Model-Based Design for CPS with Learning-Enabled Components

Charles Hartsell
Institute for Software Integrated Sys.
Vanderbilt University
charles.a.hartsell@vanderbilt.edu

Nagabhushan Mahadevan
Institute for Software Integrated Sys.
Vanderbilt University
nag.mahadevan@vanderbilt.edu

Shreyas Ramakrishna
Institute for Software Integrated Sys.
Vanderbilt University
shreyas.ramakrishna@vanderbilt.edu

Abhishek Dubey
Institute for Software Integrated Sys.
Vanderbilt University
abhishek.dubey@vanderbilt.edu

Theodore Bapty
Institute for Software Integrated Sys.
Vanderbilt University
theodore.a.bapty@vanderbilt.edu

Taylor Johnson
Institute for Software Integrated Sys.
Vanderbilt University
taylor.johnson@vanderbilt.edu

Xenofon Koutsoukos
Institute for Software Integrated Sys.
Vanderbilt University
xenofon.koutsoukos@vanderbilt.edu

Janos Sztipanovits
Institute for Software Integrated Sys.
Vanderbilt University
janos.sztipanovits@vanderbilt.edu

Gabor Karsai
Institute for Software Integrated Sys.
Vanderbilt University
gabor.karsai@vanderbilt.edu

ABSTRACT

Recent advances in machine learning led to the appearance of Learning-Enabled Components (LECs) in Cyber-Physical Systems. LECs are being evaluated and used for various, complex functions including perception and control. However, very little tool support is available for design automation in such systems. This paper introduces an integrated toolchain that supports the architectural modeling of CPS with LECs, but also has extensive support for the engineering and integration of LECs, including support for training data collection, LEC training, LEC evaluation and verification, and system software deployment. Additionally, the toolsuite supports the modeling and analysis of safety cases – a critical part of the engineering process for mission and safety critical systems.

CCS CONCEPTS

• **Software and its engineering → Application specific development environments**;

KEYWORDS

cyber physical systems, machine learning, model based design

ACM Reference format:
Charles Hartsell, Nagabhushan Mahadevan, Shreyas Ramakrishna, Abhishek Dubey, Theodore Bapty, Taylor Johnson, Xenofon Koutsoukos, Janos Sztipanovits, and Gabor Karsai. 2019. Model-Based Design for CPS with Learning-Enabled Components. In *Proceedings of DESTION '19: Design Automation for CPS and IoT, Montreal, QC, Canada, April 15, 2019 (DESTION '19),* 9 pages.
https://doi.org/10.1145/3313151.3313166

ACRONYMS

ALC	Assurance-based Learning-enabled CPS
CPS	Cyber Physical System
CNN	Convolutional Neural Network
DSML	Domain Specific Modeling Language
GSN	Goal Structuring Notation
LEC	Learning Enabled Component
UUV	Unmanned Underwater Vehicle
ROS	Robot Operating System

1 INTRODUCTION

Cyber Physical Systems (CPSs) are often required to operate in highly uncertain environments, with significant degree of autonomy. For such systems, it is typically infeasible to explicitly design for all possible situations within the environment. CPS designers are increasingly using data-driven methods such as machine learning to overcome this limitation. Learning Enabled Components (LECs) have demonstrated good performance for a variety of traditionally difficult tasks such as object detection and tracking [18], robot path planning in urban environments [37], and attack detection in smart power grids [29]. Some systems have even used end-to-end learning components for complex CPS tasks, such as the NVIDIA DAVE-2 [4] which used a Convolutional Neural Network (CNN) to map images from a front-facing camera directly to steering commands for an autonomous vehicle. However, component-based design where only selected functionality is implemented with LECs remains the more widely used approach.

Development of CPSs requires strong coordination between multiple engineering disciplines, challenging the traditional principle of "separation of concerns". Each discipline relies on their own specialized modeling languages and methods, typically supported by numerous software tools (e.g. simulators, analysis tools, CAD software, etc) with little to no support for the methods of other disciplines. Previous work with the OpenMETA tool suite [39]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DESTION '19, April 15, 2019, Montreal, QC, Canada
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6699-1/19/04…$15.00
https://doi.org/10.1145/3313151.3313166

addressed this problem with the introduction of the CyPhyML model-integration language and supporting model and tool integration platforms. However, the platform focused on CPSs using conventional components based on analytical understanding of the domain. Such techniques do not account for epistemic uncertainties in the models of the physical components as well as their environments. Data-driven techniques, including machine learning methods, are a promising approach to account for these limitations. In this work, we consider how the concepts developed for the OpenMETA platform can be extended to learning-enabled systems and their assurance.

CPSs are commonly used in mission-critical or safety-critical applications which demand high reliability and strong assurance for safety. Assuring safety in these systems requires supporting evidence from testing data, formal verification, expert analysis, etc. Machine learning relies on inferring relationships from data instead of deriving them from analytical models, leading many systems employing LECs to rely almost entirely on testing results as the primary source of evidence. However, test data alone is generally insufficient for assurance of safety-critical systems. Techniques for formal verification of learning-enabled systems are an active area of research [35][42] and will need to be incorporated into the safety assurance of such systems.

Additionally, evidence used for safety assurance should be traceable and reproducible. Manual data management across the complex toolsuites often used for CPS development is a time consuming and error-prone process. This issue is even more pronounced for systems using LECs where training data and the resulting trained models must also be properly managed. Clearly, the task of maintaining traceability and reproducibility in all phases of the development cycle should be automatically handled by an appropriate development environment. In this paper, we introduce a model-driven design methodology for Assurance-based Learning-enabled CPS (ALC) and present the supporting development environment called the ALC Toolchain. Our approach combines multiple Domain Specific Modeling Languages (DSMLs) to support various tasks including architectural modeling, experiment configuration/data generation, LEC training, performance evaluation, and system safety assurance. All generated artifacts - including system models, simulation data, trained networks, etc. - are stored and managed to allow for both traceability and reproducibility. Methods are provided for constructing static, design-time safety assurance arguments as well as dynamic, run-time assurance monitors. Certain artifacts, such as formal verification results and testing evaluation metrics, may be referenced as evidence in static system assurance arguments.

