[KPRS19] E. Kusmenko, S. Pavlitskaya, B. Rumpe, S. Stüber:
On the Engineering of AI-Driven Systems.
In: ASE'19. Software Engineering Intelligence Workshop (SEI'19), pp. 126--133, San Diego, Nov. 2019.
www.se-rwth.de/publications/

On the Engineering of AI-Powered Systems

Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, Sebastian Stüber
Chair of Software Engineering, RWTH Aachen University, Aachen, Germany, kusmenko@se-rwth.de

***Abstract*—More and more tasks become solvable using deep learning technology nowadays. Consequently, the amount of neural network code in software rises continuously. To make the new paradigm more accessible, frameworks, languages, and tools keep emerging. Although, the maturity of these tools is steadily increasing, we still lack appropriate domain specific languages and a high degree of automation when it comes to deep learning for productive systems. In this paper we present a multi-paradigm language family allowing the AI engineer to model and train deep neural networks as well as to integrate them into software architectures containing classical code. Using input and output layers as strictly typed interfaces enables a seamless embedding of neural networks into component-based models. The lifecycle of deep learning components can then be governed by a compiler accordingly, e.g. detecting when (re-)training is necessary or when network weights can be shared between different network instances. We provide a compelling case study, where we train an autonomous vehicle for the TORCS simulator. Furthermore, we discuss how the methodology automates the AI development process if neural networks are changed or added to the system.**

***Index Terms*—MDE, deep learning, neural networks**

I. Introduction

Intelligent systems benefit more and more from the advances in machine learning. Potential application domains include autonomous driving, medical diagnostics, social networks, and speech processing, to name only a few. The rising demand for machine learning in industry and academia leads to the question, how next generation software using the new technology can be developed efficiently. Deep learning is inherently different to today's software development methodologies and requires a rethinking of paradigms, languages, and processes.

In this paper we are going to tackle this challenge by analyzing, how the deep neural learning domain can be grasped by domain specific language concepts, how these concepts can be integrated with existing methodologies, and how the usages of domain specific modeling frameworks can affect the overall system engineering process. As the basis of our work we will use MontiAnna, a Domain Specific Modeling Language (DSML) framework for the design and training of artificial neural networks. In this paper, we are going to show, how MontiAnna is embedded into the Component & Connector (C&C)-based architecture modeling language EmbeddedMontiArc and discuss how the software engineering process can benefit from this symbioses by demonstrating the design and evolution of a deep learning based autonomous driving model tested in a racing simulator. Furthermore, we show how components can be enriched by additional information concerning deep learning aspects using the tagging approach.

This work was supported by the Grant SPP1835 from DFG, the German Research Foundation.

II. Background

This work heavily builds on the two Domain Specific Language (DSL) families MontiAnna [1] and EmbeddedMontiArc [2], [3], which we are going to introduce in this section.

EmbeddedMontiArc is a C&C-oriented modeling language with a particular focus on embedded, automotive, and cyber-physical systems, It is inspired by Simulink [4] and developed as a compatible textual alternative to tackle some of its shortcomings, like the lack of dynamic runtime reconfiguration and a unit-based type system. In EmbeddedMontiArc components are first-level citizens encapsulating functionality and communicating with other components only via clearly defined interfaces. A component's interface is composed of a set of strictly and statically typed input and output ports, cf. L.3-6 in Fig. 5. EmbeddedMontiArc provides an abstract mathematical type system hiding the implementation details and letting the developer concentrate on the application logics. The type system comes with the primitive types `N`, `Z`, `Q`, `C`, `B` denoting the non-negative integers, integers, rationals, complex numbers, and Booleans, respectively. Furthermore, a data type can be refined with a range to incorporate the limits of the system to be modeled, e.g. the type in L.3 of Fig. 5 represents images and therefore only allows values from 0 to 255. The compiler decides automatically which implementation data type is most appropriate for each individual case. A data type can be extended to a vector, a matrix, or a higher order tensor using a LaTeX-inspired circumflex notation followed by the respective dimensions in curly brackets (`^{...}`), as is done in L.3 of Fig. 5 to model an image as a tensor with its dimensions representing the width, height, and the channels of the input image. Moreover, a type can be extended by a physical unit it represents, e.g. the `speedIn` port in L.4 of Fig. 5 expects *meters per second* (`m/s`). This enables the compiler to identify connections between incompatible ports (e.g. velocities and masses) as well as to convert compatible units automatically (e.g. `km/h` to `m/s`).

EmbeddedMontiArc components can be hierarchically decomposed into smaller subcomponents, which can be instantiated using the `instance` keyword, cf. L.8-15 in Fig. 5. A dataflow from an output port to an input port is established explicitly by creating a connector using the `connect` keyword, cf. L.17-27. Implicit communication is forbidden to ensure testability and maintainability. If a component cannot or

should not be decomposed into smaller components any more, it can be implemented using MontiMath, the matrix-oriented behavior language of EmbeddedMontiArc incorporating the strict type system and designed for the implementation of math-heavy algorithms. The EmbeddedMontiArc code generator creates pure C++ code as well as a corresponding CMake build infrastructure from EmbeddedMontiArc and MontiMath models, which can then be used to create executables using off-the-shelf software.

