Unintended Behavior in Learning-Enabled Systems: Detecting the Unknown Unknowns

Darren Cofer
Collins Aerospace
darren.cofer@collins.com

***Abstract*—One of the important certification objectives for airborne software is demonstrating the absence of unintended behavior. In current software development processes, unintended behavior is associated with some identifiable structural feature, such as specific lines of code or a model element. However, in learning-enabled systems (like neural networks or other machine learning approaches), unintended behavior emerges from the data used to train the system. New inputs not encountered during training may result in novel activations in a neural network, leading to unexpected (and potentially dangerous) outputs. In this paper we will first review the rationale and methods for detecting unintended behavior in current airborne software systems, including the use of model based development techniques and formal methods for software verification. Then we will consider the challenges posed by learning-enabled components (LECs) and examine new techniques that are being developed to address these challenges, as well as how these techniques may shape new certification guidance.**

***Index Terms*—machine learning, certification, assurance**

I. Introduction

> Reports that say that something hasn't happened are always interesting to me, because as we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also *unknown unknowns*–the ones we don't know we don't know. And if one looks throughout the history of our country and other free countries, it is the latter category that tends to be the difficult ones.
>
> —Donald Rumsfeld [5]

For software in commercial aircraft, DO-178C [15] provides the latest version of guidance regarding software aspects of certification and is used by the aviation industry and regulators as a means of compliance with airworthiness regulations. At a high level, DO-178C helps manufacturers achieve two main goals: 1) to demonstrate that software complies with its requirements (intended functionality), and 2) that it does nothing unexpected (unintended functionality). Unintended functionality or unintended behavior can therefore be defined as software behavior than cannot be traced back to any requirement. Unintended behaviors are the "unknown unknowns" that need to be detected and eliminated from software to ensure that there are no surprises during aircraft operation that might impact safety.

This can be a challenge for many systems based on machine learning technologies such as neural networks. In this paper we will refer to such technology as *learning-enabled components (LECs)*. In a typical LEC, much of the complexity and design information resides in its training data rather than in the actual models or code produced in the training process. One of the key principles of avionics software certification (covered in DO-178C) is the use of requirements-based testing along with structural coverage metrics. These activities not only demonstrate compliance with functional requirements, but are intended to show the absence of unintended functionality. However, it can be difficult to precisely state requirements for LECs, especially those implementing perception functions. Even when requirements are available, it is usually not possible to associate any particular lines of code with a specific requirement. Furthermore, complete structural coverage (using current metrics) can be achieved for a typical neural network with a single test case, but this provides almost no confidence in its correctness. Showing that a component or system is correct and does not do harm because of behaviors that were unintended by designers or unexpected by operators is a critical aspect of the certification process.

An illustration of unintended behavior made the news when a self-driving car being tested by Uber in Tempe, AZ in 2018 ran into a pedestrian [13]. According to documents released as part of the government investigation, the learning-enabled software in the vehicle was not designed to detect pedestrians outside of a crosswalk. This incident highlights at least two challenges faced by LECs: 1) producing complete and precise requirements for perception-based systems, and 2) responding safely to unexpected or novel inputs to the system.

In another incident the same year, a Tesla vehicle operated using its Autopilot feature crashed into a concrete lane divider [18]. The vehicle was in an autosteering mode with adaptive cruise control (maintaining a fixed distance from the car ahead) when it veered into the area between two marked lanes. One possibility that has been suggested is that the vehicle was confused by the concrete lane divider as the adaptive cruise system was designed to ignore stationary objects along the road to avoid false alarms. This may have resulted in unintentionally ignoring an actual hazard.