The rest of this paper is organized as follows. First, Section 2 discusses various related research efforts. Section 3 explains our methodology and the various modeling languages used to support it, followed by a description of the tool chain implementation in Section 4. Next, Section 5 provides illustrative examples of the methodology applied to multiple CPS platforms. Finally, Sections 6 and 7 discuss possible directions for future research and concluding remarks respectively.

2 RELATED RESEARCH

Various existing architectural description languages provide integration with analysis tools, such as the Architecture Analysis and Design Language (AADL) [12]. AADL is a standard for modeling and analyzing real-time embedded systems. Components in AADL may represent hardware, software, or system entities, and connections between components are used to model component interactions. AADL allows for analysis of various system properties including performance, schedulability and reliability. OMG's SysML [28] provides a general-purpose modeling language for systems engineering. SysML can be extended with the Modeling and Analysis of Real-time and Embedded systems (MARTE) profile for UML [16] to allow for system analysis similar to AADL. These languages and tools offer strong support for general-purpose system development, but are not well equipped for many domain-specific tasks. In particular, they do not consider how data-driven techniques such as machine-learning may be integrated into the development of these systems.

DeepForge [6] is a machine learning development environment which aims to reduce the barriers to entry and development times for deep learning models. It provides a DSML with modeling concepts for describing neural network architectures and their training pipelines. Additionally, DeepForge uses a version control system to ensure traceability and reproducibility during the design of a deep learning model. The Google Colaboratory [1] is another interactive environment for machine-learning research, but provides a free-form environment to the user instead of following any predefined methodology. Similarly, OpenAI Gym [5] provides another machine-learning toolset focused on reinforcement-learning techniques. However, these environments only consider the development of machine learning models, and do not address how these models may be incorporated into a larger development framework for CPS. In particular, they do not integrate LEC assurance and verification techniques which may be essential for systems used in safety-critical applications.

While machine learning offers several significant advantages over conventional design techniques, it also poses some unique challenges. In [34], the authors consider the hidden costs of machine learning with the idea of *technical debt*. They identify several ways in which machine learning can incur significant costs for long-term maintenance of a system, and propose development techniques to mitigate these costs. Many of these techniques (eg. using appropriate levels of abstraction and careful management of data dependencies between components) can be enforced with an appropriate methodology and should be automated supporting tools.

Recent work has shown that many machine learning techniques are susceptible to adversarial examples where small input perturbations can cause the model to produce incorrect outputs with high confidence [14]. This is particularly troublesome for LECs used in safety-critical applications. The authors of [32] address this issue by introducing DeepXplore - a whitebox testing framework for deep learning models. DeepXplore provides an algorithm for generating test inputs which exercise the corner cases of a particular learning model. Also, the authors introduce the concept of "neuron coverage", analogous to branch coverage in traditional software testing, for

[1] colab.research.google.com

systematically measuring the amount of a learning model exercised by a given test set. This approach significantly improves testing of machine-learning models, providing more confidence that the system will perform as expected. However, testing alone is not sufficient for safety-critical systems and will need to be supplemented with other safety assurance techniques.

The Model-based Demonstrator for Smart and Safe Cyber Physical Systems (MoDeS3) [40] combined several model-based design languages and techniques for the development of a model railway system. The authors specified the system architecture with SysML block diagrams and the component level behavior with the Gamma Statechart Composition Framework [24]. Code generation tools provided by the Gamma framework were used for implementation of the software components. Both design-time and run-time safety assurance techniques were applied including formal verification methods and run-time monitors. The authors successfully demonstrate the effectiveness of Model Based Systems Engineering (MBSE) for CPS. However, MoDeS3 does not consider the integration of machine learning with CPS. Additionally, MoDeS3 does not provide a unifying tool-chain for the MBSE methodology used.

Other projects have developed DSMLs specifically for machine learning applications in CPS. The authors of [17] advocate for a fine-grained learning approach which operates on small regions of a larger data set, as opposed to typical coarse-grained approaches which learn global behavior patterns from the complete data set. Their approach combines system properties learned from this fine-grained approach with properties derived from domain knowledge of the system, and the authors present a DSML derived from UML class diagrams for modeling this combination.

3 METHODOLOGY

Our methodology combines multiple DSMLs to support common tasks for CPS development including architectural modeling, experiment configuration/data generation, performance evaluation, and system safety assurance. Special considerations are made for CPS which use LECs with LEC training models for both supervised and reinforcement learning methods. The following sections describe each DSML and how they fit into the development workflow.

3.1 System Architecture Model

The CPS development workflow begins with high-level specification of the system architecture, and our approach supports a subset of the block diagram models from the SysML modeling standard for system architecture design. *Components* are abstract building blocks of the system architecture, and may be hierarchically composed into complete systems. Components are first defined in a *block library* before instances of a component can be created in a *system model.* This approach promotes reusability and maintainability of components across many system models. Component interfaces are defined using *ports* which can represent signal, power, or material flows. Software components use directional signal ports to model data flow between components, and each signal port produces or consumes a particular *message type.* Message types are similar to C-style data structures, and must be defined in a *message library.* Physical components use acausal power and/or material ports.

Components may contain one or more concrete *implementation alternatives* which fulfill the component's functional requirements. For example, a software object detection component may include one implementation based on conventional (analytical) methods and another based on machine-learning methods. For software components, implementation models contain the information necessary to load, configure, and initialize the software 'nodes' when the system is deployed as well as the business-logic associated with its runtime implementation.

