CPS Design with Learning-Enabled Components: A Case Study

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

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

Xenofon Koutsoukos
Janos Sztipanovits
Gabor Karsai
Institute for Software Integrated Sys.
Vanderbilt University
gabor.karsai@vanderbilt.edu

ABSTRACT

Cyber-Physical Systems (CPS) are used in many applications where they must perform complex tasks with a high degree of autonomy in uncertain environments. Traditional design flows based on domain knowledge and analytical models are often impractical for tasks such as perception, planning in uncertain environments, control with ill-defined objectives, etc. Machine learning based techniques have demonstrated good performance for such difficult tasks, leading to the introduction of Learning-Enabled Components (LEC) in CPS. Model based design techniques have been successful in the development of traditional CPS, and toolchains which apply these techniques to CPS with LECs are being actively developed. As LECs are critically dependent on training and data, one of the key challenges is to build design automation for them. In this paper, we examine the development of an autonomous Unmanned Underwater Vehicle (UUV) using the Assurance-based Learning-enabled Cyber-physical systems (ALC) Toolchain. Each stage of the development cycle is described including architectural modeling, data collection, LEC training, LEC evaluation and verification, and system-level assurance.

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. CPS Design with Learning-Enabled Components: A Case Study. In *Proceedings of 30th International Workshop on Rapid System Prototyping(RSP'19), New York, NY, USA, October 17–18, 2019 (RSP'19),* 7 pages.
https://doi.org/10.1145/3339985.3358491

RSP'19, October 17–18, 2019, New York, NY, USA

ACM ISBN 978-1-4503-6847-6/19/10…$15.00
https://doi.org/10.1145/3339985.3358491

ACRONYMS

ANN Artificial Neural Network
ALC Assurance-based Learning-enabled CPS
CPS Cyber Physical System
CNN Convolutional Neural Network
GSN Goal Structuring Notation
LEC Learning Enabled Component
UUV Unmanned Underwater Vehicle
ROS Robot Operating System
URI Uniform Resource Identifier

1 INTRODUCTION

Cyber Physical Systems (CPSs) are used for a wide variety of applications in many domains which often require significant or total autonomy. Moreover, these systems frequently operate in highly uncertain environments where it is infeasible to explicitly design for all possible situations within the environment. Data-driven methods such as machine learning are being applied in CPS development to address these challenges and Learning Enabled Components (LECs) have demonstrated good performance for a variety of traditionally difficult tasks such as object detection and tracking [15], robot path planning in urban environments [33], and attack detection in smart power grids [27]. However, little tool support exists currently for the development of these systems.

Traceability and reproducibility at every step of development is necessary for CPSs, particularly those used in safety-critical or mission-critical applications. These systems require strong safety assurance supported by well-documented evidence. Regulations for these systems often require that developers follow specific procedures and provide documentation of the development process (eg. Design Assurance Levels (DAL) defined in DO-178C [30]). Additionally, traceability and reproducibility are necessary for maintaining data provenance when working with LECs. Since LECs are trained with data instead of derived from analytical models, the quality of an LEC is dependent on the history and quality of the training data. Manually maintaining traceability and reproducibility during the design cycle - including the large data sets used for LECs - is a time-consuming and error-prone process. Instead, this responsibility should be automated by an appropriate development environment.

Integrated tool support for the development of CPSs which use LECs is an active area of research, and the Assurance-based Learning-enabled CPS (ALC) Toolchain [13] is one such development environment. In this paper, we examine the development of an autonomous Unmanned Underwater Vehicle (UUV) using the ALC Toolchain with discussion on each stage of the development cycle: architectural modeling, data collection, LEC training, LEC