MontiAnna is a textual modeling framework for the design and training of Artificial Neural Networks (ANNs). It consists of two main languages: the neural architecture language, used to describe the structure of the neural network being modeled, and the training language, specifying how this architecture has to be trained. A basic architecture example is given in Fig. 1. The header is defined using the keyword `architecture`, followed by the network name and an optional set of parameters which can be used to adapt the network to specific applications. In MontiAnna, a neural network is built from neuron layers. A neuron layer is an array of unconnected neurons of the same type. MontiAnna supports many types of neuron layers used in practical deep learning engineering, such as fully connected (with respect to the previous layer), convolutional, ReLU, and LSTM layers.

The neural network is assembled by instantiating different types of neuron layers and by creating directed connections between them using the "`->`" operator, cf. L.5-9 in Fig. 1. Thereby, the output of the operand to the left of the "`->`" operator is used as input to the operand on the right. For the design of more complex architectures, MontiAnna offers the parallelization operator "`|`", which can be used to create parallel propagation paths. If a neuron layer is regarded as a node in a graph, by using these two operators, we can model any possible directed acyclic graph (DAG). However, this is not always sufficient, in particular, if we need to model recurrent network architectures. In MontiAnna we can instantiate anonymous layer instances, e.g. the `FullyConnected` layers in Fig. 1. This is convenient, since most of the time we never refer to an instantiated layer again. However, we can instantiate a named layer instance before adding it to the neural network, e.g. if we need to create cycles. Special named layer instances are the input and the output layers, defined in L.2-3 of Fig. 1. These layers define the entry and exit points of the network, thereby constituting its interface.

For the description of the training procedure, MontiAnna offers a dedicated training DSL, cf. Fig. 7. The training language allows the definition of primitive and nested hyperparameters such as the number of epochs to train, the batch size to use, and the optimization algorithm to apply including its own hyperparameters like learning rate. The training model is a recipe for the MontiAnna code generator to generate the training code which, in turn, is used to create the final neural network.

For more details on MontiAnna language elements, the code generation framework, and the variability concepts we refer to the basic MontiAnna paper [1].

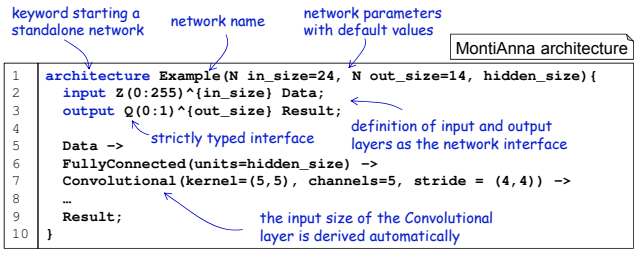

Fig. 1. MontiAnna architecture model with a parameterizable interface and two hidden layers.

III. Related Work

The rise of machine learning leads to a demand for frameworks, which simplify the development of programs using machine learning. In this section we discuss Model Driven Engineering (MDE) solutions for the deep learning domain used in practice and compare them to MontiAnna.

Traditionally, machine learning methods are offered by libraries such as TensorFlow [5], Torch [6] or Keras [7]. They provide methods to build, train and evaluate neural networks in general programming languages like C++ or Python. This allows the direct use of machine learning alongside existing code. The high-level framework Keras is especially popular since it greatly abstracts from the details of the neural network. A detailed overview of these frameworks based on general purpose languages (GPLs) is given in [1].

Solutions with a stronger focus on the domain allowing the developer to concentrate on the "what" instead of the "how" are required. MDE frameworks and graphical modeling tools enable even untrained users to create solutions interactively. Sensible default values, automatic hyperparameter tuning [8] and complex out-of-the-box components reduce the burden on the AI developer.

IBM SPSS Neural Networks adds neural networks to *IBM SPSS*, a software for statistical analytics. Instead of defining the neural network manually, an algorithm can create the network architecture automatically. Hence the neural network can be used as a black-box without configuration, while still being specific to the users problem. IBM SPSS focuses on traditional statistical tasks using neural networks, for example as an alternative to linear regression. Hence, image processing and similar problems are out of scope. A detailed description of the capabilities and multiple demonstrations are given in [9].

Azure Machine Learning is an environment for machine learning developed by Microsoft. It is integrated into further cloud services from Microsoft. Azure is also based on a C&C architecture, but represented purely visually. The predefined components focus on forecasting and classification. After answering a few questions concerning the problem, the developer gets recommendations for components and tutorials how to use them. Many components have configuration options. If

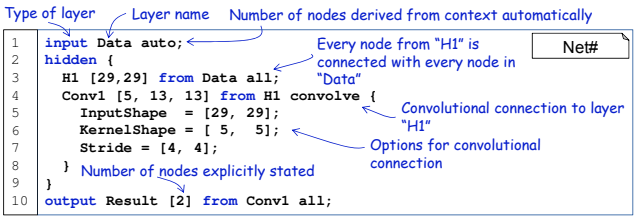

Fig. 2. Neural network architecture defined in Net#.

the predefined components are not enough, Azure offers the integration of R or Python code.