3.2 Development Workflow Model

Once the system architecture is established, the developer executes one of two LEC development workflows – supervised or reinforcement learning - shown in Figure 1. Supervised learning begins with the collection of training data using one or more *experiment* or *campaign* models. In a *supervised learning* setup, an LEC model is trained against the collected data set, with the option to train run-time assurance monitors as well. Then, new *experiment* or *campaign* models may deploy the system with the trained LEC for integrated system testing and evaluation.

In a *reinforcement learning* setup, the LEC model is trained while the system or environment is being executed (or simulated). During training, the LEC model is updated based on the environment response (state and reward) to the input action. The training is repeated for a specified number of episodes, with each episode lasting a specified maximum time-limit or step-size. The trained LEC is evaluated by executing the reinforcement learning setup in a non-training mode.

3.3 Experiment Model

The experiment model captures all information necessary for configuration and execution of a system. First, the architecture model of the system under test is refined to an *assembly model* where one specific implementation from the available alternatives is selected. This is done for each component that includes multiple implementation alternatives. Execution of the assembly model requires additional parameters divided into three sets: environment, mission, and execution. *Environment parameters* define any environmental variables (terrain, weather conditions, etc.) and provide the files necessary for launching the simulation. *Mission parameters* define the objective for the system (eg. Track and follow a pipeline on the seafloor) and the relevant parameters to set up the experiment. *Execution parameters* define miscellaneous parameters for simulator management, data server configuration, etc.

An experiment model can be enriched with a *Campaign* model which configures iterations of a experiment. Campaigns include a *parameter sweep* block for varying system or environment parameters over a range of possible values. Campaigns are commonly used to tune system parameters for optimal performance, or to gather data in a variety of environments for LEC training.

3.4 Training LECs

Integration of model-based design with learning-enabled components is a focus of our methodology. Therefore, LEC training is an essential part of this methodology, and models for supervised [25] and reinforcement [38] learning are provided. Both techniques

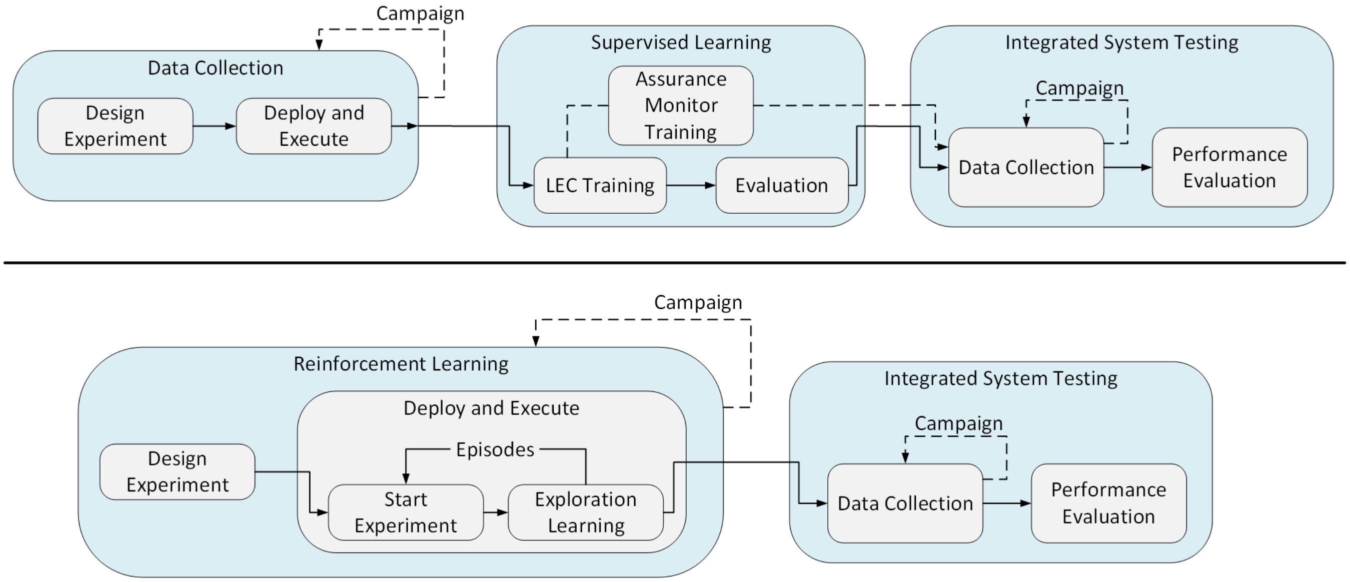


Figure 1: LEC Development workflows for supervised learning (Top) and reinforcement learning (Bottom).

utilize the *LEC model* concept, which allows for learning architectures to be defined graphically or through code. Currently, the LEC model supports specification of deep neural networks.

The model for *supervised learning* setup includes the LEC model to be trained, the training data set(s), the training code, and any necessary hyper-parameters. The *training data* provides a reference to the desired data sets, which are typically results from previous experiments or campaigns. The *parameter* block captures hyper-parameters relevant to the specific training exercise (e.g. batch size, number of epochs, etc.).

Unlike supervised learning, reinforcement learning involves training the LEC online (i.e. interacting with the environment). Just as in the experiment model setup, the *reinforcement learning* model includes an *assembly model* where the specific component implementation(s) are selected from the possible alternates. The setup definition includes the LEC model to be trained, underlying reinforcement learning algorithm, associated rewards function for agent action given the current state of the system, and any training hyper-parameters.

3.5 Safety Assurance

Safety assurance is a critical component of any CPS which operates in mission-critical or safety-critical applications, but certification requirements and techniques vary between different regulating bodies. Safety cases are one such method of system certification which have been accepted by certain industries for years (eg. UK Ministry of Defense [26]). Safety cases are essentially structured arguments, often represented graphically, with supporting evidence that a particular system is acceptably safe. They have gained popularity in areas including CPS software development, and more regulating bodies have published guidelines and standards for their use (eg. Appendix D of FAA Unmanned Aircraft Systems Operational Approval document [2]).