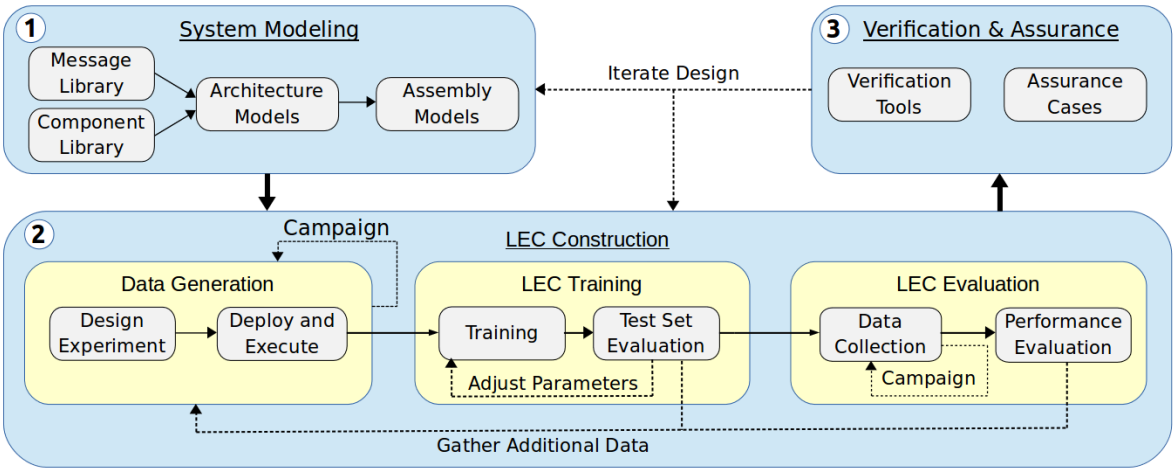

Figure 1: ALC Toolchain development workflow.

evaluation and verification, and system-level assurance. Additionally, key features of the toolchain are described including the ability to rapidly iterate during the design cycle, automated data tracking and management, and reproducible experiments and LEC training.

The rest of this paper is organized as follows. First, Section 2 discusses related research regarding challenges unique to machine-learning and existing development environments for CPS. Section 3 briefly describes the ALC Toolchain and presents the UUV example used as a case study. Next, Section 4 outlines the modeling concepts provided by the toolchain, followed by a description of the development process for an LEC in Section 5. Section 6 describes the available LEC verification tools and assurance techniques. Section 7 gives a brief explanation of how the various data management features are implemented in the toolchain. Finally, Section 8 discusses possible directions for future research and concluding remarks.

2 RELATED RESEARCH

Conventional CPS design is a well studied problem with many supporting tools available. In particular, model-based techniques have been effective for the development of CPSs with conventional, analytically derived components. For example, the Model-based Demonstrator for Smart and Safe Cyber Physical Systems (MoDeS3) [36] combined several modeling languages and techniques for the development of a model railway system. The authors modeled the system architecture with SysML block diagrams, then used code generation tools provided by the Gamma Statechart Composition Framework [24] for implementation of the software components. Multiple safety assurance techniques including formal verification methods and run-time monitors were utilized. Tool sets such as the INTO-CPS platform [20] support multi-disciplinary development by providing environments which integrate several modeling languages. Similarly, existing standards such as the Functional Mockup Interface (FMI) [3] and System Structure and Parameterization (SSP) [18] provide tool-independent specifications for co-simulation of these multi-domain systems. However, for a large class of problems it is becoming clear that machine learning will outperform conventional model-derived techniques due to the difficulty of developing accurate analytical models. Current CPS tools do not consider how the existing model-based development techniques can be extended for machine learning.

Machine learning solutions often outperform traditional techniques for a variety of challenging problems, but development of these systems also presents unique challenges. Sculley *et al.*[31] consider the hidden costs of machine learning with the software engineering concept of *technical debt*. They examine several ways in which machine learning can incur significant costs both during the development cycle and for long-term maintenance of a system. Methods to identify and mitigate these costs are presented, several of which can be enforced with an appropriate methodology and automated with supporting tools (e.g. careful management of data dependencies among components and reproducibility of training and evaluation).

