NASA/CR–2020–5000437

MBSwE for Autonomous Systems with Reuse: Software Assurance Best Practices and Gaps Technical Report on Gap Analysis

Johann Schumann
SGT, Inc., NASA Ames Research Center, Moffett Field, CA 94035, USA

Yuning He
NASA Ames Research Center, Moffett Field, CA 94035, USA

January 2020

NASA/CR–2020–5000437

MBSwE for Autonomous Systems with Reuse: Software Assurance Best Practices and Gaps Technical Report on Gap Analysis

Johann Schumann
SGT, Inc., NASA Ames Research Center, Moffett Field, CA 94035, USA

Yuning He
NASA Ames Research Center, Moffett Field, CA 94035, USA

National Aeronautics and
Space Administration

Ames Research Center, Moffett Field, CA 94035

January 2020

Executive Summary

Examples of gaps include difficulties to specify safety requirements, coverage of training and test sets for deep learning, and the missing explainability of decisions made by autonomous AI systems.

Contents

List of Figures

Chapter 1

Introduction

Modern unmanned systems require substantial autonomous capabilities to enable ambitious missions in space. Autonomous components (AUCs) range from model-based diagnostics and prognostics components to artificial intelligence (AI) based systems for analyzing and "understanding" the environment, making informed decisions, and acting on those decisions. Software components of a mission with autonomous operations are both safety- and mission-critical.

Failures or wrong decisions can put the mission at risk, for example, an unmanned Europa mission. Autonomous aircraft of cars can endanger human life. Because of the huge state space and often probabilistic nature of such software systems, combined with a dynamic and unknown environment, current best practices of software assurance (SWA) are not sufficient for autonomous systems.

In this report, we analyze gaps that occur in Software Assurance of an autonomous (software) system (AUC). We identify several distinct areas, where gaps occur:

Requirements Which gaps exist in formulating and checking of requirements for AUC?

Code/Algorithms Autonomous systems may need to employ specific algorithms for autonomous operations, for example machine learning, search, or probabilistic algorithms. For many of those algorithms V&V is still at its infancy, so large gaps in SWA can be expected.

Data Many AUCs rely on large amounts of data to operate. A typical example is the data that is used to train a Deep Neural Network (DNN) for image understanding. How can assurance be performed for such data? Handling and quality assurance for training data comprises a substantial gap.

Process It is to be expected that an autonomous system might require more run-time assurance than a traditional software system. Gaps exist on level of specification, architecture, and effectiveness of such run-time assurance components. Furthermore, the heavy use of data and off-line machine learning is not yet properly integrated into the software (assurance) process

In this report, we will focus on each of these topics. In Section 2, we will present a short overview of the generic structure of an autonomous system. Section 3 focuses on challenges and gaps with respect to requirements for AUCs followed by overview of algorithms and languages, which are prominently used for Autonomous Systems (Section 4). Algorithms will be characterized and gaps identified. In Section 5, we focus on three different kinds of algorithms: search-based, data-based, and Neural networks. Then we discuss topics and gaps in the assurance of "data" (Section 6) followed by summary and conclusions in Section 7.

Chapter 2

Anatomy of an Autonomous System

The high-level architecture of a typical autonomous system (AS) is shown in Figure 2.1. It reflects an abstraction of system architectures found, e.g., in [1,2,3]. The hardware system (e.g., spacecraft, UAS, robot, car) is equipped with numerous sensors that allow the AUCs to sense and perceive the environment as well as its own state.

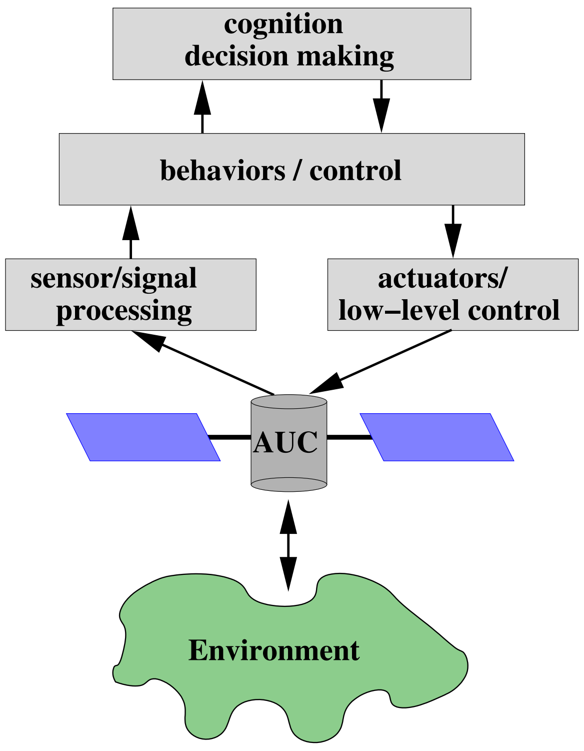

Figure 2.1: Typical High-level architecture of an autonomous system

Sensor drivers and processing units are in charge of communicating with the hardware and to perform basic data processing (e.g., filtering or data conversion). These sensor data are then used by components on the *behavioral* level to perform basic behavioral tasks. Typically, feed-back control (e.g., PID control), mode logic, or navigation components are found on this level. Low-level (path) planners can be present on this level as well.