Our methodology uses Goal Structuring Notation (GSN) [19] to allow for construction of safety cases. GSN is a language for representing structured arguments in a tree-like graphical form where primary *goals* are decomposed into logical combinations of *subgoals*, often in a hierarchical manner. Goals at the lowest level are represented by leaf nodes in the tree and are supported by *solution* nodes which provide evidence that the goal is satisfied. *Strategy* nodes can be used when composing multiple subgoals into a single, higher-level goal to provide a detailed explanation for the reasoning of the argument. Any assumptions made during the construction of a safety case can be explicitly stated with *Assumption* blocks, which may be attached to any node in an argument. A more comprehensive introduction to hierarchical safety cases and the GSN standard can be found in [10].

As part of our development of the Systems Engineering And Assurance Modeling (SEAM[2]) toolsuite, we extended the GSN models and integrated them with system architecture models to provide context to each branch of the assurance case argument. GSN models - integrated with system architecture and fault propagation models - were used to build assurance cases for radiation-reliability of CubeSat payloads with commercial off the shelf parts[3]. The assurance arguments were grounded in Reliability and Maintainability standards (NASA-STD-8729.1 [3]) established by NASA's Office of Space and Mission Assurance (OSMA) for space flight systems[15].

The ALC toolchain builds upon the extensions to GSN from SEAM. GSN goals and solutions often address one particular component or subsystem. *Model reference* blocks provide a reference from a GSN node to a block in the system architecture model to make this relationship explicit. Similarly, solution nodes can provide a link to any supporting data - formal verification results, testing evaluation metrics, etc. - using *evidence source* blocks.

[2] https://modelbasedassurance.org/
[3] https://standards.nasa.gov/standard/nasa/nasa-std-87291

3.6 Verification and Run-Time Assurance

Currently, assurance of LEC-based systems is heavily reliant on testing results and such systems often require constant supervision from a human operator for acceptance [27]. To supplement testing-based assurance, our methodology also supports new formal verification techniques [43] as well as dynamic assurance monitors [31] and provides modelling concepts for both. However, there is a fundamental problem with LECs: the training set is finite and it may not capture all possible situations the system encounters at operation time. For such unknown situations the LEC may produce incorrect or unacceptable results – and the rest of the system may not even know that. Hence, the safety assurance of CPS with LECs is very problematic. One concept that might help to mitigate this situation is to use continuous monitoring on the LEC to its performance and indicate when the level of confidence in the output of the LEC is low – i.e. if the LEC does not perform as expected. This monitoring process is termed 'assurance monitoring', which oversees the LEC and gives a clear indication of problematic situations. These assurance monitor techniques are an active area of research, and a more detailed explanation can be found in [30]. Once a problematic indication is given, a higher-level control loop – or a 'safety-controller' – may take over and perform a safe action (e.g. slow down the vehicle) to mitigate the lack of performance in the LEC. Note that the 'safety controller' must be independently designed and verified such that an overall safety assurance case can be constructed for the complete system, and executed in a way that allows hot-swapping.

4 IMPLEMENTATION

The physical architecture of our toolchain is shown in Figure 2, and is centered around the WebGME infrastructure [23]: a meta-programmable collaborative modeling environment which provides several advantages. First, the WebGME user interface is a web-based environment that can be accessed from most web browsers and allows for real-time collaboration between multiple users. WebGME supports a flexible meta-modeling paradigm that allows for development of customized DSMLs. Further, WebGME API provides support to write custom code for model visualization and interpretation. While WebGME allows for code to be executed within the browser, it is not intended for execution of computationally intensive tasks such as simulation or LEC training. Instead, these jobs are dispatched to execution servers equipped with appropriate hardware. All generated data resulting from execution of a job is uploaded to a central fileserver, with only the relevant metadata being returned to WebGME and stored in a version-controlled database. Each aspect of this architecture is discussed in more detail in the following sections.

4.1 Modeling Language

In WebGME, the meta-model for a DSML can be created from scratch or it can be built on top of a library of existing DSMLs. The ALC DSML borrows upon existing meta-model libraries: SEAM [2], DeepForge [4] and ROSMOD [5]. Our toolchain provides an integrated modeling framework that supports multiple models including:

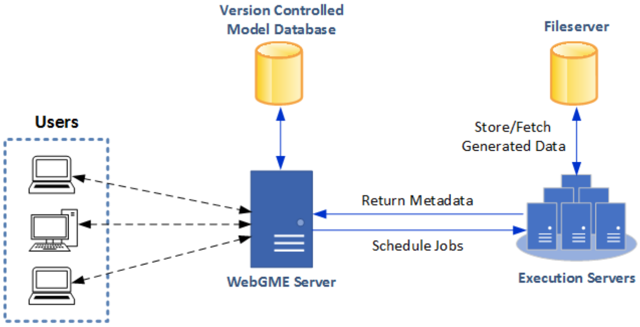


Figure 2: Toolchain architecture overview.

- A system architecture model based on SysML Internal Block Diagrams allows the user to describe the system architecture in terms of the underlying components (hierarchical blocks) and their interaction via signal, energy, and material flows.
- An experiment configuration model allows the users to configure execution instances for data collection, training of LECs with supervised or reinforcement learning, deployment and evaluation of trained LECs.
- Assurance case models based on GSN allows the user to create assurance arguments for the safety, performance, and reliability of the system.

The ALC toolchain uses the WebGME visualization and decorator framework to customize the visualization based on the model context. It borrows upon the WebGME CodeEditor to allow the users to develop and edit their code within the context of an ALC model.

4.2 Execution

The ALC toolchain uses the WebGME plugin and executor framework to launch execution instances on appropriate server machines (labeled "Execution Servers" in Figure 2). These execution instances could be a system execution (or simulation) for data collection or training exercises of LECs. Such simulation and training exercises are often computationally intense, and usually require Graphics Processing Units (GPU) or other forms of hardware acceleration. The ALC toolchain extends upon the DeepForge Pipeline model and its execution to allow remote deployment of computationally intense tasks on appropriately equipped servers. This enables developers of CPS to configure and launch computationally intensive simulation and training exercises on powerful machines from local web browsers, while collaborating with a distributed team of developers.