Various existing tools facilitate development of machine learning models, particularly for users who are not experts in the field. DeepForge [7] is one such environment which provides a model-based approach to machine learning development with concepts for defining neural network architectures and corresponding training pipelines. DeepForge also uses a version control system to ensure traceability and reproducibility during development. For reinforcement-learning techniques, the OpenAI Gym [6] toolkit provides a standardized set of environments for development and comparison of different learning algorithms. The Google Colaboratory [1] is another example based on the Jupyter Notebook [2] interactive environment with strong support for machine-learning, but provides an ad-hoc environment to the user instead of following any predefined methodology. Additionally, the TensorFlow [1] software library provides the TensorBoard toolkit for visualization of TensorFlow graphs including machine learning models. These tools simplify and automate much of the machine learning development process, but they do not sufficiently maintain data provenance particularly when using custom generated data. However, these tools are not concerned with how these techniques can be incorporated into a larger development framework tailored for CPSs.

[1] colab.research.google.com
[2] jupyter.org

3 ALC TOOLCHAIN

The ALC Toolchain is an integrated set of tools and corresponding workflows specifically tailored for model-based development of CPSs that utilize learning-enabled components. Figure 1 shows an overview diagram of the general workflow which is detailed in the following sections. In order to promote reproducibility and maintain data provenance, all models, training data, and contextual data are stored in a version-controlled database and data management is automated. The toolchain is built on the WebGME infrastructure [22] which provides a web-based, collaborative environment where changes are automatically and immediately propagated to all active users, not unlike in Google Docs [3]. More detail on how these features are implemented can be found in Section 7.

For this paper, we developed an autonomous UUV controller using the ALC Toolchain, as a case study. The goal of this system is to follow a pipe placed on the seafloor using images from a forward-looking camera. The system was developed following a component based software engineering paradigm [35]. Currently, the ALC Toolchain supports the Robot Operating System (ROS) [4] [28] middleware which provides a communication framework for exchanging messages between components, but support for other similar frameworks can be added. Experiments were performed using the Gazebo [5] [17] simulation environment with the UUV Simulator [6] [21] extension packages. The system is used as a running example throughout this paper and is described in more detail in the following sections.

4 SYSTEM MODELING

The first step in CPS development is specification of the system architecture and the various components it is composed of. The ALC Toolchain provides modeling concepts for representing components and their interfaces, the ROS message types used to communicate between components, composed system architecture models, and environment models. Currently, architectural models in the ALC Toolchain can be used to generate executable ROS launch deployment files for the composed system. However, no behavioral code is generated for individual components.

4.1 Message Type and Component Libraries

Before a system architecture model of the UUV can be constructed, the available message types and components must be defined in corresponding block libraries. The *message library* defines the structure of all ROS message types used in the system. Message definitions are similar to C-style structures, and each message type is named for both the ROS package where it is defined and the data it represents.

Once the available message types have been defined, blocks for each component in the system can be constructed in the *component library*. First a component block defines an interface by declaring the input and output ports used by the component. Each port has an associated message type from the message library and a topic name used to identify the port in the ROS publish/subscribe messaging system. Component blocks may contain any necessary parameter definitions as well as a ROS launch file for initializing and configuring the component. Each component has an associated block-type such as "hardware", "software", or "implementation" which is discussed in more detail in section 4.2. Additionally, standard component blocks may be composed hierarchically to form complex components. For our UUV example, the message and component libraries contained 27 and 21 elements respectively.

4.2 System Architecture Model

The ALC Toolchain provides an extended version of block diagram models from the SysML modeling standard [26] for system architecture modeling. Blocks from the component library are instantiated in a *system architecture model* and their ports can be connected to form a composed system model. Since all component blocks in a model are instances of the original block defined in the component library, any changes made to the original block are automatically propagated to all instances. This approach promotes reusability and maintainability of components across multiple system models.

For our UUV example, a complete system architecture model was constructed including both hardware and software components. Figure 2 shows a portion of this model containing the vehicle controller software which must autonomously operate the UUV to follow a submerged pipeline placed on the seafloor. This controller consists of four primary components: a path planner, a PID controller, and thruster and fin mappings. The path planner component consumes sensor inputs (images from a front-facing camera and vehicle odometry data) and produces a desired heading value (i.e. desired vehicle yaw). Multiple implementations of this component were used as discussed in Section 4.3. The PID controller calculates the error between the current and desired vehicle heading and applies a correction to eliminate this error. Additionally, vehicle pitch and roll components are controlled with fixed setpoints for the desired depth and roll. Finally, the thruster and fin mappings translate desired thrust and vehicle orientation into actuator commands for the propeller and control fins respectively.