Given these challenges and the current state of the art, why would we consider using LECs in safety-critical systems? It is because LEC technologies are capable of providing advanced functionality that can be of tremendous benefit in many systems:

- LECs can implement functions that would not be pos-

sible with traditional methods. Perception functions for rapid identification and classification of objects in images can be implemented by deep neural networks. Complex highly non-linear functions that may not even have a closed-form solution can be learned from easily produced simulation data.
- LECs can provide reduced computation time for complex functions, leveraging parallelism in GPUs or FPGAs. They can be used to replace large table lookups, dramatically reducing memory requirements on resource-constrained platforms.
- LECs can learn from new data gathered during operation to continuously improve performance and adapt to new situations. Note, however, that in this paper we are limiting the scope to systems that are trained offline.

The challenge now for manufacturers and regulators is to determine how can we provide safety assurance for LECs that is comparable to systems built using traditional technologies.

This paper provides a summary of the current situation with regard to certification of LECs, focusing on unintended behavior. In section II we describe what certification standards require now for detecting and eliminating unintended behavior in commercial airplane software. In section III we consider the ways in which current methods are incompatible with LECs. Section IV describes new methods and technologies that may help address this problem. In section V we summarize recent activity to develop new certification guidance specifically for LECs.

II. Current Certification Guidance

In this section we provide an overview of current certification guidance for assuring the correctness of aircraft software.

A. DO-178C

The development process described in DO-178C begins with system requirements that have been allocated to software for implementation. These system requirements are subsequently refined into high-level requirements, low-level requirements, and software architecture, from which source code is produced and ultimately compiled into executable object code.

According to DO-178C, the purpose of software verification is to detect errors that may have been introduced during software development. More specifically, the software verification process must verify that "the Executable Object Code satisfies the software requirements (that is, intended function), and provides confidence in the absence of unintended functionality." (sec. 6.1.d).

Software verification activities in DO-178C center on requirements-based testing. *Coverage* refers to the extent to which a verification activity satisfies its objectives. Two specific measures of test coverage required in DO-178C are *requirements coverage* and *structural coverage*.

Requirements coverage analysis determines how well the requirements-based testing verified the implementation of the software requirements (section 6.4.4.1), and establishes traceability between the software requirements and the test cases. Requirements coverage analysis should show that test cases exist for each software requirement, and that the test cases satisfy the criteria for normal and robustness (abnormal range inputs) testing.

Verification that provides complete requirements coverage is not necessarily a thorough test of the software. For example:
- The software requirements and the design description may not accurately specify all of the behavior in the executable object code.
- The software requirements may be too abstract and do not ensure that all the behaviors implemented in the source code are tested.

Structural coverage analysis determines how much of the code structure was not executed by the requirements-based tests (6.4.4.2), and establishes traceability between the code structure and the test cases. Three different coverage metrics are defined, with the more rigorous metrics applied to code that is more safety-critical: 1) statement coverage, 2) decision (or branch) coverage, and 3) modified condition/decision coverage. These are all based on assessing the control flow among statements in the source code, including the logical expressions that govern that flow. An excellent tutorial on structural coverage in DO-178 can be found in [8]

DO-248C, *Supporting Information for DO-178C and DO-278A* [16], provides further rationale for structural coverage analysis. FAQ #43 states that structural coverage analysis (and associated resolution of coverage shortcomings) are intended to:
- Show that the code structure was verified to the degree required for the applicable software criticality level.
- Establish the thoroughness of requirements-based testing.
- Support demonstration of the absence of unintended functions.

The FAQ observes that requirements-based testing alone cannot completely verify the absence of unintended functionality. This is because code that is implemented without being linked to requirements may not be exercised by requirements-based tests, and this code could result in unintended functionality. Structural coverage analysis was added to address this problem. If requirements-based testing shows that all intended functions are properly implemented, and if structural coverage analysis shows that all existing code is reachable and adequately tested, these two together provide a greater level of confidence that there is no unintended functionality.

B. DO-331

Guidance for model-based software development is provided in DO-331 [17], *Model-Based Development and Verification Supplement to DO-178C and DO-278A*.