Finally, on the *cognitive level*, we find components that try to "make sense" of environment and system status, like image understanding, sensor fusion, diagnostics, prognostics, and root cause analysis. The information provided by these system are used as a basis for the *decision-making* components. Decisions and high-level plans are then percolated down to the behavioral components for refinement and execution, which will be carried out through control of the system's actuators.

Some of these components can be considered as "traditional" software, which means that SWA approaches are mature and well studied. Examples include drivers for sensors and actuators, feedback controllers, and middle-ware (e.g., cFS/cFE). Although all layers must work together to achieve the autonomous goals, we will mainly focus on the behavioral and cognitive layers and their interactions.

Chapter 3

Requirements

Requirements for a system (hardware and/or software) must concisely describe what the system shall or shall not do. Different levels of requirements are used to lay down the description and required behavior of the overall system and its components on different levels of granularity and details. For traditional software systems, numerous methods and tools for the acquisition and management of requirements exist, as well as tools that support their analysis.

When considering an autonomous system as depicted in Figure 2.1, the question will be, inhowfar the "autonomous nature" of the system impacts formulation and analysis of requirements.

On a high level of requirements, very generic high-level requirements about the mission and the system will be defined. In contrast to a traditional system, an autonomous system usually will have a substantially different mission profile. Instead of command-reaction requirements ("after receiving a launch command, ignite the thruster"), the autonomous system must be able to correctly perceive its status and react properly in myriads of possible circumstances. A damage-adaptive system, for example, might be required to "in all unforeseen circumstances, the battery voltage must not be below 10V". It is obvious that such a requirement cannot be tested properly.

More specifically, we can identify the following challenges when working with requirements for autonomous systems:

Unknown environments cannot be specified in requirements (or they would be known). Yet, an AUC must be able to operate in unknown and changing environments.

Complex tasks which need to be carried out by the autonomous system must be specified and potentially broken up into smaller pieces (e.g., perception of landscape, identification of target, path-planning to reach target). Such complex tasks require a tight interaction between all SW components of the system (the AUCs and the non-autonomous components), which needs to be reflected in the requirements.

Health status of the autonomous system is comprised of diagnostic and prognostic information, as well as their interpretation. For larger AUCs, requirements regarding system health and status can be extremely complex.

Cognition is an extremely important task within the autonomous operations. Sensor and environmental data must not only be processed and analyzed, but must undergo a cognitive process in order to be useful for decision making. Typical example include image understanding or the prediction of the behavior of another system.

Probabilistic behavior of the system must somehow be captured in the requirements. For example, a stop-sign should be recognized in 99.99% of all cases.

Ethical behavior is playing an increasing role for autonomous systems, which interact with humans or a sensitive environment. Typical examples include real-time decisions to be made by an autonomous car ("kill the passenger or the pedestrian crossing the street?") or the trade-off between scientific return versus contaminating/damaging an off-world environment.

Emerging behavior can be the result of operations of an autonomous system. E.g., the autonomous system might adapt toward unforeseen circumstances, might "invent" new modes, or even might take over earth as depicted in numerous Science Fiction movies (e.g., Matrix, Terminator).

For all the above challenges and special cases of requirements, we must ask:

1. Do we have the language/formalism to concisely express the requirement?
2. Can the requirement be written in a compact and concise manner, which is still human-understandable?
3. Can such a requirement be (formally) analyzed and/or tested against the actual system?

Table 3.1 lists AUC-specific characteristics for requirements against these questions and will thus identify major gaps in the ability to capture, express, and use the requirements for V&V.

Table 3.1: Requirements characteristics for AUC and challenges. Y=Yes, (Y)=limited, ?=unknown

AUC characteristics	Req. language	Compact/Concise	Analysis	Test
Unknown environment	Y	(Y)	?	?
Complex Tasks	Y	Y	(Y)	Y
Health status	Y	(Y)	Y	Y
Cognition	(Y)	?	(Y)	(Y)
Ethical behavior	?	?	?	?
Emerging behavior	?	?	?	?

Chapter 4

Software Algorithms and Languages

The software load of an autonomous system can be quite heterogeneous. As depicted in Figure 2.1, it usually contains a large number of components that closely interact with each other. Many software components, in particular on the lower layers consist of algorithms and software, traditionally used in safety-critical systems (e.g., sensor data processing, feedback control, or middle-ware). For those components, mature methods and tools exist for their verification and validation (V&V).

4.1 Kinds of Algorithms used in AUCs

Here, we therefore focus on the "hard" autonomous algorithms, which most often, can be found on the cognitive and decision-making layer of the architecture. We can distinguish the following types of algorithms:

Rule based: In a rule-based system, all decisions are made along predefined rules, which have been set up during the development of the system. Such rules can be implemented as if-the-else cascades, or, among others, finite state machines. There also exist several program languages that are specifically tailored for rule-based systems (e.g., Prolog, OPS5, CLIPS).