4.3 Assembly Model

During the design of a CPS, there may be multiple implementation options for a particular component. This is particularly true during the development of LECs where training data may be generated by a conventional 'controller' or 'perception' component and used to train an LEC. As an example, the path planner in Figure 2 is a complex component shown in an exploded view so that the various sub-components can be seen. Three of these sub-components are labeled as *implementations*, indicating that there are three different alternatives available which are capable of fulfilling the path planner's requirements. In this case the three options are as follows:

- A conventional control algorithm which receives ground-truth data from the simulator about the position of the pipe being tracked. Note that this data would not be available in a real-world deployment, making this implementation a simulation-only option.
- An LEC using a neural network trained with supervised learning to mimic the conventional algorithm. The LEC only takes camera sensor images as input and can be used in both real and simulated environments.

[3] docs.google.com
[4] ros.org
[5] gazebosim.org
[6] uuvsimulator.github.io

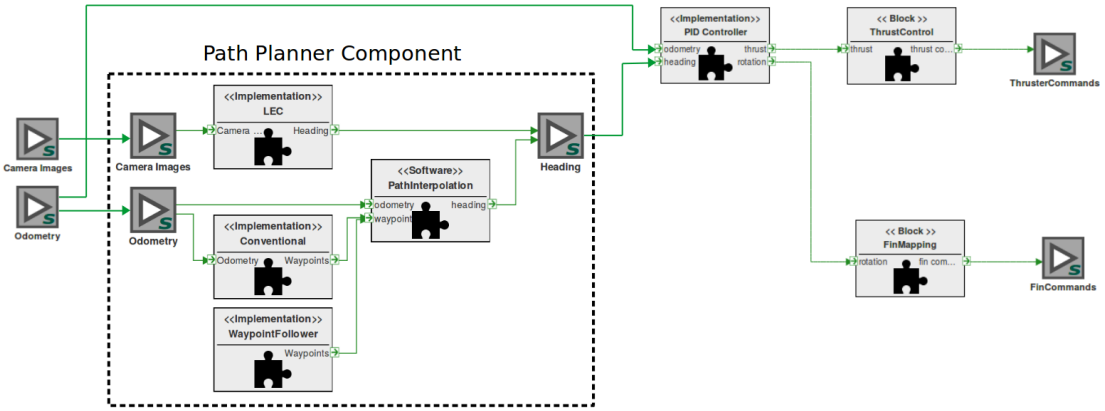

Figure 2: UUV controller architecture diagram.

- A waypoint following algorithm which follows a set of fixed waypoints provided as a static parameter.

Assembly models refine an architectural model by selecting a concrete implementation from the available options for each component with multiple implementation options. In our example, the path planner component is the only component with alternative options and there are only three valid assembly models of our system (one for each path planner implementation).

5 LEC CONSTRUCTION

Once the system architecture and available assembly models have been defined, the user can move to the next step in the development workflow: LEC construction. LEC construction involves multiple sub-steps which vary depending on the training method (ie. supervised or reinforcement learning). These include data generation, LEC training, and LEC evaluation as described in the following sections.

5.1 Data Generation

Development of an LEC is a data-driven process that requires generating sufficient data from the system for training the LEC. *Experiment* models allow the developer to define an execution of a system in a particular (simulated) environment. Construction of an experiment model begins by selecting the desired system configuration from one of the available assembly models. Next, an *environment block* is used to define the simulation environment and set any environmental parameters. *Mission* and *Execution* blocks contain parameters which define the objective of the experiment and any additional housekeeping settings (e.g. maximum timeout, startup delays, data storage location, etc.) respectively. Experiment models contain the information necessary to configure a simulation and can be deployed and executed on an appropriately configured server. When an execution run is complete, all generated data is automatically stored and made available in the ALC Toolchain.