Furthermore, *Net#* was developed by Microsoft as a purely textual specification description language for neural networks. It focuses on complex neural networks architectures, for example deep neural networks. Azure allows the definition of custom components specified in Net# alongside normal components. A Net# specification consists of layers and the connection between these layers. Multiple hidden layers can be defined, each with a custom number of nodes and an activation function. The number of nodes in the input and output layers can be automatically deduced from the context. For hidden layers there is no smart automatic computation of the number of nodes. Instead, a constant value is used as default value. To repeat a layer in the architecture, multiple layers have to be defined manually.

The resulting graph must be acyclic. This requirement excludes cyclic network architectures like recurrent neural networks (RNNs). Apart from that there are no restrictions on layer connections. A layer can be connected to multiple layers both for input and output.

There are multiple configuration options, how the nodes of both layers can be connected. Default is the "full" connection, which connects every source node with every destination node. In the "filtered" connection a predicate expression can remove some connections. The predicate is a Boolean expression taking the indices of two nodes as input and returning "true" iff the nodes should be connected.

Many neural networks for image classification [10], [11] use a convolutional architecture. Net# supports this kind of layer and allows multiple parameters to customize it.

To compare MontiAnna and Net#, an example is given in Fig. 2. It describes the same network architecture as the MontiAnna specification in Fig. 1. The network has one input layer named "Data", the two hidden layers "H1" and "Conv1" and one output layer "Result". The number of nodes in the input layer is automatically derived. Layer "H1" contains $29 \times 29 = 841$ nodes, logically arranged in a two dimensional array. The second hidden layer "Conv1" contains $5 \times 13 \times 13 = 845$ nodes. All layers are connected in series; the layer "Conv1" is a convolutional layer with configuration parameters in L.5-7.

Net# can only be configured with the predefined options. There are ten different activation functions, but a user is unable to define a custom activation function. Same holds for the type of nodes and the connection between layers. This restricts the usability of Net# for many use cases.

IV. Modeling AI-driven Architectures

A. The modeling language family EmbeddedMontiArcDL

Autonomous cyber-physical systems can often be decomposed into multiple modules dealing with tasks like perception, navigation, planning, control, etc, which need to be developed by experts and then put together. The design and implementation processes of these modules can be as different as their purposes and might require the usage of different and appropriate paradigms and tools. The goal of this paper is to present a multi-paradigm modeling language family tackling the design and composition of math-heavy and deep learning-based systems. Therefore, we are going to discuss our methodology using an example developing an autonomous racing car system based on the direct perception principle [12] and evaluating it in the TORCS simulator [13]. The idea behind this system is that a vehicle, instead of using a set of classical sensors, e.g. to measure the distance to the vehicle in front, receives a picture from a front camera and tries to extract this information using a neural network. This information can be thought of as a vector of fourteen so called *affordance indicators* including the yaw angle of the vehicle relative to the road, the distance to the left/right/central lane markings, distance to the preceding vehicle in the left/right lanes and so on. As the name suggests, the direct perception network covers only the perception part of the system. Denoising, decision making, and control need to be designed separately, in contrast to end-to-end systems where a neural network receives an image as input and produces the actuator commands as its output [14]. The advantage of the direct perception architecture is that we can benefit from neural image processing without giving away control over the vehicle's behavior to a black-box network we don't fully understand.

Our aim is to develop the racing car using an MDE approach so that we can tackle the heterogeneous problems using appropriate paradigms. In particular, we want to model the system architecture using the EmbeddedMontiArc C&C language and benefit of its type system making physical quantities explicit. Furthermore, we need to model and train the direct perception neural network using a deep learning framework, i.e. MontiAnna. Finally, we need to design the signal processing parts such as filters and controllers using a language designed to describe mathematical operations, i.e. MontiMath.

However, working with different languages in a single project can be cumbersome, require a lot of glue code not contributing to the application logic, and lead to a complex build infrastructure. Therefore, we are going to introduce a composed language and generator family, where the single languages are adapted and can talk to each other. This facilitates the multi-paradigm modeling using the incorporated languages and enables comprehensive inter-language checks and tools. In the following, we will refer to this language family as EmbeddedMontiArc+DeepLearning (EMADL). EMADL is con-

structed using the language composition principles *language extension*, *language aggregation*, and *language embedding* of the language workbench MontiCore 5 [15], [16]. The main languages and generators of EMADL are depicted in purple and pink, respectively, in Fig. 3.

