DNN for autonomous centerline tracking relative to image contrast (X-axis) and brightness/exposure (Y-axis). Areas of high robustness are in dark blue



(b) System-level robustness: Dots denote the endpoint of a successful (green) or failed (red) taxiing run. The diagonal line denotes the runway centerline

Figure 8. Robustness analysis with SYSAI

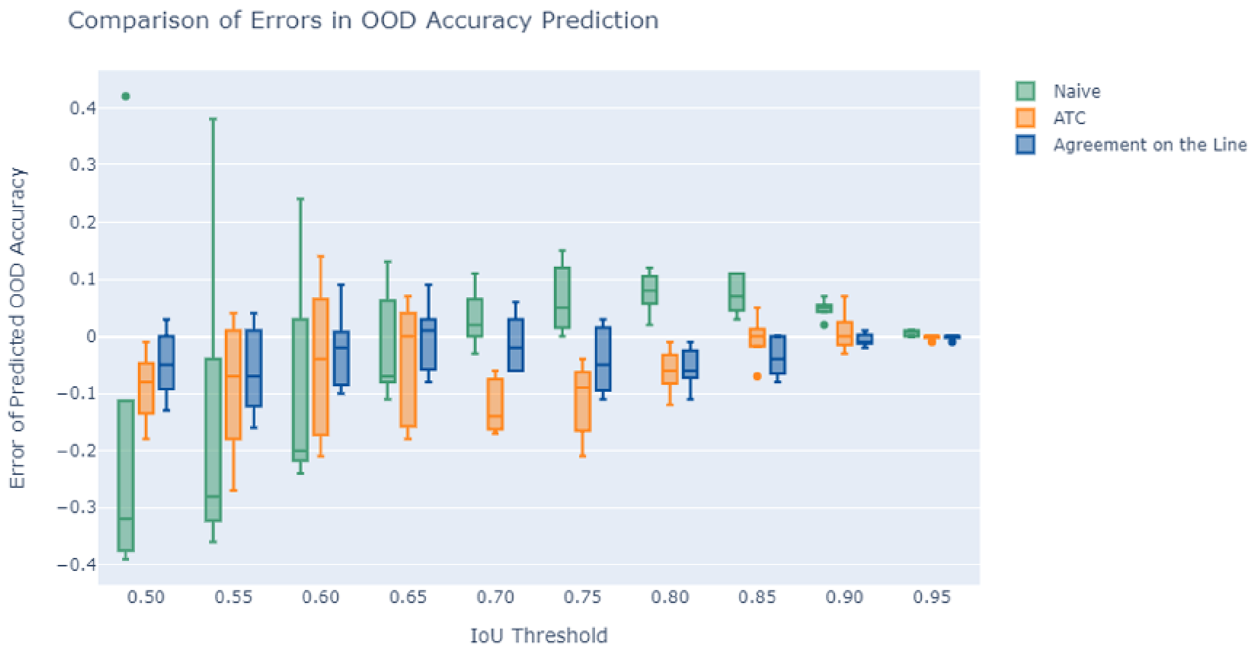


Figure 9. Different metrics for OOD accuracy measurement and prediction

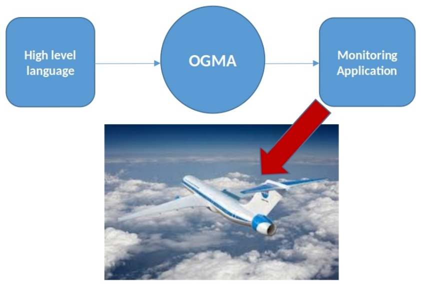


Figure 10. High Level Application of Ogma

A.3 Runtime Monitoring with Ogma and Copilot

Two kinds of tools to introduce runtime monitoring into larger systems are described: those tools that seek full integration of runtime verification (RV) and runtime assurance (RTA) solutions into larger systems, and specific RV languages.