Whereas the rules are usually easy to understand by the human expert, their sheer number and questions of correctness and consistency can make V&V of rule-based systems extremely hard. The software-structures used to implement rule-based systems are in most cases straight-forward, so that in this area, only few SWA gaps exist. However, if search is needed to find applicable rules, the situation changes (see below).

A typical rule-based system is the TCAS on-board collision-avoidance system [4]. Although used for commercial transports, the operations of TCAS shows many characteristics of an autonomous systems. [5] show challenges and difficulties to V&V such a system.

Planning: For most automated and autonomous systems, planning and plan execution play important roles. Virtually all satellites and spacecraft have the capability to execute plans. There, sequences of actions (e.g., control thruster, make photo, trigger science experiment) are executed, depending on the current state of the system. In traditional systems, these plans are developed on the ground and up-linked for execution. Higher degrees of automation and autonomy require the adaptation of plans or even the automatic in-situ construction of new plans.

For example, PLEXIL [6] is a planning execution system used for many NASA space and Aeronautics missions. The PLEXIL execution engine has a small computational footprint and does not require dynamic memory for its execution. [7] describes formalization and V&V approaches for the planning system PLEXIL.

The on-board construction of plans, however, requires search and optimization algorithms, which will be described below.

Statistical Algorithms and Filtering: Such algorithms are mainly used for processing of sensor data, sensor fusion, prognostics, as well as probabilistic reasoning.

Typical filtering algorithms, like the Kalman filter and its variants (e.g., EKF, UKF), have been around for a long time and are well used in embedded and safety-critical applications. Their underlying recursive least-linear square approach is well understood and can be executed in a fixed-time steps without the need of dynamic memory.

Particle filters on the other hand can deal with more complex probability distributions, but require more computational resources and might, for some variants, even use random initializations. Still, PF algorithms can be executed in a fixed time step.

Probabilistic reasoning algorithms (e.g., a Bayesian graph reasoner) can be used to determine the likelihood of a certain situation, given observables (e.g., sensor information) and their probability. Bayes reasoners are typically used for fault detection and diagnosis [8]. The reasoning steps can be executed with a fixed worst-case execution time and fixed memory by compiling the graph structure into an arithmetic circuit [9].

Machine Learning (offline): Many AUCs contain algorithms that are controlled by parameters that have been determined using machine learning algorithms on often large amounts of data. The actual learning—often called training—is performed during the development of the AUC. Then, the parameter values are frozen and remain unchanged during system deployment. The most prominent example is a Deep Neural Network (DNN), which will be discussed in detail in Section 5.

The execution code for such an algorithm like a DNN is very simplistic, does not require dynamic memory and has a fixed worst-case execution time. All the "knowledge" of such a component is sitting in its parameters ("weights"), which have been estimated by an external machine-learning algorithm. During deployments, these weights do not change, hence the name "pretrained NN".

High dimensions of inputs and large numbers of parameters often require the use of non-conventional hardware, e.g., GPUs (Graphical Processing Units) or Tensorflow [10] for the training of such networks during system development time.

Optimization (numerical): these kinds of algorithms try to iteratively find a minimum (or maximum) of a high dimensional function. This class of numeric functions is very generic and can be used for multiple purposes in autonomous systems: trajectory planning, adaptation, machine-learning applications, or sensor-related applications.

Typically such an algorithm contains a loop, which must be executed until a specific criterion is fulfilled, e.g., the error $E < \theta$. Even if each individual execution of the loop has a fixed execution time, there is no way to determine, when the algorithm actually terminates. Therefore, such algorithms are not used in traditional safety-critical software, where a fixed execution time is required. However note, that some optimization tasks can be solved by recursive methods like the Kalman filter.

In addition, multivariate optimization algorithms can not be guaranteed to find the global minimum or maximum. Rather, they might only find local minima, which might cause questions about validity and suitability of the solution.

Symbolic/discrete search: Whereas optimization algorithms discussed above perform a search in a high-dimensional numerical space, symbolic or discrete search enumerates or traverses search spaces, which consist of discrete states. Typical examples for such algorithms include automatic logic reasoners, Model Checkers, or SAT solvers. Given a Boolean formula, a SAT solver tries to find a variable assignment that makes the Boolean formula true. The discrete state spaces, which can be huge, are traversed in a clever manner to reduce search time in many instances. Such algorithms find their application in planning, reasoning about the system's health status, contingency planning, and many other AUC applications.

Typically, such search algorithms require dynamic memory and do not have a bounded execution time.

Machine Learning (online): Finally, if the autonomous system has to adapt toward unforeseen situations or needs to learn new things during deployment, then on-line machine learning algorithms are being used. They can be of various complexity and can use filtering, optimization, or even discrete search.

However, they might use dynamic memory, don't have a fixed worst case execution time, and are, in general, very hard to verify and validate.