4.2.1 Interactive Execution and Debugging. The toolchain supports embedded Jupyter notebooks within the context of an experiment, training, or evaluation model. The toolchain can configure the code in the Jupyter notebook to execute the model. This allows users to launch their execution instances in an interactive manner and debug their code if required. Additionally, it allows users to write custom code to evaluate the system performance.

[4] http://deepforge.org/
[5] https://cps-vo.org/group/ROSMOD

4.3 Data Management

Simulation environments often generate large amounts of data, which are needed for effective LEC training. In the ALC toolchain, large data sets (eg. simulation data and trained LEC models) are stored in a dedicated file server as shown in Figure 2. Currently, the implementation uses SSH File Transfer Protocol (SFTP) [13] with the standard ext4 Linux filesystem. Once a data set is uploaded, a corresponding metadata file is returned to the WebGME server and stored in the model. The metadata files provide enough information for retrieving a particular data set from the file-sever when needed for other tasks such as LEC training, performance evaluation, or LEC deployment.

When the experiment results are uploaded to the file-server, configuration files and other artifacts used to execute the experiments are stored with the generated data. This allows for the experiment to be repeated and for any generated data to be reproduced as needed. Additionally, this pattern of uploading the data to a dedicated server and only storing the corresponding meta-data in the model frees WebGME from handling large files and improves efficiency as well as model-scalability.

4.4 Traceability

Since we consider safety-critical systems, traceability and reproducibility at every step in the development process is a primary focus of our toolchain. WebGME provides a version control scheme similar to Git [6] where model updates are stored in a tree structure and assigned an SHA1 hash. For each update, only the differences between the current state and the previous state of the model are stored in the tree. This allows for the model to be reverted to any previous state in the history by rolling back changes until the hash corresponding to the desired state is reached. For a more detailed explanation of this approach, see [23].

4.5 Experiment Gyms

The ALC toolchain is intended for both simulated and real-world experiment environments, referred to as "gyms". Currently, we support three gyms using open-source simulation environments: UUV Simulator, CARLA, and TORCS. UUV Simulator [7] [22] is an extension to the Gazebo [8] [21] simulation environment which provides additional plugins for simulation of Unmanned Underwater Vehicles (UUVs). CARLA [9] [11] is an automotive simulator intended for the development of autonomous vehicles. The Open Racing Car Simulator (TORCS) [10] [41] is another automotive simulator which has been used in several research projects. Currently, all software components are implemented using the Robot Operating System (ROS) [11] [33] middleware.

Training and execution of LEC models is done using the Keras neural network library [7] on top of the TensorFlow machine learning framework [1]. Additionally, Jupyter notebooks [20] have been integrated into the WebGME environment which allows users to interactively perform simulations, training, or performance evaluation.

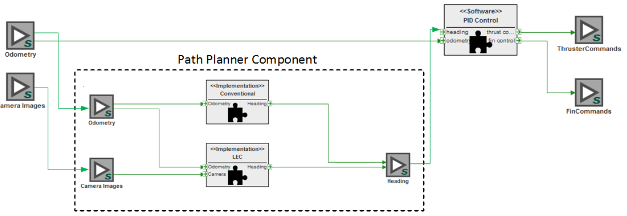


Figure 3: System architecture diagram of the UUV controller.

5 EXAMPLES

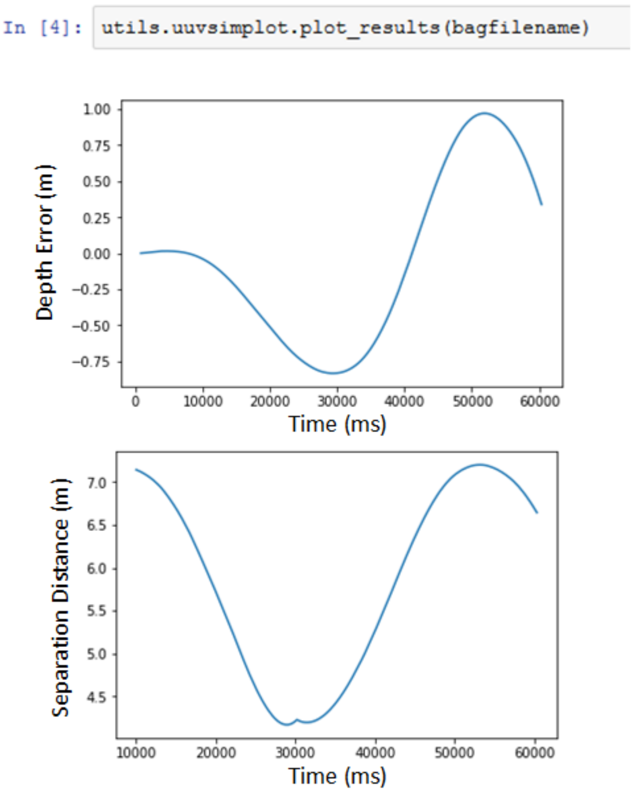
5.1 Unmanned Underwater Vehicle

5.1.1 System Overview. As an example, we consider the design and implementation of a UUV controller tasked with following a pipeline on the seafloor. The controller is built on the ROS middleware, and experiments were performed using the Gazebo simulation environment with the open-source UUV Simulator extension packages. The ECA A9 vehicle provided with UUV Simulator was the chosen UUV. This vehicle is equipped with four control fins, one propeller/thruster, a forward-looking camera, and vehicle speed and position sensors. Additional sensors are available but were not used for this example.