Ogma Ogma is a tool a tool to generate monitoring applications from high-level languages. The goal of Ogma is to simplify the process of generating runtime monitors for safety-critical applications (Figure 10).

Ogma is able to produce not just the implementation of the code that monitors a given set of properties, but also additional code that facilitates using the monitors in a target platform. By doing so, Ogma makes it possible to generate monitoring applications that are ready to run, with no changes needed after the generation, or as properties are updated or new properties are added to the monitoring system.

In the context of assurance of ML for flight software, Ogma could simplify the process of producing assured flight software, by automating the process of going from high-level property or requirement descriptions into runtime monitors that are ready to fly. More specifically, Ogma can convert ML system properties or requirements written in several languages, such as Copilot, SMV, different forms of temporal logic, and the output formulas produced by FRET, and generate runtime monitoring applications that can be incorporated in some of the most frequent flight software stacks such as NASA's Core Flight System, F', and the robotics framework ROS. Under the hood, Ogma leverages existing runtime monitoring languages to implement the core code of monitors such as Copilot (described below) and R2U2 (see Appendix A.4).

Ogma eliminates a large source of potential errors in the application generation process, as both writing the property in a monitoring language and connecting the monitors to the rest of the system happen automatically.

Copilot Copilot is an open-source, real-time programming language that has been historically used for runtime monitoring of aircraft and robots. Copilot's designed is similar to the languages used in digital signal processing that have been historically used in aerospace. Monitors are translated into MISRA-compliant C99 code with predictable memory requirements and real-time guarantees. Additional Copilot libraries extend the core language with higher-level constructs, Boyer-Moore majority voting, and temporal logic and statistics. Users can write their own Copilot libraries, so the language is fully extensible to accommodate for other kinds of properties. Specifically for the purposes of monitoring runtime properties of ML systems, Copilot can be used to write both discrete properties and probabilistic properties. The handlers of potential violations can access data synthesized by such properties. For example, they may execute some correction mechanism, but also know the degree of deviation from the norm based on a system property that has been specified in Copilot or in any of the high level languages supported by Ogma. Apart from generating hard-realtime code, Copilot also generates a formal proof of correctness of such code that establishes that the C code that implements the actual monitors follows the denotational semantics of the language. In other words: it is a formal proof that the C code behaves as it should.

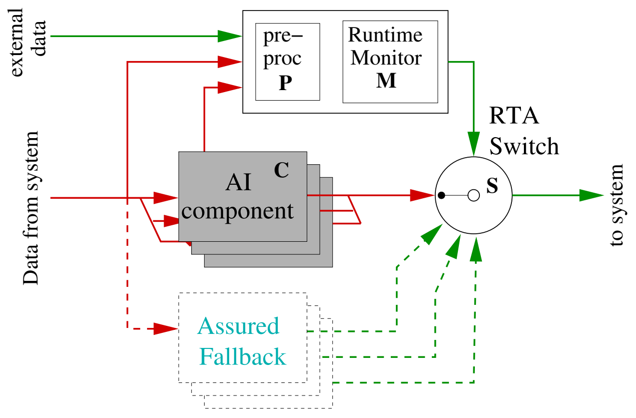


Figure 11. Typical RTA Architecture, inspired by [59], with an AI component or an MLC (C), signal processing (P), runtime monitor (M), and RTA switch (S) to enable fallback components. Verified signals are in green, others in red. Compared to the architecture in [59], this configuration permits the monitor to use unverified signals (in red), e.g., signals from C.