Experiment models can be extended with additional blocks for more functionality. *Campaign* blocks can define multiple possible values for a parameter and are useful for performing multiple experiments while varying environmental conditions, mission objectives, or system parameters. When a single campaign block specifies sets of values for multiple parameters, the complete parameter set is defined as the Cartesian product of all the individual sets. When a campaign is executed, the corresponding experiment is executed once for each valid combination of parameters in this set. *Post processing* blocks define Python code to be executed on the generated data after an experiment completes. These blocks are often used for data analysis tasks such as calculating performance metrics, plotting relevant data, etc.

To generate training data for the UUV example, we created an experiment model based on the conventional path planner assembly described in Section 4.3. The experiment was configured to use Gazebo with the UUV Simulator extensions and used a flat, empty seabed for the operating environment. For this system, the objective of an experiment always involves following a pipe placed on the seafloor. However, this can be accomplished in one of three ways: autonomous control by the path planner component, manual control using a joystick, or by following a predefined set of waypoints. The autonomous control option was selected by setting the mission parameters appropriately.

A campaign block was added which defined six different pipe geometries, each consisting two straight sections joined by a bend of varying angle. The campaign specified that each pipe should be tested in both standard and mirrored orientations (i.e. pipe with a left-hand bend would be mirrored to test the right-hand bend case). Additionally, each experiment was done once following the ideal path as closely as possible, then again with noise added to the path in order to generate data in non-ideal conditions resulting in a total of 24 experiments. All available simulation data was stored by the ROS recorder utility in the ROS bag file format and used for training the LEC as described in the following section.

5.2 LEC Training

The next step in the development of an LEC is training a machine learning model. The ALC Toolchain supports training of Artificial Neural Networks (ANNs) [14] using one of two classes of machine learning algorithms: supervised or reinforcement learning (For detailed introductions see [11] and [34] respectively). Both classification (discrete output range) and regression (continuous output range) based models are supported.

5.2.1 Supervised Learning. The goal of supervised learning is to approximate the ideal mapping function from a set of input

variables to a corresponding set of output variables as closely as possible. This requires generating *labeled* data where the correct value of each output point is known before starting LEC training. In CPS design, supervised learning is commonly used to imitate an existing controller that cannot be used in a real-world system deployment. For instance, our UUV example generates labeled data in a simulated environment with a controller which uses the true location of the submerged pipeline to plan an ideal path for the vehicle. However, the true location of the pipeline would not be available in a real-world deployment. Instead, an LEC was trained to approximate the ideal controller using images from the forward-looking camera as input.

In the ALC Toolchain, training of an LEC using a supervised learning approach is defined by a *SL Training* model. Constructing this model starts with the definition of an *LEC model* which specifies the desired ANN architecture as well as basic formatting functions used to interface with the toolchain. A *training data* block specifies which of the available data sets should be used to train the ANN. The available data sets include any data generated by Experiment models (described in Section 5.1) and any data generated externally from the ALC Toolchain which has been uploaded using the provided utility. A *training parameter* block defines hyperparameters which configure the training algorithm. Finally, an optional *post processing* block defines Python code to be run after training is complete.

Design of a suitable ANN architecture (or selection from existing architectures) is often an iterative process driven by empirical results. Performance indicators including model loss and accuracy against a testing data set are the most common evaluation metrics. CPSs often operate in resource-constrained environments and must also consider factors such as model size, inference time, and power consumption. To support rapid design iterations, a new LEC architecture can be tested simply by updating the architecture definition - provided as a Keras[8] model - within the LEC model block and executing the SL Training model.

In our UUV example, three existing Convolutional Neural Network (CNN) architectures of varying complexity were evaluated. First, an architecture presented by Rausch *et al.* for end-to-end steering control of a simulated autonomous vehicle [29]. This architecture, hereafter referred to as Rausch-Net, consists of three convolutional layers followed by a single fully-connected layer. Second, the Nvidia DAVE-2 architecture [4] developed for a similar purpose and tested using a real vehicle on public roadways. The DAVE-2 architecture consists of five convolutional layers followed by three fully-connected layers. Third, a modified version of AlexNet [19] with only one convolutional pipeline instead of two known as CaffeNet [7]. CaffeNet contains five convolutional layers followed by two fully connected layers. The final output layer in each architecture was modified to output a single value.