It is common in model-based development (MBD) processes for source code to be generated directly from a design model that corresponds to traditional low-level software requirements. MBD processes introduce a new concern related to unintended behavior. Model elements implementing unintended behavior could inadvertently make it into the design model and subsequently result in source code through the code generation process. Such code may not be detected by structural coverage testing because it traces to a model element that is now part of the low-level requirements (i.e., the corresponding code is exercised by a requirements-based test case).

For this reason, DO-331 introduced new objectives for model coverage analysis (MB.6.7) showing that all model elements have been exercised by requirements-based model verification activities. To facilitate this analysis, every model must have identified requirements from which the model was developed (MB.1.6.1).

Model coverage analysis is defined as an analysis that determines which requirements expressed by a design model were not exercised by verification based on the requirements from which the model was developed. The stated purpose is to support the detection of unintended behavior in the design model.

Coverage of the requirements from which the model was developed must be achieved by the requirements-based verification cases. Model coverage analysis is different from structural coverage analysis and therefore model coverage analysis does not eliminate the need to achieve the objectives of structural coverage analysis per DO-178C section 6.4.4.2.

C. DO-333

Guidance for the use of formal methods in the certification process is provided in DO-333, *Formal Methods Supplement to DO-178C and DO-278A*. This document describes how mathematical analysis tools based on formal logic can be used to satisfy verification objectives. It includes provisions for performing coverage analysis when using formal methods.

Requirements coverage is essentially unchanged from DO-178C in that applicants must demonstrate that all requirements have been verified by formal analysis, and establish traceability between the software requirements and the verification cases. The need for structural coverage analysis, however, is based on the impracticality of achieving exhaustive testing and the consequent need to establish metrics to ensure that the testing performed is rigorous and sufficient. Unlike testing, the use of formal methods may provide an exhaustive assessment of the software. However, additional activities are required to achieve comparable coverage analysis.

The additional activities are intended to show that the software requirements are complete and precisely specified, that the analysis itself is complete (corresponding to a mathematical proof), and that all assumptions in the analysis have been justified.

Formal analysis can show that there are no inputs to the system that result in incorrect or unintended behaviors. However, this does not demonstrate the absence of extraneous code — it just shows that such code cannot impact the observable software behavior. Additional activities (analyses or reviews) must be performed to detect unintended dataflow relationships in the software, and to detect unreachable or deactivated code.

III. Challenges Presented by LECs

LECs present unique challenges that may be barriers to the use of traditional, model-based, or formal methods guidance currently defined in DO-178C and its supplements. Fundamentally, this is due to the reliance on requirements-based testing (or verification) and structural coverage metrics, as described in the previous section.

It should be obvious that requirements-based testing requires requirements. Clearly stated requirements are also a necessary part of MBD and formal methods development and verification processes. However, the ability to implement complex functionality by learning from data in the absence of clear requirements and to generalize when faced with new data is actually a strength of machine learning. It may be possible and necessary to retroactively add high-level functional requirements to LEC designs, but this is often not the usual starting point.

Even when requirements are available, it is still difficult to determine whether enough testing has been performed to provide a complete assessment of an LEC design and provide confidence in the absence of unintended behaviors.

Structural coverage metrics were constructed with the understanding that much of the complexity of traditional software is manifested in the logical decisions that are being implemented. This logic should be traceable to specific software requirements. When requirements-based tests fail to exercise part of the software logic as revealed by structural coverage metrics, it is reasonable to conclude that something is amiss (either a missing requirement or some unintended function).

Since neural networks do not primarily implement logical decisions, structural coverage can usually be achieved with one test case (or possibly a small number of them). Individual lines of code in the software representation of a neural network do not represent design choices that implement specific requirements. Therefore, current structural coverage metrics are not helpful in identifying unintended behaviors.