A.4 Realizable Responsive Unobtrusive Unit (R2U2)

R2U2 (Realizable Responsive Unobtrusive Unit) is a runtime monitor that combines observers for past and future time metric logic, an efficient prognostics engine, and Bayesian reasoners. Versions of R2U2 have been developed to run as FPGA configurations, on parallel co-processors, and as a small, memory-efficient software component. R2U2 is typically used for system and software health management, detection of cybersecurity attacks, and to monitor autonomous operations on uncrewed aircraft. With respect to monitoring of ML components, the following capabilities of R2U2 are of particular interest:

Past-time and Future-time Temporal Observers Many properties that need to be checked for an ML/AI component or a system with such component have temporal aspects. R2U2 is using efficient ring-buffers to keep memory and computational overhead low. Besides past-time monitors, R2U2 also provides synchronous future-time observers that, at each point in time, can provide a verdict of true, false, or maybe.

R2U2 within a Runtime Assurance Architecture R2U2 can be used as the runtime monitor within an RTA architecture as discussed above. Using temporal logic and properties, R2U2 can monitor a multitude of signals that are processed and discretized using R2U2's signal processing unit. Probabilistic properties and properties about (failure) rates can easily be expressed. Thus R2U2 can define a maximum rate of failure occurrences or a maximum interval for a persistent failure, before switching will occur, to reduce the number of false alarms.

Monitoring of Formalized Requirements stated in FRET R2U2 can work on requirements as captured by FRET (Section A.1). This enables a smooth integration of R2U2 monitors in processes where FRET is used.

R2U2 Property Groups For monitoring systems with MLCs, R2U2 use different classes of (temporal) properties. Table 7 shows the major classes, and an example for each class, taken from the ACT case study. Note that in this architecture, both verified and unverified signals are monitored. This allows R2U2 to analyze important "internal" states and calculations in a white-box fashion.

The R2U2 engine produces a verdict for each property at each discrete point in time. Usual update rates for R2U2 are 0.1s or 1s. Verdicts can either be Booleans or probabilistic values. With that information, an RTA switch can be triggered (see Figure 11) or can be stored as a time series for later analysis. Typically, these data are only available after system deployment.

Table 7. Types of R2U2 properties to be checked. Examples are shown in informal natural language.

Property Class	Description and Example
SS	System Safety: Properties to ensure safe operation of the system
	Example: *When the aircraft is near the right border of the runway, the AC shall not move further to the right ($he > 0$) for an extended time*
SC	Component Safety: Safety properties concerning the component
	Example: *The DNN output values shall be reasonably limited*
PS	System Performance: Properties concerning performance (e.g., progress of mission)
	Example: *The aircraft shall reach the end of the runway within T seconds*
PC	Component Performance: Properties concerning performance of the ML component
	Example: *The DNN outputs should not fluctuate*
ES	Properties concerning environmental (e.g., weather, visibility) or operational (e.g., operational hours) conditions
	Example: *ACT shall only operate between 9AM and 2:30PM local*
FS	Properties for system regarding failure modes
	Example: *A camera obstruction (no signal in certain area of image sensor) shall not show up for an extended time*
FC	Properties on failures of ML components
	Example: *If DNN1 is active and a camera obstruction occurs in a sensitive area, DNN2 shall be activated*

Contribution to Verification and Validation

Table 8 gives an overview of the elements of an ML lifecycle where R2U2 and an R2U2-based runtime assurance architecture (see Figure 11) can be employed. Although the typical use of R2U2 focuses on post-deployment monitoring and assurance, R2U2 can also be used during traditional model development and V&V. During model development, R2U2 can be used to monitor the behavior of an MLC during validation and testing. The capability to access internal component data makes R2U2 a powerful and convenient tool to quickly and reliably check numerous properties during one run. Its use avoids coding of test conditions and allows a quick check of requirements formalized using FRET. During traditional V&V, R2U2 can be used to automatically check numerous properties during a single test run. This can help reduce the number of test runs and minimize the effort to set up test filters and evaluators.