The LEC model block included a simple Python class (33 lines of code) with functions for extracting and formatting the desired data streams from the full data set. In particular, this class used OpenCV [5] to resize images from the forward-looking camera from their original 768 x 492 resolution to the appropriate size for each CNN architecture. The data set generated from the Campaign described in Section 5.1 was selected for training data. A random split was used to divide this data set into 80% training data and 20% testing data. Table 1 provides a comparison of the three tested CNN architectures. The 'Size' column reflects the size of the trained network when stored on the hard-disk using the save function provided by Keras and is an indication of overall model complexity. The 'Test Set Loss' column lists the Mean Squared Error (MSE) of each trained model when used to predict the outputs of each point in the testing data set. The final column labeled 'Heading Error' is discussed in Section 5.3.

Table 1: Comparison of CNN Architectures

Model Name	Size (MB)	Test Set Loss (MSE)	Heading Error (RMSE, deg)
Rausch-Net	308.3	0.0020	0.590
DAVE-2	3.0	0.0018	0.439
CaffeNet	535.1	0.0453	3.643

5.2.2 Reinforcement Learning. Reinforcement Learning (RL) has evolved to be a powerful data-driven technique in which the learning (or training) happens in a closed-loop interaction between the agent and environment. Unlike supervised learning, RL does not require labeled data during the training process. Instead the goal of reinforcement learning is to learn a policy of selecting the best action from a list of actions for a given state, that maximizes the reward function. Similar to supervised learning, RL also has two phases called the exploration and exploitation phases, which correspond to training and testing respectively. The entire learning in the exploration phase is performed using a reward based system, where the agent receives a positive reward for correct actions, and a negative reward otherwise. During the exploitation phase, the agent uses the learned actions (normally stored as tables, or trained neural networks). Several RL algorithms exist with various learning strategies (model-based vs. model-free) and optimality criteria (policy vs. value), and the ALC toolchain currently supports the actor-critic method [23].

In supervised learning, there is a clear separation between data collection and training activities. However, RL relies on learning by selecting actions at run-time, then using a reward based evaluation system to converge on optimal actions with increased agent-environment interactions. This process requires a closed loop agent-environment simulation, and the *RL Training* model available in the ALC Toolchain is very similar to the *Experiment* model described in Section 5.1. The primary difference is the addition of an *RL Agent* block which defines both the actor and critic models used for training. While RL learning is supported by the ALC Toolchain, it was not utilized by our UUV example.

5.3 LEC Evaluation

The training metrics discussed in Section 5.2.1 are useful for determining how well a particular ANN has learned to approximate the provided training data set. However, these metrics are not always good indicators for how well a trained LEC will perform when deployed as part of a larger system. This is often a result of an incomplete training data set which does not capture a sufficient range of possible inputs. For LECs used as control components, this issue is more pronounced since a trained LEC controller may drive

[7] https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet

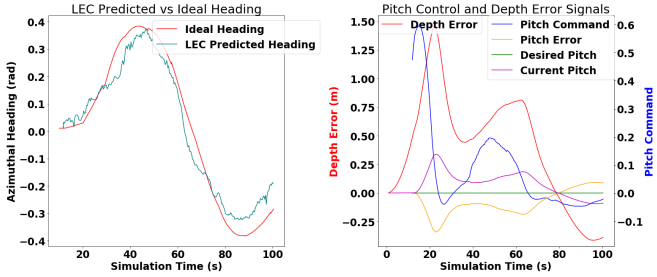

Figure 3: Post processing plots from DAVE-2 LEC evaluation experiment. LEC predicted and ideal headings vs. time (left). Pitch control and vehicle depth error signals vs. time (right).

a system into previously unseen states which were not represented in the training data set. Instead of relying on training metrics, a trained LEC can be evaluated as part of an integrated system using an *Experiment* model.