The C&C language EmbeddedMontiArc serves as a basis for our language family. However, it can only describe hierarchical component-based architectures. We use it to subdivide our racing car system into smaller components as is shown in the block diagram in Fig. 4 to tackle the development in a divide & conquer manner. The input and output ports of a block are depicted as small rectangles on its left and right side, respectively. The parent component containing all other components gets an image and the current velocity as input. The output consists of the command ports for acceleration, braking, and steering. Additionally, we output the smoothed version of the predicted affordance indicators for testing and validation purposes. The subcomponents of the main component are described in the following.

The DPNet is a Convolutional Neural Network (CNN) predicting affordance indicators for input images according to the direct perception approach. The architecture of DPNet follows the one defined in [12] and is derived from AlexNet [17] with the following changes: the local normalization layers are removed and two further fully connected layers with 256 and 14 units respectively are added. Moreover, [13] suggests to add a sigmoid layer between the final fully-connected layer and Euclidean loss. This change aims at normalizing the loss to the range [0.1, 0.9]. However, we observed that adding this layer leads to a considerably slower convergence. This might be because removing the sigmoid layer allows for outputs with larger discrepancy to true value to learn faster. On the other hand, if a sigmoid layer is attached, outputs that are wildly off might actually learn more slowly because of the saturation of sigmoid layer. Therefore, we decide to omit the sigmoid layer in the final architecture.

Since the DPNet returns normalized predictions, we have to rescale them back to the original ranges in the `Denormalizer` component. The denormalized affordance indicators are then stored in an `Affordance` structure (similar to a C/C++ struct).

The predictions of the DPNet, similar to ordinary sensor signals, are noisy. Therefore, we apply Kalman filtering to smoothen the affordance indicators. Kalman filters are commonly used for smoothing out noise in signal processing applications including neural networks. Since we need to filter multiple values independently, the `KalmanFilterBank` contains an individually parameterizable filter for each of the signals to be filtered.

The `Localization` component estimates the number of lanes by analyzing the affordance indicators, namely the distance between outer left and right lane markings, and provides this information to the `DriverController` so that an appropriate behavior can be selected.

The `DriverController` is a complex component which realizes the vehicle control based on the current affordance indicators and mostly follows the corresponding algorithm in the DeepDriving project [12].

The main task of the controller is to decide, whether a car should accelerate or brake and how much to steer in which direction. These three values are calculated based on the current speed, distance to the lane markings and presence of the other cars in the currently occupied and neighboring lanes. The input values for the `DriverController` component are affordance indicators received from the DPNet component as well as the current vehicle speed. For instance, the maximal speed is set to 70 km/h. If a car drives slower than that, the acceleration command is set to a positive (normalized) value in the range $(0, 1]$. The same logic applies to the brake command if the car drives faster than 70 km/h. Another factor is a distance to the preceding car - if it is less than 20 m, brake and acceleration commands are calculated to execute the car following behavior. The third factor is a sum of the five previous steering commands. If a car was steering significantly and the current speed is too high, then speed should be reduced. This way a vehicle is driving slower while passing through sharp curves.

Finally, the steering command is calculated based on the distance to the lane markings. The `DriverController` makes the car drive in the middle of the current lane until another car is detected in front. In this case, if another free lane is present, the `DriverController` switches to the free lane.

The `SteeringBuffer` component is utilized by the `DriverController` to buffer the steering command, lane change state as well as lane change timers between controller calls. In particular, a history of up to five steering commands is used in the `DriverController` to calculate the speed reduction when the car is performing a long turn. When a lane change maneuver is performed, a special flag is set to notify the controller that the car is in the process of lane changing and not simply turning. For convenience we introduce the `LaneChange` enumeration for lane changing maneuvers with the values `NO_CHANGE`, `TO_LEFT`, `TO_RIGHT`, `IN_RIGHT`, and `IN_LEFT`. We use lane changing timers for both changing to the left and to the right in the same way - these values are needed for a smooth lane change maneuver implementation.

The graphical architecture of Fig. 4 can be written in EmbeddedMontiArc according to the syntax introduced in section II. The code of the main component, named `DeepDriving`, is depicted in Fig. 5. Similar to the main component, the subcomponents instantiated in L.8-15 are defined in separate files and can contain subcomponents themselves. The EmbeddedMontiArc model can be generated by its corresponding generator EmbeddedMontiArc to C++ (EMA2CPP), cf. Fig. 3. The output is a C++ implementation of the component stubs, their interfaces, as well as the dataflow infrastructure, i.e. connectors between ports. Furthermore, the generator decides which types to use and creates unit conversion code, e.g. to convert a velocity from `km/h` to `m/s`.