Table 8. Applicability of R2U2 and R2U2-based RTA during ML development and certification process

ML Lifecycle Element	R2U2	R2U2-RTA	Description
Operating Context Definition	-	-	R2U2 is not directly applicable; FRET requirements can be translated into R2U2 properties for later checking
Data Management	-	-	Not applicable
Model Development	✓	-	R2U2 can support by monitoring model behavior during model development and validation
Pre-deployment V&V	✓	-	R2U2 supports efficient component and system testing with complicated test conditions
Post-deployment V&V	✓	✓	Monitoring of system and/or AI/ML components during runtime

Table 9. Suitability of R2U2 and R2U2-based RTA for Model Properties

Model Property	R2U2	R2U2-RTA	Description
Generalizability	✓	✓	R2U2 can monitor situations with an excessive generalization gap, based upon external signals and (unverified) confidence signals produced by an MLC component (if available). Properties and parameters are to be defined during static V&V.
Robustness	✓	✓	R2U2 can perform fault detection and diagnosis for the an MLC and other system components. Model-based reasoning enables R2U2 to determine if a given situation affects robustness properties of the MLC and then R2U2-RTA can switch over to more robust components when appropriate.
OOD Performance	✓	✓	R2U2's capability to process signals from different system components. Combined with Bayesian reasoning R2U2 can help to detect and protect against OOD inputs via R2U2-RTA

A.5 SafeDNN

The SafeDNN (Safety of Deep Neural Networks) effort explores techniques and tools for design-time analysis of trained Deep Neural Network (DNN) models and systems that use these models. Research directions being pursuing in this project include: symbolic execution for DNN analysis, parallel and compositional approaches to improve formal verification of DNNs, property inference and automated program repair for DNNs, probabilistic reasoning and compositional verification for DNNs and systems that use DNNs (particularly for vision tasks). In the following we highlight some of the research advances from SafeDNN and how they address the three questions above.

A.5.1 Verification and Validation of the Learnt Model

Checking that a set of inputs cannot produce an erroneous output is paramount to using the learnt model in safety-critical settings. Techniques addressing this include testing and formal verification. Testing may be insufficient for proving that faulty behaviors do not exist within the continuous DNNs while formal verification can provide assurance gurantees. Properties to consider include: robustness (to adversarial or natural perturbations) and safety properties (as in the case of ACAS-Xu).

Robustness It has been observed that state-of-the-art networks used in image classification are vulnerable to *adversarial perturbations*: given a correctly-classified input x, it is possible to find a new input x' that is very similar to x but is assigned a different label [60]. Typically these perturbations correspond to small changes imperceptible to the human eye. More recent works have shown that such perturbations can severly impact the accuracy of not just image classifiers but also regression models as well as those processing other types of input such as text data. The vulnerability of neural networks to adversarial perturbations is thus a major safety and security concern, and it is essential to explore systematic methods for evaluating and improving the robustness of neural networks against such perturbations.

Local Robustness with respect to L^p norms Local robustness of a model indicates consistent behavior (absence of adversarial perturbations) within local regions surrounding validly classified inputs. SMT solvers such as Reluplex [61] have been used to formally verify *local robustness* of a model. Given an input x, the solver is used to check if there is another point x' within a close distance δ to x ($\|x - x'\| < \delta$) for which the network assigns a different label.The distance between the points in the input space is typically measured wrt L^p norms, such as Euclidean distance (the L^2 norm) or the Manhattan distance (L^1 norm). For points $x_1 = \langle x_1^1, \ldots, x_n^1\rangle$ and $x_2 = \langle x_1^2, \ldots, x_n^2\rangle$ these are defined as:

$$\|x_1 - x_2\|_{L^1} = \sum_{i=1}^{n} |x_i^1 - x_i^2| \quad ; \quad \|x_1 - x_2\|_{L^2} = \sqrt{\sum_{i=1}^{n} (x_i^1 - x_i^2)^2}$$

The absence of a solution is a proof for local robustness of the model at the given input. Our tools such as Prophecy [30], and DeepCheck [31] employ SMT-based solvers such as Reluplex and symbolic execution to provide local robustness guarantees for neural networks. Local robustness can be viewed as local properties of a model which can act as building blocks to reason about its robustness towards more complex domain-specific perturbations.