The controller was required to produce all actuator commands necessary to follow the pipeline at a desired separation distance using the image stream from the camera and the vehicle odometry data as input. This task was further divided into two components: a path planner and a lower-level PID controller. The path planner component was responsible for determining a suitable heading for the vehicle to follow based on images from the camera. This heading is sent to the PID controller which then must produce commands for all four fins and the thruster. The desired heading was provided as a 3 dimensional vector with components for both pitch and yaw control of the vehicle. However, the PID controller used fixed setpoints for both depth and speed control of the vehicle, and the pitch component of the desired heading was discarded.

A block library of components was created to model vehicle sensors, actuators, and the controller. The controller block was composed from two component blocks representing the path planner and the PID controller. Each component was assigned an implementation with the ROS launch files necessary for configuring and deploying the component. The path planner component contains two implementations: an analytical planning algorithm and a Neural Network based solution. The analytical planner had access to ground truth information about the pipeline from the Gazebo simulator, and was used to generate training data for the LEC based implementation. Instances of the library blocks were then connected into a system model, and the UUV controller section of this model is shown in Figure 3. This figure shows an exploded view of the path planner component with both conventional and LEC implementation blocks.

[6] http://git-scm.com
[7] uuvsimulator.github.io
[8] www.gazebosim.org
[9] www.carla.org
[10] torcs.sourceforge.net
[11] www.ros.org

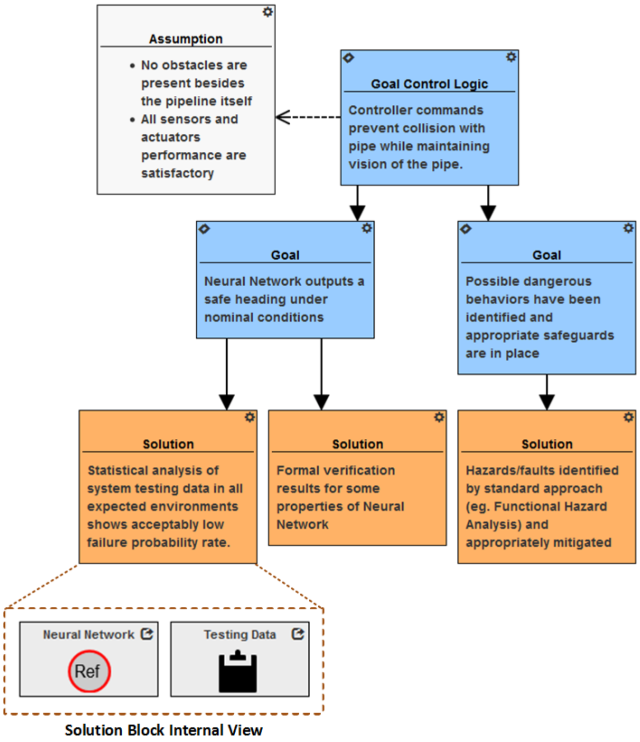


```
In [4]: utils.uuvsimplot.plot_results(bagfilename)
```

Figure 4: Interactive plotting and evaluation of system performance metrics through Jupyter notebook. Depth error (upper) and separation distance (lower) plots are shown.

5.1.2 Data Generation and Training. An experiment was configured using the UUV system model with the analytical implementation selected for the path planning component. For the experiments, the environment was set to a world model with a flat seabed and good water visibility. The experiment mission was for the UUV to follow a pipe on the seabed. In order to generate sufficient training data, campaign models were setup to run multiple iterations of the experiment, with varying pipe layouts. The qualities of the pipe (e.g. size or color) were not changed. Additionally, as part of the data generation campaign, each experiment was executed twice: first without disturbing the path of the vehicle, then with random noise added. This was done to gather additional training data for situations where the vehicle is in a non-optimal location relative to the pipe.

A supervised learning model was constructed for the LEC implementation of the path planner, and the data generated from the campaign was referenced for training data. Initially, a simple CNN architecture was chosen and trained. A second campaign model was created using the LEC based path planner for evaluation of the trained CNN. Multiple CNN architectures were trained and evaluated in an iterative process before finally selecting a modified version of the NVIDIA DAVE-2 [4] model. Additionally, training hyper parameters such as batch size and number of epochs were adjusted for optimal performance. Each CNN was evaluated based on several metrics including: error between the CNN predicted heading and the ideal heading from ground-truth data, how well the system maintained the desired separation distance from the pipe, and if the system successfully kept the pipe in view of the camera at all times. Jupyter notebook integration allows for interactive plotting and evaluation of system performance metrics using either provided utility functions, as shown in Figure 4, or custom python functions written by the user. The upper plot in Figure 4 shows the error between the desired depth and the actual depth of the UUV against time, while the lower plot shows the separation distance between the UUV and the pipeline during the same time range.



- No obstacles are present besides the pipeline itself
- All sensors and actuators performance are satisfactory

Controller commands prevent collision with pipe while maintaining vision of the pipe.

Neural Network outputs a safe heading under nominal conditions

Possible dangerous behaviors have been identified and appropriate safeguards are in place

Statistical analysis of system testing data in all expected environments shows acceptably low failure probability rate.

Formal verification results for some properties of Neural Network

Hazards/faults identified by standard approach (eg. Functional Hazard Analysis) and appropriately mitigated

Figure 5: Simplified safety case for UUV system with internal view of solution block shown.

5.1.3 System Assurance. The primary safety goal for our UUV is to avoid collision with the submerged pipeline at all times. Additionally, the system should also keep the pipe in view of the camera while progressing at a set minimum speed. With this in mind, a GSN safety case was developed for the complete system which consisted of approximately 100 total blocks constructed hierarchically. However, due to space constraints, a simplified, single-level version of this argument is used as an example here and is shown in Figure 5. The primary goal of this argument has not changed, but only the software controller portion of the system is considered. Assumptions made in this argument are listed in the assumption block connected to the top-level goal. The top level goal is broken into two sub-goals: the neural network path planner outputs a safe heading, and possible dangerous behaviors have been identified and appropriately mitigated.