4.2 Algorithm Characteristics

The members of each algorithm group presented above can be characterized by a number of specific features. Here, we focus on features that have a substantial impact on the ability to verify and validate (V&V) such algorithms. These features include

Dynamic Memory (DM) Dynamic memory allocation "malloc" are traditionally hard to V&V and numerous bugs (like memory leak, use-after-free) can occur. Standards for traditional safety-critical software do not allow the use of dynamic memory allocation.

No Worst Case Execution time (NWCET) Most embedded software systems are executed in a way, where each component is being executed at a given rate, e.g., 80Hz. This automatically limits the maximal execution time per invocation. Overrun can lead to hard-to-detect and dangerous errors. Therefore, most safety-critical software systems require that each SW component has a worst case execution time (WCET) lower than the component rate. WCET analysis is a standard V&V technique, but it fails if algorithms do not have a fixed WCET. Typically such algorithms contain convergence loops, potentially unbounded tree or graph search, or solve machine learning tasks. The proper V&V of such algorithms within an embedded environment is still largely unsolved.

Non-determinism (NDET) Non-determinism of a calculation can be caused by effects of parallel or multi-threaded execution or the explicit use of (pseudo) random number generators. NDET leads to the situation that specific runs of the software cannot be exactly reproduced, making proper validation very hard. Therefore, embedded and safety-critical software is not supposed to contain any non-deterministic behavior.

Yet, many statistical and machine learning algorithms require the use of randomized sampling, therefore introducing NDET. V&V and certification of such algorithms comprises a large gap.

Non-Markovian (NMARK) A non-Markovian behavior, often called "non-deterministic" by controls experts, will always reproduce exactly the same run, given the same input data. It is therefore not necessarily non-deterministic in the sense mentioned above. The main characteristics of such an algorithm is that the current state x_t at time t cannot be calculated only based upon x_{t-1}, but information from all previous states (implicit or explicit) $x_0, x_1, \ldots, x_{t-1}$ are used by the algorithm. Due to this state history requirement, testing of such an algorithm can complicated and resource-consuming. Typically, on-line adaptive algorithms exhibit these characteristics.

Data-driven (DATA) In a data-driven algorithm, the control of the program is not present in the code (i.e., the structure of the program), but depends on data and parameters. Typical examples of data-driven algorithms include Neural Networks, Markov-Decision Processes, or large lookup tables. Also data bases (e.g., SQL) exhibit these characteristics.

In many cases, the code-fragments implementing such an algorithm (e.g., a DNN classifier) consist of very small and straight-forward generic algorithms, which can be fully covered (e.g., MC/DC coverage) by extremely few test cases. Therefore, code-coverage testing of such algorithms (as prescribed by, for example, DO-178C [11]) does not provide any assurance.

Rather, V&V techniques, specifically tailored toward these kinds of algorithms must be used. Here, one gap lies in the development and maturation of such techniques, and the second gap lies into incorporating of analysis results into the V&V and SWA process.

Any Time (AT) If an algorithm cannot be guaranteed to provide the full solution within a fixed time, sometimes "anytime" variants of the algorithm exist. Such an algorithm can provide, at each point in time, a partial or approximate solution. However, this trade between execution time and correctness or quality poses substantial V&V gaps, as for those algorithms, the quality and suitability of the approximate or partial solution must be studied.

Numerical (NUM) Many algorithms in the area of embedded and autonomous systems are numerical algorithms. Important V&V tasks include the analysis of potential roundoff errors, coverage testing, automatic test case generation (see, .e.g., [12]), robustness, and requirements for calculation accuracy. Although several techniques and tools exist to address these issues, most formal tools (like model checkers, static analyzers, or symbolic execution engines) only work for the domain of integer numbers. Therefore, numerous V&V gaps exists in this area.

Table 4.1 relates algorithm types from the previous subsection with their typical characteristics. Each non-empty field can mean a V&V gap.

Table 4.1: AUC algorithm types and their typical characteristics

Algorithm Type	DM	NWCET	NDET	DATA	AT	NMARK	NUM
Rule-based							
Planning system	X	X			X		
statistical/filtering			X	X	X	X	X
ML (offline)				X	X		X
Optimization		X	X	X			X
symbolic/discr search	X	X	X	X	X		
ML (online)	?	X	X	X	X	X	X

Chapter 5

AUC Algorithms

In this section we discuss, in more detail, three popular kinds of algorithms which are used to implement AUC functionality: search-based algorithms, data-driven algorithms, and—because of the current popularity—Artificial Neural Networks.

5.1 Search-based Algorithms

This category of algorithms includes numerical optimization algorithms as well as discrete algorithms. The former are mainly used to find optima, i.e., minima or maxima of an objective function in an often high-dimensional space. Discrete search is usually employed within rule-based decision systems as well as logic reasoning systems.