Local Robustness to Natural Perturbations In [32] we introduce DeepCert, a tool-supported method for verifying the robustness of deep neural network (DNN) image classifiers to contextually relevant perturbations such as blur, haze, and changes in image contrast. While the robustness of DNN classifiers has been the subject of intense research in recent years, the solutions delivered by this research focus on verifying DNN robustness to small perturbations in the images being classified, with perturbation magnitude measured using established L^p norms. This is useful for identifying potential adversarial attacks on DNN image classifiers, but cannot verify DNN robustness to contextually relevant image perturbations, which are typically not small when expressed with L^p norms. DeepCert addresses this under-explored verification problem by supporting:

- the encoding of real-world image perturbations;
- the systematic evaluation of contextually relevant DNN robustness, using both testing and formal verification;
- the generation of contextually relevant counterexamples; and, through these,
- the selection of DNN image classifiers suitable for the operational context:
  - envisaged when a potentially safety-critical system is designed, or
  - observed by a deployed system.

The effectiveness of DeepCert is demonstrated on checking the robustness of DNN image classifiers built for two benchmark datasets ('German Traffic Sign' and 'CIFAR-10') to multiple contextually relevant perturbations.

In [62] we consider the related problem of assuring the robustness of deep neural networks against real-world distribution shifts, such as change in weather conditions in perception tasks. We have studied real-world perturbations such as variation in snow; also brightness, fog, scale, contrast, and Gaussian blur.

To do so, we bridge the gap between hand-crafted specifications and realistic deployment settings by considering a neural-symbolic verification framework in which generative models are trained to learn perturbations from data and specifications are defined with respect to the output of these learned models.

Global Robustness Local robustness provides limited guarantees since it is limited to checking robustness around a few individual points, giving no indication about the overall robustness of the network for all inputs (termed as *global robustness*). In principle, one can apply the local check to a set of inputs that are drawn from some random distribution thought to represent the input space. However, this would require coming up with minimally acceptable distance δ values for all these checks, which can vary greatly across different input points. Furthermore, the check will likely fail (and produce invalid adversarial examples) for the input points that are close to the legitimate boundaries between different labels.

In DeepSafe [33], we decompose global robustness requirements into a set of local robustness checks for regions in the input space. Each region encompassing groups of valid inputs that are similar to each other and share the same label. We then utilize verification techniques (such as Marabou) to confirm that these regions are safe or to provide counter-examples showing that they are not safe.

Safety Properties Safety properties refer to conditions for the safe operation of the system, typically provided by domain experts. Effectively these can be encoded in requirements allocated to an MLC against which implementation correctness needs to be shown.

Consider the ACAS-Xu (Airborne Collision Avoidance System for Unmanned Aircraft) system, which consists of a set of 45 DNNs, taking in sensory inputs and predicting horizontal advisories. A set of ten input-output properties of the networks have been specified by domain experts [63], catering to safety considerations such as avoiding unnecessary advisories, uniformity and consistency of alerts, and so on. An example is shown,

$$(\text{range} > 55947.691) \wedge (-3.14 \leq \theta \leq 3.14) \wedge (-3.14 \leq \psi \leq 3.14) \wedge \\ (1145 \leq v_{\text{own}} \leq 12000) \wedge (0 \leq v_{\text{int}} \leq 60) \Rightarrow \text{Clear of Conflict (COC) Advisory}$$

In neural networks processing raw data such as images or sound or text, it is a challenge to specify such *input-output* properties of the model, since it is not possible to express the pre-condition on the inputs in terms of raw data such as pixels. In such cases, the property is typically expressed only in terms of the model outputs such as in the case of TaxiNet, a system for autonomous center line tracking using neural networks. The regression model takes in images of the runway as inputs and generates two outputs; cross-track error (y_0), and heading error (y_1). The expert-provided safety properties specify conditions for the safe operation of the plane in terms of runway dimensions: $|y_0| \leq 10.0m, |y_1| \leq 90$ *degrees*.