In order to evaluate the three LEC architectures trained for our UUV example, an experiment model was constructed similar to the model described in Section 5.1 with a few differences. First, the *assembly* model was changed to use the LEC implementation option for the path planner component instead of the conventional implementation. Second, a new pipeline model was selected which was not contained in the data set used for training. This pipeline model was similar to the models used for training, but included multiple bends - one 60° bend to the left followed by a 30° bend to the right placed a short distance apart. Finally, additional Python code was added to the *post processing* block which calculated the Root Mean Squared Error (RMSE) between the heading predicted by each LEC and the ideal heading provided by the conventional path planner.

The calculated heading RMSE values, listed in degrees, are in the last column of Table 1. The DAVE-2 architecture outperformed both other networks in both test set loss and evaluated heading error for our UUV path planner use-case, while also being the smallest and simplest architecture tested. The amount of training data generated may not be sufficient to accurately train the two larger networks. Additionally, CaffeNet was originally designed for image classification tasks which may have contributed to the order of magnitude accuracy decrease.

The left plot in Figure 3 shows both the LEC predicted heading and ideal heading values against time when using the trained DAVE-2 architecture. The LEC predicted heading contains significant noise, but otherwise closely tracks the ideal heading value. A similar plot was automatically generated for each evaluation experiment and made available to the user in an interactive Jupyter notebook. While the ALC Toolchain is intended for the development of LECs, the automatic plotting functions also proved useful for parameter tuning of the PID-based vehicle depth control. The right plot in Figure 3 shows the depth error from a fixed setpoint along with multiple signals for monitoring the current and desired vehicle pitch.

6 VERIFICATION & ASSURANCE

CPSs used in mission-critical or safety-critical applications demand high reliability and strong assurance for safety. Safety cases are one method for safety assurance which requires assembling multiple sources of supporting evidence (eg. testing data, formal verification, expert analysis, etc.) into convincing arguments for system-level safety. This approach has long been accepted by certain industries (eg. UK Ministry of Defense [25]), and has gained popularity in areas including CPS software development with more regulating bodies publishing guidelines and standards for their use (eg. Appendix D of FAA Unmanned Aircraft Systems Operational Approval document [2]). The ALC Toolchain uses Goal Structuring Notation (GSN) [16] to allow for construction of these safety cases.

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 LECs are an active area of research [32], [37] which will need to be incorporated into the safety assurance of these systems. The ALC Toolchain provides the *Verification* model concept with some initial support for these tools.

For our UUV example, a safety case was developed where the primary safety goal of the system is to avoid collision with the submerged pipeline at all times. Additionally, the system should keep the pipe in view of the camera while progressing at a set minimum speed. The completed safety case consisted of approximately 100 total blocks constructed hierarchically. If the formal verification results invalidate a required system property or if the system is deemed unsatisfactory after construction of the assurance case, then the developer should return to one of the previous development stages to address the issue as shown in Figure 1.

7 IMPLEMENTATION

The ALC Toolchain utilizes WebGME's built-in version control system to maintain a complete history of all models created and allow developers to revert their models to any previous state as needed. However, this system is not suitable for storing the large data sets generated by simulations and used for training of LECs. Instead, data files are uploaded to a secondary file storage server and a Uniform Resource Identifier (URI) is generated containing both the location and SHA-1 [9] hash of the data. This URI is stored in the version control system, and the SHA-1 hash algorithm ensures that data stored on the file storage server remains identical to when it was originally generated.

Whenever any model is executed, all parameters and configuration data needed to repeat the execution are stored in a metadata file with the results. This metadata file also contains references to any artifacts used as inputs to the model in order to maintain data provenance. Metadata files for LEC training contain the URI of each data file used in the training set as well as a copy of the parent LEC metadata if training was continued from a previously trained ANN model. Similarly, metadata for an evaluation experiment contains references to any trained LECs used in the experiment. This ensures that the complete history of any artifact can be traced back to the original data regardless of how many iterations of the design cycle are required. Additionally, the toolchain includes a dataset manager for viewing and analyzing this lineage.

8 CONCLUSION

We described the development of an LEC for an autonomous UUV using the ALC Toolchain. This model-based approach provided significant benefits over an ad-hoc development process by reducing the time to iterate design cycles, automatically handling data management and tracking data lineage, ensuring experiments were reproducible, and providing tools for verification and safety assurance. In particular, the toolchain simplifies the development of ANNs for CPS developers who are not machine-learning experts.