Now let's go back to the `DPNet` component. Of course, deep learning functionality can be integrated into a language

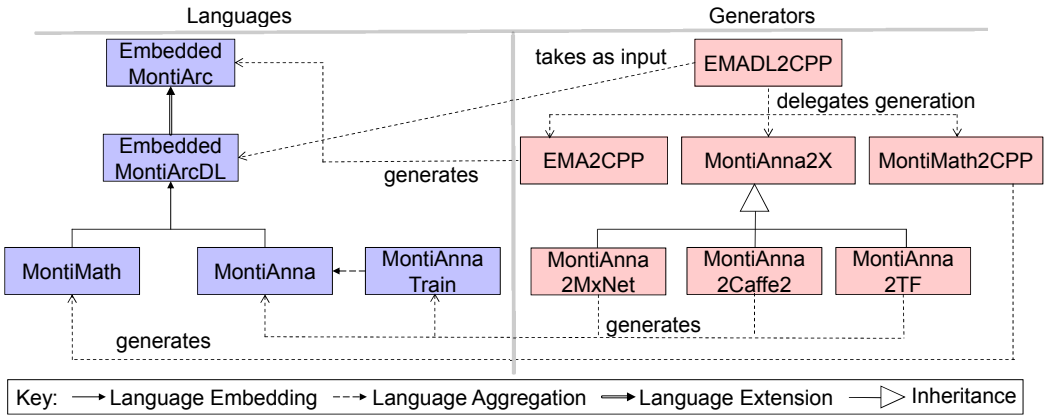

Fig. 3. The EmbeddedMontiArcDL modeling language family: languages an code generators are depicted as purple and pink boxes, respectively.

like MontiMath. However, this comes with the drawback that the framework engineers are constraint by the concepts provided by the host language. Furthermore, deep learning code can be intertwined with "standard" code hindering the code generators and compilers to identify neural networks and their interfaces and apply a machine learning specific life-cycle to them. Therefore, we integrate the MontiAnna architecture language with EmbeddedMontiArc by means of language extension and language embedding separately from MontiMath.

First, we create the EMADL language as an extension of EmbeddedMontiArc, which is similar to inheritance in object-oriented languages. We add a new rule to the grammar of this language allowing us to insert an implementation block into a component definition, cf. L.6 in Fig. 6. The block starts with the keyword `implementation` followed by the name of the implementation language (MontiMath or MontiAnna) and the actual code implementing the component behavior, cf. L.7-33.

Second, we *embed* MontiAnna and MontiMath, into the implementation block by using the main production rule of the respective language grammars. Note however, there is no MontiAnna header or input/output layer definition in the MontiAnna implementation block in Fig. 6. Instead, we map the component parameters to MontiAnna configuration parameters. The input and output ports of the component are created as MontiAnna input and output layers with the corresponding dimensions in the MontiAnna symbol table.

The implementation code of the `Dpnet` component contains only the pure neural network architecture code according to the description given above. The component input port `image` is used as the input layer of the network in L.21, while the final layer `predictions` is mapped to the component's output port with the same name.

The network structure built in L.21-33 is completely linear, i.e. we only need the "−>" operator to model the network. As is typical for image processing networks, we make strong usage of convolutional layers. Furthermore, we use fully connected, ReLU activation, and pooling layers. Dropout regularization is modeled in MontiAnna as a layer, as well, cf. L.18 and L.31. In L.8-14 and L.15-19 we define two custom layers, each composed of three predefined layers, which helps us keep the architecture modular and concise.

Context conditions of the EMADL language ensure that layer names and variables used in MontiMath and MontiAnna implementation blocks have been defined as EmbeddedMontiArc ports in the component header.

MontiAnna comes with the abstract code generator MontiAnna2X. This code generator provides the basic generation functionality for neural network, but is backend independent. Since we might want to use different deep learning frameworks depending on the application and the required functionality, we can add concrete MontiAnna code generators by subclassing MontiAnna2X if needed. Currently, we support generation of MxNet, Caffe2, and Tensorflow code, cf. Fig. 3.

All the subcomponents of Fig. 4 except the `Dpnet` are designed using the MontiMath language, which is particularly convenient for the mathematical algorithms such as the Kalman filter. Complete and fully functional models including the subcomponents as well as the tools needed to generate code and integrate it with TORCS are provided in our repository[1]. The MontiMath code of these components is generated using the MontiMath2CPP generator. Math operations are generated by the MontiMath2CPP generator using the Armadillo Basic Linear Algebra Subprograms (BLAS) library[2] to speed up computations at runtime.

However, the user does not need to run the three generators separately. Instead, we provide an orchestrating super-generator denoted as EmbeddedMontiArc+DeepLearning to C++ (EMADL2CPP), cf. Fig. 3. This generator gets EMADL models as input and delegates the actual generation to the respective specialized generators based on the modeling language being processed. Thereby, code generated from MontiMath and MontiAnna models by MontiMath2CPP and MontiAnna2X, respectively, is included directly into the respec-

[1]https://git.rwth-aachen.de/autonomousdriving/torcs_dl/tree/develop
[2]http://arma.sourceforge.net/

tive component code generated by EMA2CPP. Furthermore, EmbeddedMontiArc ports are mapped to MontiMath variables and MontiAnna layers based on their names. This allows the implementation code to read the component's actual input and provide output to the corresponding output ports.

Now, having introduced the main parts of the deep driving model as well as the languages and generators required to generate it, we ae going to discuss how the toolchain deals with training and how this can affect the development process.