An image classification model on the MNIST dataset could have a requirement that the classification label 9 (the functionality of identifying an image of digit 9) is accurate and is generalizable (applicable to all images of digit 9). In order to verify this requirement, the post-condition or output property to check is that the classifier model assigns the highest score to class 9 versus the remaining classes. However, the pre-condition cannot be expressed in terms of input pixels. It is not possible to define mathematically over pixel values what "all images of digit 9" means. It is therefore a challenge to corroborate the *generalizability* of the functionality of a model beyond the inputs in the training and test data.

Considering this example, Prophecy [30] looks into the inner layers of the model to capture neuron-patterns that are commonly satisfied by many images of digit 9. Each neuron-pattern potentially captures high-level features (such as "curve", "loop" so on) which should be present in the digit in the image to classify it as a 9. The rules have a simple mathematical form and can be checked using existing solvers, if proved they become properties of the network.

In the ACAS-Xu system, Prophecy was also used to simplify proofs [30] for the expert provided properties which are very expensive when checked on the full network. Our work used the mined neuron-pattern based rules to decompose the proof into parts that could be solved separately and in parallel. Prophecy was also used to aid in the verification of th safety properties on the regression model of the TaxiNet system [64]. The Marabou solver was employed to check constraints of the form `pre` $\Rightarrow$ `post`, where the `post` was the safety property, and `pre` was neuron-pattern (conditions on the neurons of the network).

These neuron patterns were mined by Prophecy and characterized sequences of valid images where the model outputs satisfied the safety property. The proofs provide guarantees of safe behavior and also generate counter-examples that represent cases where the aircraft could potentially steer off the runway.

Probabilistic Analysis Formal verification of neural networks using tools like Reluplex [63] or Marabou is challenging due to network sizes that are gigantic.

Further, properties often do not hold due to the approximate nature of neural networks (with counterexamples showing noise). Statistical techniques are often used to check properties of neural networks, such as robustness to noise and inaccurate inputs or the fairness of their decisions. While scalable, statistical methods may underestimate the impact of low probability inputs that lead to undesired behavior of the network.

In [65], we investigate the use of symbolic execution and constraint solution space quantification to precisely quantify probabilistic properties in neural networks. We collect symbolic constraints corresponding to the network's response to concrete inputs, while efficiently rejecting inputs whose responses have been seen before. We further propose a quantification procedure for the collected constraints, producing tight, sound interval bounds on the estimated probabilities. The proposed approach is an anytime algorithm, increasing in precision with more paths explored. We implemented our approach in the SpaceScanner tool, and demonstrate its potential in analyzing fairness, robustness, and sensitivity properties of neural networks. This tool has also been employed to provide quantified estimates of the local robustness of the ACAS-Xu model. This highlighted exactly input points where the model was particularly vulnerable to adversaries and specifically to which target labels. A related tool, QuantifyML [40], applies model counting to assess the learnability, safety, and robustness of machine learning models.

Testing Formal verification is often infeasible in real-sized DNN models, therefore a statistical validation of models via testing is often the most practical method for validation. However, it is essential to ensure that the tests cover all the possible behaviors of the model, including corner cases. Coverage metrics can be used to measure the adequacy of testing. Code coverage metrics have been typically used to measure the adequacy of testing of traditional software programs. A number of coverage metrics have been proposed for DNN testing as well.

In [66], we focussed on structural coverage criteria for DNNs, due to their popularity, wide use, and similarity to established metrics for general-purpose software. There are also other, non-structural proposals for measuring the adequacy of testing DNNs, e.g., safety coverage [67], and surprise adequacy [68]. We proposed a tool which can measure the coverage of structural entities of a neural network by the tests in a give test-suite.