Although a large variety of such algorithms exist, they tend to have these common characteristics:

- the number of search iterations is not or cannot be limited. Typical control structures include while(E > theta){...} or while(converging){...}. This means, that such an algorithm has, in most cases, no upper bound for the run-time [13]. Traditional safety-critical software does not allow loops like that, as they make a worst-case runtime analysis impossible. Functions which unknown and potentially unlimited runt-time behavior are hard to V&V in general and thus form a gap in traditional V&V for safety-critical systems.
- Some tasks can be solved with any-time algorithms, which can provide, at any point in time, with an approximated solution, but then the assurance question arises about the usefulness and correctness of that partial answer.
- In many cases, search algorithms require large amounts of memory or dynamic memory. This again does not satisfy traditional assurance requirements for safety-critical software.
- Multivariate numerical algorithms cannot be guaranteed to find the global optimum [13]. Rather, they might get stuck in some local minimum. Here again, no assurance can be given, exhibiting a gap.
- Probabilistic search and optimization algorithms are used successfully in autonomous systems. However, their V&V is still in their infancy and gaps with respect to handle non-deterministic software as well as the other issues concerning search algorithms exist.

5.2 Data-driven Algorithms

In traditional safety-critical software, “data” show up mainly in the form of (engineering) constants and (small) lookup-tables. Control is exclusively done by programming language constructs (if-then-else, state machines, etc.). Other than that, data-bases are used sometimes, which are comprised of well-understood algorithms and mature implementations.

Autonomous systems, however, use numerous variants of “data-driven” software, where “a lot of information” is hidden in the data. The most prominent example is a (Deep) Neural network.

Given a high-dimensional input x it calculates a value y using a large number of numerical parameters (or weights) W, which have been estimated during development time by machine learning.

The result can be a numeric value, a discrete label (e.g., an index of a recognized image), or a temporal sequence of values (time series).

The implementation of the actual algorithm is extremely simple and basically consists of some linear-algebra operations. This algorithm basically does not change between different applications. Rather, its customization toward the specific autonomy task is entirely driven by data.

Other algorithms with similar characteristics include, for example, Markov processes and their implementations as, for example, found in ACAS X [14,15]. Even some filtering and estimation algorithms used in Prognostics belong to that category.

Similarly, but working on discrete data and structures are graph-based algorithms as used, for example, for planning, Bayesian networks for reasoning, or graph/tree-based algorithms used for logic reasoning (e.g., SAT).

All these algorithms have in common

- the (code) structure of the algorithm can be very simple
- control of the algorithm’s behavior and output is entirely governed by data
- these data are established during system development time (e.g., for a pre-trained DNN) or even can change during operation (e.g., on-line adaptive systems)
- some systems have constant time and memory (e.g., neural networks), some have not (e.g., graph-based algorithms)

V&V gaps for such algorithms show up in different dimensions:

- algorithms like DNN algorithms are extremely small and simple. Traditional code-coverage testing, e.g., as prescribed by DO-178C [11], can be performed with almost no test cases. However, even a 100% MC/DC coverage does not say anything about the quality or correctness of the results calculated by the DNN. Therefore, the metric of code coverage is meaningless for such algorithms as the “meat” is hidden in the data. Therefore, other V&V techniques need to be used. Gaps include suitability of DNN specific V&V approaches/tools as well as their integration into the software V&V process.
- Quality and correctness of the design-time data. In most current DNN applications, the DNNs are being trained during design time using large data sets “training data”. A proper V&V must take into account the quality and coverage of these training data as well as the quality and suitability of the DNN architecture and the training

algorithm used. All these topics lead to active research areas but many questions are still unsolved.

- Quality and correctness of dynamically adapting data. When data (e.g., weights of a neural network) are adjusted during system operation, care must be taken that these updates do not diminish quality and correctness of the system behavior. These changes need to reflect the actual change of the system (e.g., due to some failure) or the environment, but must not capture unwanted or artificial effects.

  Techniques have been studied in the realm of traditional adaptive neural networks, but only very few and isolated approaches toward V&V exist [16]

- non-deterministic behavior. Although most data-driven algorithms are deterministic in nature, machine learning algorithms can introduce dependency on mission history or even introduce randomness. These effects make V&V extremely difficult and therefore identifies a SWA gap. (see Conclusions)

5.3 Artificial Neural Networks

Artificial neural networks, in particular Depp Neural Networks (DNN) have become very popular in many autonomy applications, mainly driven by the development of self-driving cars.

DNN are mainly used as classifiers or function approximators and are, in most cases, trained off-line. This means that the training of the DNN parameters ("weights") is performed during development time. The evaluation of a pre-trained network is a straightforward algorithm consisting of simple linear arithmetic operations (Figure 5.1). For example, for any node in a fully connected layer, its output is calculated as $o_j = f(\sum_i w_{ij} \cdot x_i)$ for inputs x_i, weights w_{ij}, and a nonlinear function f (like a tanh or Relu).