Solution blocks describe the evidence used to support each leaf goal, and may contain direct links to the evidence itself. For instance, the "Statistical analysis of system testing data ..." solution block shown in Figure 5 contains a reference to the neural network component in the system architecture model, as well as a reference to the simulated testing data used for statistical analysis. Both of these references are traversable, which allows the user to quickly navigate to the relevant model artifacts. Formal verification techniques for Neural Networks are an active area of research, and the ALC toolchain is currently limited to reachable set estimation methods [43]. While these methods are typically not sufficient for safety assurance of a complete system, they can provide guarantees for some system properties and should be used to supplement other assurance measures including system testing. Additionally, solution nodes may reference formal verification results as evidence, similar to the testing data reference block.

5.2 DeepNNCar

This example deals with an environment that includes the hardware in the loop. The testbed shown in Figure 6 includes a Traxxas Slash 2WD 1/10 Scale radio controlled car mounted with a Raspberry Pi (RPi) (onboard computing unit) and two sensors - a forward-facing RGB camera (30 FPS, resolution of 320x240) and an IR optocoupler which measures wheel RPM to compute vehicle speed. The overall goal is to drive the car autonomously around a track without violating any safety properties (e.g. do not cross track boundaries).

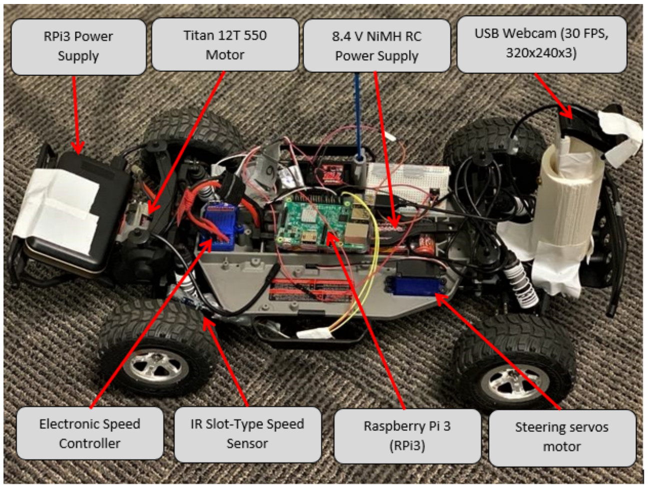

RPi3 Power Supply
Titan 12T 550 Motor
8.4 V NiMH RC Power Supply
USB Webcam (30 FPS, 320x240x3)
Electronic Speed Controller
IR Slot-Type Speed Sensor
Raspberry Pi 3 (RPi3)
Steering servos motor

Figure 6: DeepNNCar platform

5.2.1 System Architecture. The system architecture includes a LEC controller that uses a modified version of NVIDIA'S DAVE-II CNN model [4]. The control architecture includes a lane-detection based safety controller that serves as a back-up to keep the system within the tracks. These controllers are part of a Simplex Architecture (SA) [36] with a decision manager that decides the actual control command to the vehicle.

5.2.2 LEC Training. The LEC controller was trained and validated using the supervised learning setup with the data collected from the testbed. Thereafter, the trained LEC was deployed and its performance evaluated on multiple tracks. A reinforcement learning setup was employed to train and learn the arbitration logic in the decision manager to meet the performance and safety conditions. The training employed the Q learning algorithm to learn the mapping from the states (current speed, weights for each controller output) to actions (change in speed and change in controller weights). In this setup, the reward value was computed based on the distance from the center of the track and the current speed.

5.2.3 Assurance. An assurance case for the safety and performance of the system was modeled in GSN. It takes into account the performance of the individual components - the LEC controller, the safety controller, the lane detection algorithm, and the decision manager. The evidence to the GSN arguments is based on the performance of each individual component as well as the integrated system. Additionally, we are looking into how these individual evidences can be combined to predict the safety and performance of the overall system.

6 FUTURE WORK

Our toolchain is intended to support both simulated and real-world environments. Currently, the toolchain has been integrated with gyms that are simulation environments. We plan to integrate with gyms or testbeds that correspond to physical systems that involve hardware-in-the-loop such as DeepNNCar so that experiment configuration, execution, data collection, training and evaluation could be automated through the tool-chain.

Both formal verification and run-time assurance methods for LECs are active areas of research with new techniques being rapidly developed. These new techniques should be incorporated to our tool-chain as they become available, and may require new modeling concepts to more tightly integrate with the overall system design.

Various techniques for quantitative evaluation of confidence in safety case arguments have been developed (e.g. using Bayesian Networks [9] or Dempster–Shafer theory [8]) in an attempt to formalize certain aspects of the safety assurance process. Integrating these techniques with our methodology is another direction for future research.

7 CONCLUSION

Modern Cyber Physical Systems demand ever-increasing levels of autonomy while operating in highly uncertain environments. Conventional components are often insufficient for these systems due to the epistemic uncertainties present, leading many CPS developers to utilize Learning Enabled Components. These systems are often used in mission or safety critical applications where strong safety assurance is necessary. In this paper, we introduced a model-driven design methodology for Assurance-based Learning-enabled CPS which combines multiple DSMLs to support various tasks including architectural modeling, experiment configuration/data generation, system safety assurance, and LEC training, evaluation, and verification. A supporting development environment known as the ALC Toolchain allows for collaboration between multiple users while maintaining reproducibility and traceability during all stages of the development cycle. Additionally, the examples considered show how the complete methodology may be applied during the development of CPS using LECs.

ACKNOWLEDGEMENT

This work was supported by the DARPA and Air Force Research Laboratory. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or AFRL.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Federal Aviation Administration. Unmanned Aircraft Systems (UAS) Operational Approval. *online: https://www.faa.gov/documentLibrary/media/Notice/N_8900.227.pdf*, 2013.

[3] R. A. Austin, N. Mahadevan, A. F. Witulski, J. Evans, and A. F. Witulski. Radiation assurance of cubesat payloads using bayesian networks and fault models. In *2018 Annual Reliability and Maintainability Symposium (RAMS)*, pages 1–5, Jan 2018.