To setup the training procedure, the MontiAnna code generator requires a training model as introduced in section II, cf. Fig. 7 for our DPNet example. The abstract syntax tree (AST) and symbol table of the architecture and training models are adapted internally using MontiCore's language *aggregation* concepts. This enables us to navigate from AST nodes and symbols of an architecture model to the ones of the training model and vice versa, e.g. to perform context conditions related to both models such as checking whether the loss function specified in the training model fits to the network architecture. The MontiAnna2X generator then works on this composed model containing all the necessary information to generate neural network and training artifacts.

B. Tagging AI Components

If MontiAnna is used to model and generate a standalone architecture as in Fig. 1, we can provide a training model and the path to the training data as parameters to MontiAnna2X. If we need to exchange the training model or the training data, we just need to adapt these parameters, e.g. through a command line interface. While this works well for isolated models, it is not a convenient approach for C&C architectures, possibly accommodating multiple deep learning components and having an arsenal of training models and datasets. Instead, we apply the so called *tagging* mechanism [18], to enrich the symbol table of the EMADL component model with additional information concerning the component training. Thereby, we first need to specify in a tagging schema which language elements, i.e. which symbol types, we want to be able to enrich with what kind of information, cf. listing at the top of Fig. 8. In this tagging schema, named `TrainingToEmadlTagSchema` we define the tag type `Training`. This tag type contains three entries: `datapath`, `dataformat`, and `training` specifying the path to the training data, its format, as well as the fully qualified name of the MontiAnna training model to be used. In L.6 we declare that this tag type is meant to tag components and component instances (the relation between the two is similar to the relation between classes and objects in object-oriented languages). This tagging schema is a recipe, we reuse for all EMADL models.

The listing at the bottom of Fig. 8 is a concrete tag definition file for our direct perception example. In L.1 we define that the tag model has to conform with the tag schema defined in the listing above. In this model we provide two tags. While the first tag, defined in L.4, is meant to tag the component `Dpnet` with a `Training` tag, the second tag is applied to the concrete component instance `dpnet`, instantiated as a subcomponent of the main component `DeepDriving`. The EMADL parser reads the tags together with the other model artifacts and attaches the data contained in the tags to the referenced symbols of the EMADL model. This is realized by the name resolving mechanism of the EMADL symbol table.

Tagging a component type, e.g. `Dpnet` with a tag means that the tag applies to *all* component instances of this type. On the other hand, tagging a component *instance* does not have an effect on other instances of this component type. Furthermore, a component instance tag overrides component type tag information, which enables us to define exceptions in large AI systems with many MontiAnna components.

For our example we have been discussing so far, one of the two tags defined in Fig. 8 would suffice. However, imagine that, in order to improve the precision of our direct perception system, we decided to install several cameras on our vehicle. Assuming that the errors emerging from the different mounting positions can be canceled by appropriate parameters, we can instantiate a `Dpnet` component for each of the cameras in our system. Of course, we would also like to use the same training model for each instance, i.e. we should use the component type tag given in L.4-8 of Fig. 8 (bottom). When processing the tagged model, the EMADL2CPP generator realizes that all `Dpnet` instances need to be trained using the same data stored in `/home/se/torcsdata` as an HDF5 database using the training model `Dpnet`. Therefore, it executes the training script only once and reuses the weights for all `Dpnet` instances. Furthermore, as the network is stateless, EMADL2CPP creates only a single *flyweight* instance which is reused for all `Dpnet` prediction tasks. This saves us loading the network weights for each instance individually (practical networks can have millions of parameters).

Now imagine, we want to apply the same network architecture to detect a set of different features, e.g. weather parameters like rain intensity or the state of the street we are driving on while keeping the `Dpnet` array for the affordance indicator prediction. We can instantiate a corresponding new component and tag it with a component instance tag as given in L.9-13 of the tag model in Fig. 8. Thereby, the data path is changed and the EMADL2CPP compiler knows that it has to train this component individually.

Once the training of all MontiAnna components is finished, the final executable system can be built. Thereby, the EMADL2CPP generator creates a log containing training meta-data for each trained component (instance). In particular, the training model used as well as the creation time of the training database as stored. Whenever the whole system is rebuilt, the EMADL2CPP compiler checks for every MontiAnna component if the architecture, the training model, and/or the dataset have changed based on this log. If this is not the case, training of these components is skipped.