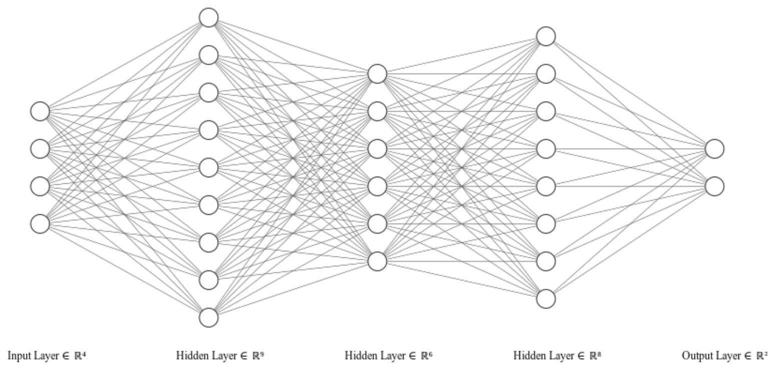

Figure 5.1: Typical architecture of DNN with several convolutional and fully connected layers. Figure generated via `http://alexlenail.me/NN-SVG/index.html`

More involved are DNNs, which are being trained on-line, i.e., the adaptation of weights occur during the deployment of the system. Such systems are currently used only in rare cases, but their application will become more important for highly autonomous missions, where the autonomous system must adapt toward unknown environments and unforeseen events. Advanced missions to icy moons will most probably require on-board learning neural networks.

From a SW assurance point of view, the on-board software portion for a pre-trained DNN is not problematic due to its simple algorithm. However, big V&V gaps concerning the behavior of the network. which is guided by its parameters. Here, selection of the network architecture, the training algorithm and regime, and the selection/availability of training and test data is of great importance. V&V approaches for DNN are still in its infancy, [17] gives an overview of V&V approaches suitable for DNNs.

[16] suggests the following activities related to V&V of NNs

1. separate NN from traditional software components
2. analyze network architecture
3. consider NN as function approximator
4. address opaqueness of NN
5. analyze learning algorithm
6. analyze selection and quality of training data
7. provide means for online monitoring

In their review article, [17] identify the following challenge categories concerning V&V of DNNs:

- State-space explosion
- Robustness
- Systems engineering
- Transparency
- Requirements specification
- Test specification
- Adversarial attacks

and Formal methods, Control theory, Probabilistic methods, Test case design, and Process guidelines as techniques and approaches to address these challenges. Figure 5.2 shows a proposed mapping between challenges and technologies.

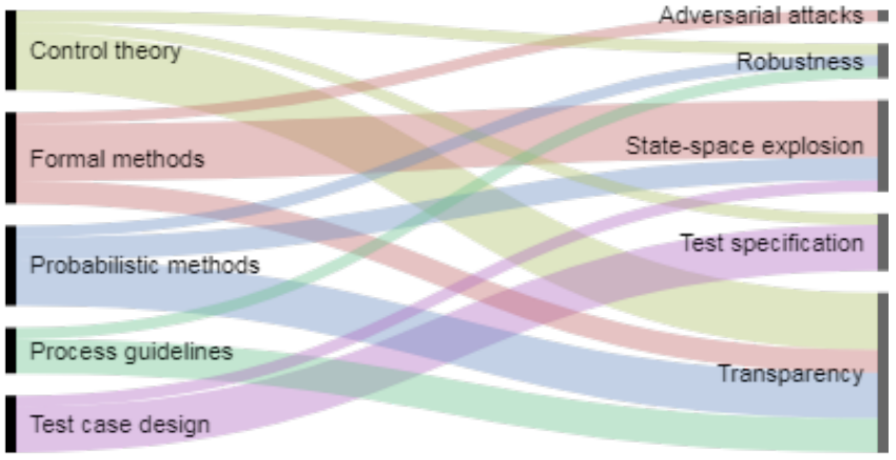

Figure 5.2: Mapping between categories of solution proposals and challenges (from [17])

Chapter 6

Data for Autonomous Systems

In traditional software systems, *data* most often shows up in the form of engineering constants, configuration data, and small lookup tables. On the other hand, numerous algorithms used for AUCs and modern software systems require, as discussed above, large numbers of data. For example, gigantic tables or representations of graphs are found, for example, in components that use neural networks of MDPs (Monte Carlo Decision Procedures). The ACAS X collision avoidance system [15] is a typical example. These data are part of the on-board software load and is executed while the system is in operation. Most often, these data are not changed during operation.

The development of AUCs, which use machine-learning based algorithms rely on often huge data sets during development time. Those are being used to estimate parameters (e.g., DNN weights) that will ultimately control the system behavior during the mission.

However, current practices and processes do not formally define a role for these data sets. In many cases, they are "kept around" somewhere. Due to the safety-criticality of the AUCs, which ultimately depend on these data, this comprises a major assurance gap. We propose to consider data as first-class citizens on the same level as software during development, V&V, and deployment.

From a process-perspective, the traditional "V" should be extended by a data acquisition and modeling phase plus a specific "machine learning testing" phase. Figure 6.1 illustrates the extended "V".