Software testing remains a critical challenge for machine learning systems and new approaches will be needed to take the place of traditional assurance methods. An excellent summary of the current state is provided in [12]. Some of the relevant additional challenges include:

- Missing test oracles. Automated oracles defining the correct outputs of LECs are typically not available and great effort must be dedicated to manually labeling of test data.
- Infeasibility of complete testing. LECs typically must deal with large amounts of data and testing is rarely able to cover all valid combinations of inputs. Neuron coverage metrics inspired by traditional structural coverage metrics have been shown to miss erroneous behaviors. Test suites providing complete neuron coverage do not identify networks that are vulnerable to trivial adversarial attacks. Extensions looking at neuron interactions or boundary values can be limited by scalability.

- Quality of test datasets. Good LEC training requires a large dataset, but effective testing must be done with yet another large and independent dataset. Creating or deriving high-quality test datasets remains a challenge.
- Vulnerability to adversaries. LECs have been shown to be vulnerable to attacks where small input modifications lead to misclassifications or significant loss of accuracy. In some cases, the adversarial inputs demonstrated are unrealistic (setting specfic pixel values in an image) and could be ruled out as sufficiently improbable. Useful test cases must be able to expose these vulnerabilities yet still be realistic for the input domain.
- Evaluating robustness. Small changes in LEC inputs may result in large changes in its outputs, and it is difficult to generate useful test data to measure how well the system tolerates noise.

Before moving on, let us consider feedback control systems and whether their structure and implementation provide a precedent for addressing LEC assurance concerns. Feedback control is used in safety-critical aviation systems and is obviously certified using current guidance and processes. Perhaps the same approach or something analogous can be used for LECs.

A typical control system can be described by the equation $\dot{x} = Ax + Bu$, where x is the state vector for the system and u is a vector of inputs. This equation for $\dot{x}$ describes the system dynamics, computing the derivative for each state element based on the current state and inputs. The linear algebra computations involved bear some similarity to the computations performed in a neural network inference model (ignoring the activation functions in the neural net). We might say that there is no real traceability between particular entries in the A and B matrices and the system requirements, and that a neural network should be no more or less challenging from an assurance standpoint.

However, there are two important differences:

- For a typical aircraft feedback control system the matrices may have tens of entries, but a typical neural network (trained to perform a perception task) may include millions of weights.
- While the entries in the control system matrix may not trace directly to requirements, their specific values are determined by an extremely well-understood theoretical framework, and the control characteristics implemented using that framework definitely trace to the system requirements.

In conclusion, LECs and their software implementations break many of the assumptions that are the basis for current certification processes. In particular, the design intent cannot be inferred from an examination of an LEC model or its software implementation.

IV. New Assurance Approaches for LECs

LEC assurance for safety-critical applications in an active area of research. New approaches are being developed and demonstrated, including the use of formal methods for LEC verification, new testing methods and coverage metrics, and architectural mitigations using run-time monitoring.

New formal methods tools are in development that permit mathematical analysis of neural network models. These are currently limited by scale and the need to precisely define requirements for analysis, but they are making rapid progress and have been used to prove critical robustness properties for real systems.

A recent example is the Marabou framework for verifying deep neural networks [11]. Marabou combines an SMT (satisfiability modulo theories) solver with a custom linear programming engine to answer queries about neural network properties. It can handle networks with piecewise linear activation functions (such as rectified linear units, or ReLUs) and topologies including fully-connected feed-foreword and convolutional neural networks. Marabou performs high-level reasoning on the network to reduce the search space and also supports parallel execution to further enhance scalability. Marabou accepts multiple input formats, including protocol buffer files generated by the TensorFlow framework.

Marabou's predecessor (Reluplex) was used to successfully analyze and prove behavioral properties about the ACAS-Xu collision avoidance system for unmanned air vehicles [10]. In this system, a 2GB lookup table was reimplemented as 45 neural networks, occupying a total of 3MB. Safety properties of these neural networks were verified using Reluplex.

LECs implementing perception functions are still beyond the reach of formal verification tools due to their size (millions of neurons and weights) and lack of precise requirements.