C. Evaluation

For training and testing we used the DeepDriving dataset[3]. The training dataset contains 484.814 images, each labeled

[3]https://deepdriving.cs.princeton.edu

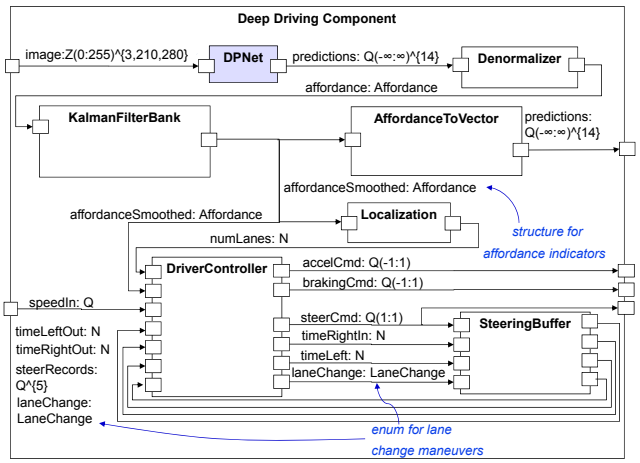

Fig. 4. C&C architecture of the direct perception based autonomous vehicle.

```
component DeepDriving<height=210, N width=280, N channels=3> {
  ports
    in Z(0:255)^{img_channels, img_height, img_width} imageIn,
    in Q(0 m/s:0.1 m/s:100 m/s) speedIn,
    out Q^{16} predictedAffordanceOut,
    out Q(-1:1)^{3} commandsOut;

    instance Dpnet<height, width, channels, classes> dpnet;
    instance DriverController driverController;
    instance Denormalizer denorm;
    instance KFMastercomponent kfm;
    instance SteeringBuffer steeringBuffer;
    instance VectorToAffordance vecToAffordance;
    instance LocalizationController locController;
    instance AffordanceToVector affToVector;

    connect imageIn -> dpnet.data;
    connect dpnet.predictions -> denorm.normalizedPredictions;
    connect denorm.affordance -> kfm.affordanceIn;
    connect kfm.affordanceSmoothed -> locController.affordanceIn,
         driverController.affordanceIn, affToVector.affordanceIn;
    connect locController.lanesCount ->
        driverController.lanesCountIn;
    connect affToVector.affordanceOut -> predictedAffordanceOut;
    connect steeringBuffer.outputBuffer -> driverController.
                                          steeringRecordIn;
    /*more connections, cf. graphical model*/
}
```

EMADL

Fig. 5. C&C architecture of the direct perception based autonomous vehicle formalized as an EmbeddedMontiArc model.

with 14 affordance indicators. Due to problems with the testing data set provided in [12], we decided to split their training dataset into training (80% of the original training dataset) and validation (20%) subsets. This way we train DPNet on 387.851 samples and validate on 96.963 additional samples not seen during training. The mean squared error of the affordance indicator prediction is plotted over the number of learning iterations in Fig. 10.

To finally test the deep driving controller, we need to integrate it with the TORCS simulator. Integration of distributed systems in the robotics or autonomous vehicles domain is usually achieved using a middleware such as ROS [19]. We reuse the tagging mechanism as proposed in [20] to add middleware specific information to the ports of our `DeepDriving` component to map EmbeddedMontiArc topics to ROS topics, cf. Fig. 9. The EMADL code generator creates additional ROS adapters for the tagged ports, which enables us to connect our

```
component Dpnet <img_height=210, N img_width=280,
                 N img_channels=3, N classes=14>{
  ports in  Z(0:255)^{img_channels, img_height, img_width} image,
       out Q(0:1)^{classes} predictions;

  implementation MontiAnna {
    def conv(kernel, channels, hasPool=true, convStride=(1,1)) {
      Convolution(kernel=kernel, channels=channels,
                  stride=convStride) ->
      Relu() ->
      Pooling(pool_type="max", kernel=(3,3), stride=(2,2),
              ?=hasPool)
    }
    def fc(){
      FullyConnected(units=4096) ->
      Relu() ->
      Dropout()
    }
    In/out ports of EmbeddedMontiArc are mapped to layers of the MontiAnna network
    image ->
    conv(kernel=(11,11), channels=96, convStride=(4,4)) ->
    conv(kernel=(5,5), channels=256, convStride=(4,4)) ->
    conv(kernel=(3,3), channels=384, hasPool=false) ->
    conv(kernel=(3,3), channels=384, hasPool=false) ->
    conv(kernel=(3,3), channels=256) ->
    fc() ->
    fc() ->
    FullyConnected(units=256) ->
    Relu() ->
    Dropout() ->
    FullyConnected(units=14, no_bias=true) ->
    predictions;
} }
```

network parameters as component generics

EMADL

The CNN interface is defined using component ports

implementation language is set to MontiAnna

User-defined layers

Fig. 6. Definition of a component with a MontiAnna architecture description of the DPNet neural network.

```
training Dpnet(…) {
  num_epoch : 100,
  batch_size : 64,
  eval_metric : mse,
  context : gpu,
  load_checkpoint : true,
  normalize : true,
  optimizer : sgd {
    learning_rate : 0.01
    learning_rate_decay : 0.9
    step_size : 8000
  }
}
```

Training

The name of the training model; parmeters can be used to set up a configuration, e.g. for automated experimentation

The stochastic gradient descent optimizer is a complex parameter consisting of multiple algorithm specific child parameters

Fig. 7. The training model for the DPNet neural network.

autopilot with TORCS without writing any glue code.

The generation and training process as well as the final result, i.e. the racing car driving in TORCS, can be seen in a demonstration video[4]. Further example projects supporting the latest features can be found in our applications repository[5].