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[6] Brian Broll, Miklos Maroti, Peter Volgyesi, and Akos Ledeczi. DeepForge: A Scientific Gateway for Deep Learning. In *Gateways 2018*, 9 2018.

[7] François Chollet. Keras. https://keras.io/, 2015.

[8] Lukasz Cyra and Janusz Górski. Support for argument structures review and assessment. *Reliability Engineering System Safety*, 96(1):26 – 37, 2011. Special Issue on Safecomp 2008.

[9] E. Denney, G. Pai, and I. Habli. Towards measurement of confidence in safety cases. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 380–383, Sep. 2011.

[10] Ewen Denney, Ganesh Pai, and Iain Whiteside. Hierarchical safety cases. In *NASA Formal Methods Symposium*, pages 478–483. Springer, 2013.

[11] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[12] Peter H Feiler, David P Gluch, and John J Hudak. The architecture analysis & design language (AADL): An introduction. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006.

[13] J. Galbraith and O. Saarenmaa. SSH file transfer protocol, Jul 2006.

[14] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. *arXiv e-prints*, page arXiv:1412.6572, December 2014.

[15] Frank J Groen, John W Evans, and Anthony J Hall. A vision for spaceflight reliability: Nasa's objectives based strategy. In *Reliability and Maintainability Symposium (RAMS), 2015 Annual*, pages 1,6. IEEE, 2015-1.

[16] Object Management Group et al. Uml profile for marte: Modeling and analysis of real-time embedded systems, version 1.0. *On line: http://www.omg.org/spec/MARTE*, 2010.

[17] Thomas Hartmann, Assaad Moawad, Francois Fouquet, and Yves Le Traon. The next evolution of mde: a seamless integration of machine learning into domain modeling. *Software & Systems Modeling*, pages 1–20, 2017.

[18] David Held, Sebastian Thrun, and Silvio Savarese. Learning to track at 100 fps with deep regression networks. In *European Conference on Computer Vision*, pages 749–765. Springer, 2016.

[19] Tim Kelly and Rob Weaver. The goal structuring notation–a safety argument notation. In *Proceedings of the dependable systems and networks 2004 workshop on assurance cases*, page 6. Citeseer, 2004.

[20] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.

[21] Nathan P Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Citeseer, 2004.

[22] Musa Morena Marcusso Manhães, Sebastian A. Scherer, Martin Voss, Luiz Ricardo Douat, and Thomas Rauschenbach. UUV simulator: A gazebo-based package for underwater intervention and multi-robot simulation. In *OCEANS 2016 MTS/IEEE Monterey*. IEEE, sep 2016.

[23] Miklós Maróti, Tamás Kecskés, Róbert Kereskényi, Brian Broll, Péter Völgyesi, László Jurácz, Tihamer Levendovszky, and Ákos Lédeczi. Next generation (meta) modeling: Web-and cloud-based collaborative tool infrastructure. *MPM@ MoDELS*, 1237:41–60, 2014.

[24] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The gamma statechart composition framework. In *Internation Conference on Software Engineering*. ICSE, 2018.

[25] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[26] UK Ministry of Defense. Safety management requirements for defence systems, June 2007.

[27] U.S. Department of Transportation. Preparing for the future of transportation: Automated vehicles 3.0, Dec. 2018.

[28] OMG. OMG Systems Modeling Language (OMG SysML), Version 1.5, 2017.

[29] Mete Ozay, Inaki Esnaola, Fatos Tunay Yarman Vural, Sanjeev R Kulkarni, and H Vincent Poor. Machine learning methods for attack detection in the smart grid. *IEEE transactions on neural networks and learning systems*, 27(8):1773–1786, 2016.

[30] Harris Papadopoulos. Inductive conformal prediction: Theory and application to neural networks. In *Tools in artificial intelligence*. InTech, 2008.

[31] Harris Papadopoulos, Vladimir Vovk, and Alexander Gammerman. Regression conformal prediction with nearest neighbours. *Journal of Artificial Intelligence Research*, 40:815–840, 2011.

[32] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18. ACM, 2017.

[33] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[34] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *NIPS*, 2015.

[35] Sanjit A. Seshia and Dorsa Sadigh. Towards verified artificial intelligence. *CoRR*, abs/1606.08514, 2016.

[36] Lui Sha. Using simplicity to control complexity. *IEEE Software*, 4:20–28, 2001.

[37] S. M. Sombolestan, A. Rasooli, and S. Khodaygan. Optimal path-planning for mobile robots to find a hidden target in an unknown environment based on machine learning. *Journal of Ambient Intelligence and Humanized Computing*, Mar 2018.

[38] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[39] J. Sztipanovits, T. Bapty, X. Koutsoukos, Z. Lattmann, S. Neema, and E. Jackson. Model and tool integration platforms for cyber–physical system design. *Proceedings of the IEEE*, 106(9):1501–1526, Sep. 2018.

[40] András Vörös, Márton Búr, István Ráth, Ákos Horváth, Zoltán Micskei, László Balogh, Bálint Hegyi, Benedek Horváth, Zsolt Mázló, and Dániel Varró. Modes3: model-based demonstrator for smart and safe cyber-physical systems. In *NASA Formal Methods Symposium*, pages 460–467. Springer, 2018.

[41] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. Torcs, the open racing car simulator. *Software available at http://torcs. sourceforge. net*, 4:6, 2000.

[42] Weiming Xiang, Patrick Musau, Ayana A. Wild, Diego Manzanas Lopez, Nathaniel Hamilton, Xiaodong Yang, Joel A. Rosenfeld, and Taylor T. Johnson. Verification for machine learning, autonomy, and neural networks survey. *CoRR*, abs/1810.01989, 2018.

[43] Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. Output reachable set estimation and verification for multi-layer neural networks. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, March 2018.