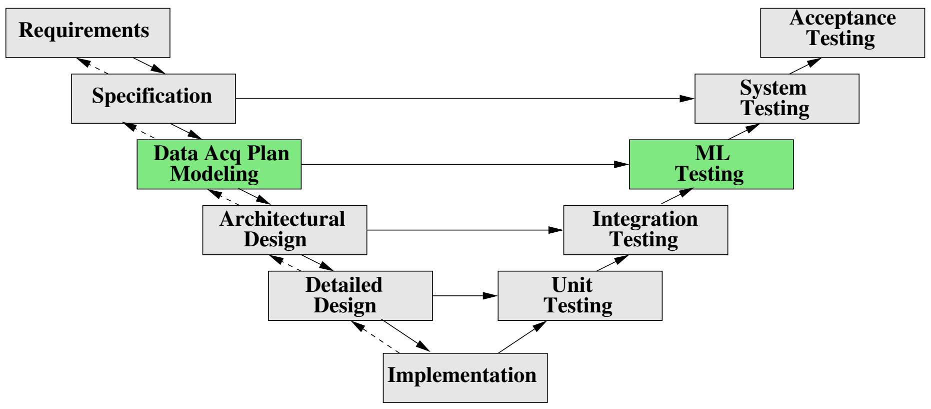

Figure 6.1: The "V" shape extended by tasks necessary to handle data driven and machine-learning algorithms

Necessary tasks (which can be currently considered gaps) include

- assurance of (training) data quality and coverage.
- assurance of the machine learning architecture, configuration, and learning algorithm. This typically concerns questions like: is the number of layers for the DNN justified and produces most reliable results?
- assurance of suitability of "data-driven software component". The question of: is a DNN the right or best algorithm for that task? Has the DNN be selected because it is "cool"?
- assurance of test coverage and quality

The data used to develop an AUC must therefore undergo similar activities for development and assurance like the software itself: documentation, version control, debugging, V&V, etc.

Chapter 7

Conclusions

In this report, we have looked at typical algorithms used in autonomous systems, grouped them and described some of their special characteristics. These characteristics, for example, dynamic memory, non-determinism, points to relevant gaps in techniques, processes, and practices of software assurance that will need to be addressed for V&V of autonomous systems. Figure 7.1 shows a high-level architecture and identifies such gaps.

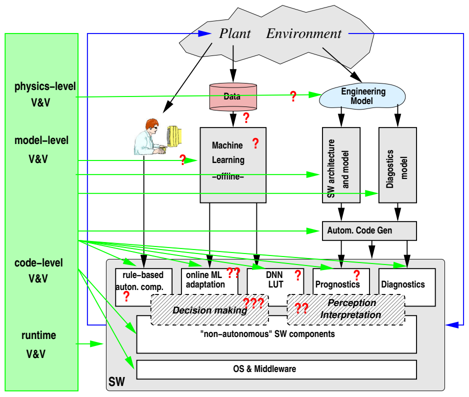

Figure 7.1: Overview and high-level architecture of an autonomous system with V&V activities (green) and gaps (red question marks: ?)

Whereas most of these gaps concern specific topics, (software) assurance for autonomous systems ultimately must address the following important issues:

- How to assure "ethical" behavior of an autonomous system? The typical example for this issue is the necessity of a self-driving car to make a real-time decision that causes the passenger to be killed or the pedestrian in a cross-walk. [18] discusses such problems.

- How to assure safety of complex autonomous functions? The ASTM publication F-3269 [19] is a first attempt toward a standard to safely bound the flight behavior of an autonomous vehicle.

- How to assure operations in a unknown or changing environment? Solving this issue is mandatory for advanced space missions, for example, missions to the Icy Moons. This issue also plays an important role in the deployment of self-driving cars: can we assure that a self-driving car, which was trained to drive in Mountain View, can safely handle the streets of San Francisco?

- How to assure safe/effective autonomy—human interaction and collaboration as needed for modern space missions (e.g., Artemis)?
- How can we assure that "emerging behavior" of an autonomous system can be detected, controlled, and, possibly, avoided? Uncontrolled emerging behavior might pose severe safety hazards.

Bibliography

1. Hügli, H.; Müller, J.-P.; Facchinetti, C.; and François: ARCHITECTURE OF AN AUTONOMOUS SYSTEM: APPLICATION TO MOBILE ROBOT NAVIGATION. 1994.

2. Lesire, C.; Infantes, G.; Gateau, T.; and Barbier, M.: A distributed architecture for supervision of autonomous multi-robot missions - Application to air-sea scenarios. *Auton. Robots*, vol. 40, no. 7, 2016, pp. 1343–1362. URL `https://doi.org/10.1007/s10514-016-9603-z`.

3. Dennis, L. A.; Fisher, M.; Aitken, J. M.; Veres, S. M.; Gao, Y.; Shaukat, A.; and Burroughes, G.: Reconfigurable Autonomy. *KI - Künstliche Intelligenz*, vol. 28, no. 3, Aug 2014, pp. 199–207. URL `https://doi.org/10.1007/s13218-014-0308-1`.

4. FAA: Introduction to TCAS II. 2011. URL `https://www.faa.gov/documentlibrary/media/advisory_circular/tcas%20ii%20v7.1%20intro%20booklet.pdf`.

5. Leveson, N. G.: *Safeware: System Safety and Computers*. ACM, New York, NY, USA, 1995.

6. Verma, V.; Jonsson, A.; Pasareanu, C.; and Iatauro, M.: Universal Executive and PLEXIL: Engine and Language for Robust Spacecraft Control and Operations. *Spacecraft Control and Operations, American Institute of Aeronautics and Astronautics Space 2006 Conference*, 2006.