New testing methods are being developed for neural networks and other LECs. An important aspect of this work is defining new coverage metrics that can be used to improve the completeness of testing and increase confidence in the absence of unintended behaviors.

A new approach to test generation based on manifold space techniques is described in [2]. A manifold is a topological space that is locally Euclidean, like the surface of the earth, allowing us to reason about nearby points in a straightforward way even though the actual space is quite complex. The approach is based on the premise that patterns in a large input data space can be effectively captured in a smaller manifold space, from which similar yet novel test cases can be sampled and generated. Tools exist to learn and capture this manifold space, and a search technique is applied to efficiently find fault-revealing inputs. Experiments show that this approach enables generation of thousands of realistic yet fault-revealing test cases efficiently even for well-trained models. This approach may also provide the basis for a meaningful metric of completeness for a test dataset based on the embedded manifold.

Since it is difficult to demonstrate assurance by examining the LEC design (as is assumed by existing certification processes), other approaches based on run-time monitoring and enforcement may be effective.

Run-time assurance architectures add high-assurance components to the system to ensure that an LEC cannot cause unsafe or unintended system behaviors. Run-time monitors continuously check variables related to the system state, inputs to the LEC, or outputs produced by the LEC and intervene to switch to a backup function that is proven to be safe. Monitors may be used to detect anomalous inputs that are outside of the data distribution used to train the system and therefore could lead to unintended behavior. The main idea is that system performance is provided by the LEC while system safety is guaranteed by high-assurance components (though with lower performance).

In the DARPA Assured Autonomy project we have used a run-time assurance architecture based on the ASTM F3269-17 standard for bounded behavior of complex systems [1], also known as a simplex architecture [14]. The standard provides guidance for mitigating unintended functionality through the use of run-time monitors. The LEC may still contain unintended functionality, but the architecture ensures that there will be no impact on system safety. This approach essentially uses the verified properties of the architecture, run-time monitor, and safety backup functions to justify a reduced level of criticality for the LEC.

The "TaxiNet" run-time assurance demonstration is described in [4]. The baseline system consisted of the aircraft (or simulation), the guidance LEC, a controller for steering the aircraft, and the Vehicle Management System (VMS) which manages actuators on the aircraft and integrates other autonomy functionality. The LEC was implemented as a deep neural network (DNN) trained to estimate the cross-track error (CTE) of the aircraft (position left or right of the runway centerline) based on images from a forward-looking camera on the aircraft. Since the images are 360x200 pixels, the resulting LEC is larger than can be analyzed by current formal methods tools for DNNs, such as Marabou.

The run-time assurance architecture added four different run-time monitors (three for system safety, one for LEC confidence), a Monitor Selector for choosing which monitor to use at any time, and a Contingency Manager to determine when intervention is needed to maintain safety and what action should be taken. In this example, the safety actions available (via the VMS) were to reduce the commanded aircraft speed or to use the brakes to halt the aircraft.

We tested the architecture in a variety of simulated environmental conditions with both well-trained and poorly-trained LECS to assess baseline performance, intervention of the assurance architecture in the presence of LEC errors, and absence of unnecessary intervention (false alarms). We found that the architecture performed in accordance with expectations in all scenarios. In every case where a faulty LEC caused the aircraft to deviate from the required centerline tracking performance, the assurance architecture detected the condition and slowed or halted the aircraft. At no time was the aircraft allowed to depart from the paved runway. Furthermore, we did not observe any false alarms, meaning that the architecture never intervened when the aircraft was performing within requirements and correctly tracking the runway centerline.

V. New Certification Guidance

There are a number of parallel efforts underway to develop new certification guidance supporting the use of LECs in aviation applications.

The European Union Aviation Safety Agency (EASA) has published its *Artificial Intelligence Roadmap 1.0* [6] which establishes an initial vision for safety in the development of LECs in the aviation domain. Important contributions include the definition of "trustworthiness building blocks" for LECs and publication of a proposed timeline, calling for first approvals of LECs (used in an advisory role) around 2025.