V. Conclusion

In this paper we presented EmbeddedMontiArcDL, a language family combining component-based architecture with deep learning modeling. The language family was presented and discussed using the models of an autonomous driving software for the TORCS simulator. Thereby, we discussed how automation in the development process of AI systems emerges from coupling the C&C paradigm with an explicit modeling of neural networks. For instance, treating weights as versionable artifacts, smart weight management and reuse can lead to enhanced efficiency of the overall development process, if the compiler can manage the life-cycle of deep learning components.

[4]https://youtu.be/hfICK4f-hR4

[5]https://git.rwth-aachen.de/monticore/EmbeddedMontiArc/applications

Tag schema

```
tagschema TrainingToEmadlTagSchema  {
  tagtype Training {
    datapath = ${path:String},
    dataformat = ${format:[HDF5 | LMDB]},
    training = ${modelName:Name}
  } for Component, ComponentInst ;
}
```

Tag model

```
conforms to TrainingDataToEmadlTagSchema

tags TrainingTags  {
  tag Dpnet with Training = {
    datapath = „/home/se/torcsdata“,
    dataformat = HDF5,
    training = Dpnet;
  }
  tag DeepDriving.dpnet with Training = {
    datapath = „/home/se/torcsdata2“,
    dataformat = HDF5,
    training = Dpnet;
  }
}
```

Fig. 8. The tagging schema for tagging EMADL components with training data is given in the listing at the top; a tagging example attaching tags to the Dpnet component instance and the Dpnet component type is depicted below.

Tag schema

```
tagschema RosToEmamTagSchema  {
  tagtype RosConnection {
     topicName = (${topicName:String},
      topicType =  ${topicType:String})
      (, msgField = ${msgField:String})?)?
  } for Port;
}
```

Tag model

```
conforms to de.monticore.lang.monticar.generator.roscpp.
                                    RosToEmamTagSchema;
tags DeepDrivingRosTags {
  tag DeepDriving.imageIn with RosConnection =
      {topic=(/camera, std_msgs/Float32MultiArray)};
  tag DeepDriving.speedIn with RosConnection =
      {topic=(/speed,std_msgs/Float32)};
  tag DeepDriving.groundTruthAffordance with RosConnection =
      {topic=(/affordance,std_msgs/Float32MultiArray)};
  tag DeepDriving.commandsOut with RosConnection =
      {topic=(/commands,std_msgs/Float32MultiArray)};
  tag DeepDriving.predictedAffordanceOut with RosConnection =
      {topic=(/predictions,std_msgs/Float32MultiArray)};
  }
}
```

Fig. 9. The tagging schema for tagging EMADL ports with ROS middleware information is given in the listing at the top; a concrete tagging example attaching tags to the ports of the DeepDriving component is depicted below.

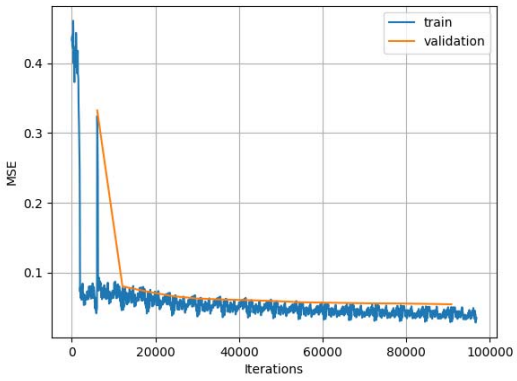

Fig. 10. Mean squared error of the affordance indicator prediction measured on the training and on the validation dataset, respectively.

References

[1] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*. IEEE, September 2019.

[2] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[3] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *MoDELS'18*, 2018.

[4] Mathworks Inc. Simulink User's Guide. Technical Report R2019a, MATLAB & SIMULINK, 2019.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[6] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, number EPFL-CONF-192376, 2011.

[7] François Chollet et al. Keras: Deep learning library for theano and tensorflow. *URL: https://keras. io/k*, 7:8, 2015.

[8] Rmi Bardenet, Mtys Brendel, Balzs Kgl, and Michle Sebag. Collaborative hyperparameter tuning. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 199–207, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[9] Keith McCormick and Jesus Salcedo. *Model Complex Interactions with IBM SPSS Neural Networks*, pages 325–353. 04 2017.

[10] 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.

[11] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. Convolutional networks and applications in vision. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 2010.

[12] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[13] 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.

[14] 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. Technical report, NVIDIA, 2016.

[15] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[16] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*. 2012.

[18] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*. ACM/IEEE, 2015.

[19] Morgan Quigley, Ken Conley, Brian 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*, volume 3, page 5. Kobe, Japan, 2009.

[20] Alexander Hellwig, Stefan Kriebel, Evgeny Kusmenko, and Bernhard Rumpe. Component-based Integration of Interconnected Vehicle Architectures. In *30th Intelligent Vehicles Symposium (IV'19). Workshop on Cooperative Interactive Vehicles*, pages 146–151. IEEE, June 2019.