We have also identified possible avenues for future work. First, techniques for formalization and quantitative evaluation of safety case arguments have been developed and would be a useful for CPSs used in safety-critical or mission-critical applications. However, these techniques have recently been subject to criticism [12] and further refinement is needed to address the issues identified. Second, LEC performance is highly dependent on the quality and coverage of the training data. The ALC Toolchain provides the Campaign extension to Experiment models to allow for data generation in a wide variety of possible scenarios. However, the current method of defining a Campaign is ill-suited to specification of complex scenarios. We are currently integrating a more powerful and expressive scenario specification language (SCENIC [10]) for this purpose.

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] M. Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2015.
[2] Federal Aviation Administration. Unmanned Aircraft Systems (UAS) Operational Approval. *online: faa.gov/documentLibrary/media/Notice/N_8900.227.pdf*, 2013.
[3] Torsten Blochwitz, Martin Otter, Martin Arnold, Constanze Bausch, H Elmqvist, A Junghanns, J Mauß, M Monteiro, T Neidhold, Dietmar Neumerkel, et al. The functional mockup interface for tool independent exchange of simulation models. In *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*, number 063, pages 105–114. Linköping University Electronic Press, 2011.
[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] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.
[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.
[7] Brian Broll, Miklos Maroti, Peter Volgyesi, and Akos Ledeczi. DeepForge: A Scientific Gateway for Deep Learning. In *Gateways 2018*, 9 2018.
[8] François Chollet. Keras. https://keras.io/, 2015.
[9] D. Eastlake, 3rd and P. Jones. Us secure hash algorithm 1 (sha1). 2001.
[10] Daniel J Fremont, T Dreossi, S Ghosh, X Yue, A L Sangiovanni-Vincentelli, and S A Seshia. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 63–78. ACM, 2019.
[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.
[12] Patrick J Graydon and C Michael Holloway. An investigation of proposed techniques for quantifying confidence in assurance arguments. *Safety science*, 92:53–65, 2017.
[13] C Hartsell, N Mahadevan, S Ramakrishna, A Dubey, T Bapty, T Johnson, X Koutsoukos, J Sztipanovits, and G Karsai. Model-based design for cps with learning-enabled components. In *Proceedings of the Workshop on Design Automation for CPS and IoT*, pages 1–9. ACM, 2019.
[14] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.
[15] 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.
[16] 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.
[17] 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.
[18] Jochen Köhler, Hans-Martin Heinkel, Pierre Mai, Jürgen Krasser, Markus Deppe, and Mikio Nagasawa. Modelica-association-project "system structure and parameterization"–early insights. In *The First Japanese Modelica Conferences, May 23-24, Tokyo, Japan*, number 124, pages 35–42. Linköping University Electronic Press, 2016.
[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
[20] P. G. Larsen, J. Fitzgerald, J. Woodcock, P. Fritzson, J. Brauer, C. Kleijn, T. Lecomte, M. Pfeil, O. Green, S. Basagiannis, and A. Sadovykh. Integrated tool chain for model-based design of cyber-physical systems: The into-cps project. In *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*, pages 1–6, April 2016.
[21] 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.
[22] 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.
[23] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.
[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] UK Ministry of Defense. Safety management requirements for defence systems, June 2007.
[26] OMG. OMG Systems Modeling Language (OMG SysML), Version 1.5, 2017.
[27] 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.
[28] 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.
[29] Viktor Rausch, A Hansen, E Solowjow, C Liu, E Kreuzer, and J K Hedrick. Learning a deep neural net policy for end-to-end control of autonomous vehicles. In *2017 American Control Conference (ACC)*, pages 4914–4919. IEEE, 2017.
[30] RTCA. DO-178C - Software Considerations in Airborne Systems and Equipment Certification. December 2011.
[31] 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.
[32] Sanjit A. Seshia and Dorsa Sadigh. Towards verified artificial intelligence. *CoRR*, abs/1606.08514, 2016.
[33] 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.
[34] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
[35] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.
[36] 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.
[37] 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.