EASA has also published a report on a research effort addressing the challenges posed by the use of LECs in aviation entitled *Concepts of Design Assurance for Neural Networks* [3]. The report describes a W-shaped development life-cycle that adds training processes in the middle of the traditional V-shaped life-cycle. It also investigates theoretical and practical *generalization bounds* as a means of establishing confidence that an LEC will perform as intended when faced with novel inputs.

Another recent publication is EASA's concept paper on *First Usable Guidance for Level 1 Machine Learning Applications* [7]. "Level 1" in this context refers to applications in which the LEC is providing assistance to a human operator as opposed to human/machine collaboration or autonomy. The report describes a set of candidate objectives to be satisfied by developers of Level 1 LECs and is intended to provide visibility into regulatory expectations for such systems.

These reports are being used as inputs by the joint SAE committee G34 and EUROCAE working group WG-114, *Artificial Intelligence in Aviation*. WG-114 was formed in 2019 and shortly thereafter merged with G34 to work together to address certification of aeronautical systems implementing artificial intelligence technologies. The goals of the joint committee are to create and maintain reports and recommended practices on the implementation and certification aspects related to AI technologies, including both airborne and ground-based systems needed for the safe operation of aerospace vehicles. The principle work product of the committee will be a new standard: AS6983 Process Standard for Development and Certification/Approval of Aeronautical Safety-Related Products Implementing AI.

The detection and elimination of unintended behaviors has been identified by the committee as a key concern to be addressed by the guidance documents ultimately published. A number of means to achieve this goal are under consideration, looking at different points in the development life-cycle. These include assessments of the completeness and representativeness of the training and test datasets, new structural coverage metrics (though the specifics have yet to be agreed upon), and the use of run-time assurance approaches at the system design level.

One final effort worth noting is the Overarching Properties initiative, which followed from the 2016 U.S. Federal Aviation Administration (FAA) "Streamlining Assurance Processes Workshop." The Overarching Properties (OP) are the product of a multi-year, international effort to develop a minimum set of properties sufficient for use in the approval process. A more detailed explanation of the OP can be found in [9].

The three OP as currently defined are:

- Intent: The defined intended behavior is correct and complete with respect to the desired behavior.
- Correctness: The implementation is correct with respect to its defined intended behavior, under foreseeable operating conditions.
- Innocuity: Any part of the implementation that is not required by the defined intended behavior has no unacceptable impact.

Innocuity captures the goal of eliminating unintended behavior — that is, any behavior that is not included in the defined intended behavior of the system. Innocuity specifically does not restrict the implementation to only contain items which are required by the defined intended behavior. Such things may be necessary when a system is implemented from previously developed items. Rather, it requires that nothing extra in the implementation can negatively affect safety.

VI. Conclusion

For most applications, the best performance that may be currently expected from an LEC (for image classification, for example) is around 99% correct. This is very far from the 10^{-6} probability of failure per hour that may be required for a reliable subsystem or 10^{-9} for aircraft systems with the highest integrity requirements.

We may be better off not thinking of a safety-critical LEC as software that must be verified to meet its allocated system requirements, but rather from the perspective of hardware reliability. Fault-tolerant design principles can then provide guidance for how to use redundancy and fail-safe design mechanisms to achieve the higher levels of reliability required in aircraft systems.

This leaves us with three alternatives:

- Determine that the reliability actually achievable by an LEC is acceptable for a given application.
- Architect the system using principles of fault-tolerant design to achieve higher levels of reliability.
- Reject the use of an LEC and implement the system using traditional non-learning-based methods.