Neuron Coverage (NC) NC can be seen as a statement coverage variant for DNNs. A neuron $n_{l,i}$ is said to be covered, if its neuron activation value ($a_{l,i}$) is larger than 0 (or some specified threshold) for at least one test in the suite. Thus, the set of test conditions to be met for NC can be formulated as follows, where L is the number of layers.

$$\{a_{l,i} > 0 | 1 < l < L\}$$

Neuron Boundary Coverage (NBC) NBC extends NC by considering the neuron activations at the maximum and minimum boundary cases. Assuming $high_{l,i}$ and $low_{l,i}$ are respectively the estimated upper and lower bounds on the neuron activation value $a_{l,i}$, we can formulate the set of test conditions for NBC as follows.

$$\{a_{l,i} > high_{l,i}, a_{l,i} < low_{l,i} | 1 < l < L\}$$

The estimation of the bounds is typically done via profiling with the training dataset. Intuitively, for new test inputs, the output of the neurons may fall outside the interval $[low_{l,i}, high_{l,i}]$ prescribed by the training set, indicating testing of new network behaviour.

Strong Neuron Activation Coverage (SNAC) SNAC focuses on test conditions on corner cases with respect to the upper boundary value.

$$\{a_{l,i} > high_{l,i} | 1 < l < L\}$$

K-Multisection Neuron Coverage (KMNC) KMNC divides a neuron's activation range between $high_{l,i}$ and $low_{l,i}$ into K equivalent sections, each denoted by $range_{l,i,k}$, and test conditions in KMNC are defined as the coverage of these activation sections.

$$\{a_{l,i} \in range_{l,i,k} | 1 < l < L, 1 \leq k \leq K\}$$

Top-K Neuron Coverage (TKNC) Given a test input x, a neuron is TKNC covered if its neuron activation value is one of the most active K neurons at its layer, denoted by $a_{l,i} \in top_K(l,x)$. Here $top_K(l,x)$ denotes the neurons that have the largest K activation values at layer l on input x. The rationale for this criterion is that top active neurons (at different layers) may be good indicators for major functionality in the network. The test conditions are as follows.

$$\{a_{l,i} \in top_K(l,x) | 1 < l < L\}$$

Modified Condition/Decision Coverage (MC/DC) Different from the coverage criteria above, MC/DC takes into account the relation between neuron activations at two adjacent layers, such that its test conditions require that any neuron activation at layer $l+1$ (decision) must be independently impacted by each neuron at layer l (condition).

$$\{\forall i, j, h, change(a_{l,i}) \wedge change(a_{l+1,j}) \wedge \neg change(a_{l,h}) | 1 < l < L-1\}$$

A Sign (S) change function and a Value (V) change function are defined in [69] for depicting how a neuron activation changes when the input changes from a test to another. As a result, there is a family of four variants of MC/DC for DNNs, including SS coverage (SS), SV coverage (SV), VS coverage (VS) and VV coverage (VV).

The structure and implementation of DNN models is very different from traditional software, therefore it is unclear if covering different structural entities of a DNN, translates to adequate testing of different functionalities, coverage of important input features, corner cases, and also misbehavior.

Furthermore, the automated generation of the ground truth for test inputs is an open, less studied problem, as current techniques use manual labelling, which is very costly, or metamorphic testing, which limits the functionality that can be effectively tested.

A.5.2 Verification in Deployment and Integration

System-level, Closed-loop Analysis We have also developed techniques for requirements-based verification of the learnt model when integrated into the containing system. Simulations and in-field testing are typically used but they are very expensive and provide no formal guarantees. Formal verification is desirable but very challenging due to the complexity of the learnt models (DNNs with millions of neurons), the sensors (e.g., cameras capturing images), and the environment conditions.