7. Rocha, C.; Cadavid, H.; Muñoz, C.; and Siminiceanu, R.: A Formal Interactive Verification Environment for the Plan Execution Interchange Language. *Integrated Formal Methods*, J. Derrick, S. Gnesi, D. Latella, and H. Treharne, eds., Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 343–357.

8. Schumann, J.; Mbaya, T.; Mengshoel, O.; Pipatsrisawat, K.; Srivastava, A.; Choi, A.; and Darwiche, A.: Software Health Management with Bayesian Networks. *Innov. Syst. Softw. Eng.*, vol. 9, no. 4, Dec. 2013, pp. 271–292. URL `http://dx.doi.org/10.1007/s11334-013-0214-y`.

9. Darwiche, A.: *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009. URL `http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521884389`.

10. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Goodfellow, I.; Harp, A.; Irving, G.; Isard, M.; Jia, Y.; Jozefowicz, R.; Kaiser, L.; Kudlur, M.; Levenberg, J.; Mané, D.;

Monga, R.; Moore, S.; Murray, D.; Olah, C.; Schuster, M.; Shlens, J.; Steiner, B.; Sutskever, I.; Talwar, K.; Tucker, P.; Vanhoucke, V.; Vasudevan, V.; Viégas, F.; Vinyals, O.; Warden, P.; Wattenberg, M.; Wicke, M.; Yu, Y.; and Zheng, X.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. URL `https://www.tensorflow.org/`, software available from tensorflow.org.

11. RTCA: DO-178C/ED-12C: Software Considerations in Airborne Systems and Equipment Certification. 2012. URL `http://www.rtca.org`.

12. Schumann, J.; and Schneider, S.: Automated Testcase Generation for Numerical Support Functions in Embedded Systems. *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*, J. M. Badger and K. Y. Rozier, eds., Springer, vol. 8430 of *Lecture Notes in Computer Science*, 2014, pp. 252–257. URL `https://doi.org/10.1007/978-3-319-06200-6\_20`.

13. Gill, P. E.; Murray, W.; and Wright, M. H.: *Practical optimization*. Academic Press Inc. [Harcourt Brace Jovanovich Publishers], London, 1981.

14. Jeannin, J.-B.; Ghorbal, K.; Kouskoulas, Y.; Gardner, R.; Schmidt, A.; Zawadzki, E.; and Platzer, A.: Formal Verification of ACAS X, an Industrial Airborne Collision Avoidance System. *Proceedings of the 12th International Conference on Embedded Software*, EMSOFT '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 127–136. URL `http://dl.acm.org/citation.cfm?id=2830865.2830880`.

15. Giannakopoulou, D.; Guck, D.; and Schumann, J.: Exploring Model Quality for ACAS X. *FM*, 2016.

16. Schumann, J.; and Liu, Y., eds.: *Applications of Neural Networks in High Assurance Systems*, vol. 268 of *Studies in Computational Intelligence*. Springer, 2010. URL `https://doi.org/10.1007/978-3-642-10690-3`.

17. Borg, M.; Englund, C.; Wnuk, K.; Durán, B.; Levandowski, C.; Gao, S.; Tan, Y.; Kaijser, H.; Lönn, H.; and Törnqvist, J.: Safely Entering the Deep: A Review of Verification and Validation for Machine Learning and a Challenge Elicitation in the Automotive Industry. *CoRR*, vol. abs/1812.05389, 2018. URL `http://arxiv.org/abs/1812.05389`.

18. McBride, N.: The Ethics of Driverless Cars. *SIGCAS Comput. Soc.*, vol. 45, no. 3, Jan. 2016, pp. 179–184. URL `http://doi.acm.org/10.1145/2874239.2874265`.

19. ASTM F3269 - 17 Standard Practice for Methods to Safely Bound Flight Behavior of Unmanned Aircraft Systems Containing Complex Functions. 2017.

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704–0188*

1. REPORT DATE *(DD-MM-YYYY)* 01-01-2020	**2. REPORT TYPE** Contractor Report	**3. DATES COVERED** *(From - To)* 01/2019–09/2019

4. TITLE AND SUBTITLE MBSwE for Autonomous Systems with Reuse: Software Assurance Best Practices and Gaps Technical Report on Gap Analysis	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER SARP
6. AUTHOR(S) Johann Schumann, Yuning He	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Ames Research Center, Moffett Field, CA 94035	**8. PERFORMING ORGANIZATION REPORT NUMBER** L–
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSOR/MONITOR'S ACRONYM(S)** NASA
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) NASA/CR–2020–5000437

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 61
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://ntrs.nasa.gov.

14. ABSTRACT
Examples of gaps include difficulties to specify safety requirements, coverage of training and test sets for deep learning, and the missing explainability of decisions made by autonomous AI systems.

15. SUBJECT TERMS
software engineering, model-based software development, code generation

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Information Desk (email: `help@sti.nasa.gov`)
U	U	U	UU		**19b. TELEPHONE NUMBER** *(Include area code)* (757) 864-9658