At the present time, there is no agreed upon solution to the detection and elimination of unintended behavior in LECs. The fundamental problem is that LECs were never imagined to be able to provide guarantees of correctness. They are best applied to challenging problem domains where traditional approaches are less effective but where the ability to learn and generalize from large amounts of data can provide unique capabilities. LEC assurance for safety-critical systems remains an active area of research. In all likelihood, certification guidance under development will rely on a combination of techniques, each contributing some bit of assurance evidence.

Acknowledgment: This work was funded by DARPA contract FA8750-18-C-0099. The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for Public Release, Distribution Unlimited.

References

[1] ASTM F3269-17. Standard practice for methods to safely bound flight behavior of unmanned aircraft systems containing complex functions, 2017.

[2] Taejoon Byun, Abhishek Vijayakumar, Sanjai Rayadurgam, and Darren D. Cofer. Manifold-based test generation for image classifiers. In *IEEE International Conference On Artificial Intelligence Testing, AITest 2020, Oxford, UK, August 3-6, 2020*, pages 15–22. IEEE, 2020.

[3] Jean Marc Cluzeau, Xavier Henriquel, Georges Rebender, Guillaume Soudain, Luuk van Dijk, Alexey Gronskiy, David Haber, Corentin Perret-Gentil, and Ruben Polak. Concepts of Design Assurance for Neural Networks (CoDANN). https://www.easa.europa.eu/sites/default/files/dfu/EASA-DDLN-Concepts-of-Design-Assurance-for-Neural-Networks-CoDANN.pdf, Mar 31, 2020.

[4] Darren D. Cofer, Isaac Amundson, Ramachandra Sattigeri, Arjun Passi, Christopher Boggs, Eric Smith, Limei Gilham, Taejoon Byun, and Sanjai Rayadurgam. Run-Time Assurance for Learning-Based Aircraft Taxiing. In *Digitial Avionics Systems Conference (DASC)*, 2020.

[5] CSPAN. Defense Department Briefing. https://www.c-span.org/video/?168646-1/defense-department-briefing, Feb 12, 2002.

[6] EASA. Artificial Intelligence Roadmap 1.0: A human-centric approach to AI in aviation. https://www.easa.europa.eu/sites/default/files/dfu/EASA-AI-Roadmap-v1.0.pdf, 2020.

[7] EASA. EASA Concept Paper: First usable guidance for Level 1 machine learning applications. https://www.easa.europa.eu/sites/default/files/dfu/easa_concept_paper_first_usable_guidance_for_level_1_machine_learning_applications_-_proposed_issue_01_1.pdf, April 2021.

[8] Kelly Hayhurst, Dan Veerhusen, John Chilenski, and Leanna Rierson. A practical tutorial on modified conditiondecision coverage. NASA Technical report TM-2001-210876, 2001.

[9] C. Michael Holloway. Understanding the Overarching Properties. NASA Technical report TM-2019–220292, 2019.

[10] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2017.

[11] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. The marabou framework for verification and analysis of deep neural networks. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 443–452. Springer, 2019.

[12] Dusica Marijan and Arnaud Gotlieb. Software testing for machine learning. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 13576–13582. AAAI Press, 2020.

[13] Aarian Marshall and Alex Davies. Uber's self-driving car didn't know pedestrians could jaywalk. https://www.wired.com/story/ubers-self-driving-car-didnt-know-pedestrians-could-jaywalk/, Nov 5, 2019.

[14] Jose Rivera, Alejandro Danylyszyn, Charles Weinstock, Lui Sha, and Michael Gagliardi. An architectural description of the simplex architecture. Technical Report CMU/SEI-96-TR-006, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.

[15] RTCA DO-178C. Software considerations in airborne systems and equipment certification, 2011.

[16] RTCA DO-248C. Supporting Information for DO-178C and DO-278A, 2011.

[17] RTCA DO-331. Model-Based Development and Verification Supplement to DO-178C and DO-278A, 2011.

[18] Jack Stewart. Tesla's autopilot was involved in another deadly car crash. https://www.wired.com/story/tesla-autopilot-self-driving-crash-california/, Mar 30, 2018.