In [36] we present a case study applying formal probabilistic analysis techniques to an experimental autonomous system that guides airplanes on taxiways using a perception DNN. We replaced the camera and the network with a compact probabilistic abstraction built from the confusion matrices computed for the DNN on a representative image data set. The resulting system can then be modeled and analyzed using standard verification techniques (we used PRISM [37] in our case study). The probabilities in the abstraction are estimated based on empirical data, so they are subject to error. We explore the use of confidence intervals in addition to point estimates for these probabilities and thereby strengthen the soundness of the analysis.

While the above technique produces probabilistic guarantees for system-level properties expressed in pCTL, in [70] we present an assume-guarantee style compositional approach for the formal verification of system-level properties that provides strong, i.e., non-probabilistic, guarantees at the system level. The analysis employs a form of abductive reasoning and is performed in the absence of the DNN components; it automatically synthesizes assumptions on the DNN behavior that guarantee the satisfaction of the required safety properties.

The synthesized assumptions are the weakest in the sense that they characterize the output sequences of all the possible DNNs that, plugged into the autonomous system, guarantee the required properties. The assumptions can be leveraged as run-time monitors over a deployed DNN to guarantee the safety of the overall system; they can also be mined to extract local specifications for use during training and testing of DNNs. The approach is illustrated on a case study taken from the autonomous airplanes domain that uses a complex DNN for perception.

Runtime Monitoring In [71], we proposed a set of runtime mitigation techniques, embodied by the tool AntidoteRT, which employs rules in terms of neuron patterns to detect and correct network behavior on poisoned inputs. We demonstrate that our techniques outperform existing defenses such as NeuralCleanse and STRIP on popular benchmarks such as MNIST, CIFAR-10, and GTSRB against the popular BadNets attack and the more complex DFST attack. The techniques presented in [36] and [70] analyze the use of runtime guards to ensure safe operation for a closed-loop AI system with DNN perception component.

In [42], we deployed feature-rules as runtime guards to filter inputs and showed that the model accuracy computed for inputs with "shadow present" and "dark skid", respectively, is poor, whereas it is high for inputs with "centerline". The rules for the centerline feature when deployed as a run-time monitor to either pass inputs satisfying the rules for "present" or reject those that satisfy the rules for "absent" ensures that the model operates in the safe zone as defined by the ODD.

A.5.3 Explainability

In [30] and [64], we used techniques including gradient-based attribution to map rules (for correct/incorrect behavior) in terms of neuron-patterns at internal layers to input images. Example: we showed that when the TaxiNet model is able to correctly identify the nose of the plane and the center-line of the runway, it estimates the distance between them correctly.

In [42], we performed a "feature-guided" analysis of the TaxiNet model for autonomous center-line tracking in airplane runways, to help users understand and debug model behavior. We explain correct and incorrect behavior in terms of combinations of features such as:

$$(\text{centerline present}) \wedge (\text{shadow absent}) \wedge (\text{on position}) \Rightarrow \text{correct}$$

$$\neg(\text{centerline present}) \wedge (\text{heading away}) \wedge (\text{position right}) \Rightarrow \neg\text{correct}$$

B Acronyms

ACT	Autonomous Centerline Tracking
AST	Adaptive Stress Testing
ANN	Artificial Neural Network
CBMC	C Bounded Model Checker
CM	Configuration Management
CNF	Conjunctive Normal Form
CTE	Cross Track Error
DNN	Deep Neural Network
FAA	Federal Aviation Administration
FHA	Functional Hazard Assessment
HE	Heading Error
IID	Independent and Identically Distributed
MC/DC	Modified Condition/Decision Coverage
ML	Machine Learning
MLC	Machine Learnt Component
MLM	Machine Learnt Model
MSE	Mean Squared Error
NN	Neural Network
ODD	Operational Design Domain
OOD	Out of Distribution
PSSA	Preliminary System Safety Assessment
QA	Quality Assurance
RE	Requirements Engineering
RTA	Runtime Assurance
RV	Runtime Verification
SMS	Safety Management System
SMT	Satisfiability Modulo Theory
SuT	System Under Test
V&V	Verification and Validation