# Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems

Evgeny Kusmenko
Department of Software Engineering
RWTH Aachen University

**Abstract.** The development of intelligent and interconnected cyber-physical systems is an interdisciplinary challenge requiring appropriate processes, languages, and tools supporting the engineering team. In this dissertation a model-driven architecture-centric approach for intelligent CPS design is presented. The foundation of this methodology is given by the architecture description language EmbeddedMontiArc. It enables a structural decomposition of the software system under development into hierarchically organized components. Based on the component-and-connector paradigm, the components of the system are side-effect free units with clear interfaces manifested as typed ports. Components communicate exclusively over explicit connectors between their ports. Implicit communication over shared memory and side-effects are forbidden in order to ensure maintainability, testability, and reusability.

EmbeddedMontiArc is equipped with an abstract type system designed for the engineering domain. Its primitive types represent mathematical sets such as integer, rational, and complex numbers. The technical details and the decision how a given mathematical type will be implemented on the target platform are delegated to the code generator or the compiler. To account for finite operational ranges of physical systems, a primitive type can be constrained by lower and upper bounds and a resolution. Furthermore, a primitive type can be enriched by a physical unit it represents. This enables the compiler to check the physical compatibility of connected ports and to perform automated type conversion of compatible units, hence eliminating a common source of errors in physical systems design and implementation. Primitive types can be composed into matrix types and structures to account for complex data flows in cyber-physical systems.

Intelligent cyber-physical systems need to adapt to new situations at runtime steadily. Cooperation with other similar systems encountered at runtime but not known at design time might be necessary in order to improve the overall system performance and to ensure the achievement of goals in the best possible way. For this reason, runtime reconfigurations of a cyber-physical system's interfaces as well as its internal structure must be possible by design. EmbeddedMontiArc offers a modeling framework for the dynamic parts of the system enabling the designer to model runtime instantiation and rewiring of components and their interfaces. These reconfigurations can be activated at runtime by triggers. Triggers are defined in the models as Boolean expressions on port values, i.e. a runtime reconfiguration can become active when a port reads a specific value, e.g. when the velocity of the cyber-physical system surpasses a predefined threshold. Furthermore, triggers can be based on other preceding reconfigurations. This enables the definition of reconfiguration chains as sequences of atomic architectural modifications. For instance, the perception of another cyber-physical system might lead to the instantiation of a dedicated communication interface. The creation of the interface leads to the instantiation of further components, e.g. for messaging, collision checking, cooperative planning, etc.

While the behavior of the designed system can be assembled from basic blocks similar to Simulink, a possibility to implement component behavior by means of other languages and paradigms is desirable. For this purpose, the presented methodology introduces two domain-specific languages. Component behavior code written in one of these two languages is embedded into the implementation block of the component and uses component ports to interact with the outer world. The first language, MontiMath, is inspired by MATLAB and enables the developer to describe mathematical algorithms in a procedural way. This language adopts the type system discussed above, thereby rendering modeling of physical processes less prone to unit-related errors.

Since artificial intelligence (AI) and deep learning have been gaining more and more interest and have been applied successfully in a variety of domains in the last years, the proposed methodology introduces MontiAnna, a second domain-specific language enabling the developer to implement the behavior of a component as a deep neural network. Providing a dedicated language instead of neural network libraries for MontiMath has two main reason: first, neural networks require a different thinking and hence, can be grasped better using an appropriate language. Second, a clear encapsulation of neural networks as components and their separation from general purpose code is the basis for the automation of the machine learning development cycle.

Targeting industrial applicability and easy usage by engineers and developers who are no AI experts, MontiAnna employs a high level modeling approach regarding neural networks as directed acyclic graphs (DAGs) of neuron layers which is in line with popular frameworks such as Keras or PyTorch. MontiAnna provides a library of different layer types, which can

be used by the developer to instantiate neuron layers in a neural network model. The library was designed based on a thorough analysis of state-of-the-art deep learning architectures and includes layer types covering fully connected or dense neuron layers, convolutional layers, attention layers, etc. Further neuron layer types can be created by combining existing ones to a more complex structure, by extending the internal library, or by providing a layer implementation in Python. The neuron layer DAG representing the neural network to be trained is created by interconnecting the layer instances with each other using a convenient connector syntax. To facilitate the instantiation of sequences of similar layers, the languages supports layer structural parameters and layer stacking by design. For instance, the layer constructor parameters can be given as lists instead of single values, indicating that a layer sequence is to be created with each element of the sequence being parameterized with the corresponding parameter of the parameter list.

Each MontiAnna component requires a JSON-like configuration file containing the hyperparameters for the training. This way, the compiler toolchain is able to detect all the deep learning components, locate the training and test data, and conduct a training using the given hyperparameters automatically. The compiler toolchain keeps track of each component's lifecycle. When the component-based system is recompiled, only those deep learning components are (re-)trained which have no satisfying training results yet or whose training data or hyperparameter configuration have been changed. Furthermore, if multiple component instances of the same neural network exist in the system and have the same training data attached to them, training is performed only once and each instance reuses the same trained weights. The development team hence, does not need to take care of training and retraining manually nor to write code for the training procedure. To accomplish this, MontiAnna offers the concept of training pipelines. A training pipeline is a module encapsulating the training logic used at compile (or training) time to optimize the weights of a given machine learning model. Each pipeline has an individual set of hyperparameters which can be set and tuned by means of the aforementioned configuration file.

MontiAnna was evaluated on multiple training pipelines including supervised, generative adversarial, reinforcement learning, and other learning approaches. To facilitate maintainability and extensibility of the pipelines, the hyperparameters for each pipeline are defined in a separate schema model. The employed configuration schema framework supports inheritance, thereby enabling the reuse and specialization of configuration schemas. For instance, MontiAnna offers multiple reinforcement learning pipelines having overlapping sets of hyperparameters. The hyperparameter set common to all these pipelines is defined in an abstract reinforcement learning schema. The specialized schemas inherit from this general schema and add their respective individual parameters.

Some pipelines require components at training time which become obsolete at runtime. For instance, in reinforcement learning sometimes a second neural network, the critic network, is used during training. The component representing the critic network at training time needs to be set using our configuration file in a dependency injection manner. Instead of defining the needed training time components in the schema models, we introduce the concept of reference models. A reference model is an EmbeddedMontiArc-based architecture which is instantiated at training time. The concrete components are injected via the configuration file, i.e. the reference model serves as a handy extension of the schema models.

Furthermore, a reference model serves as a structural foundation of the respective training pipeline, enabling the pipeline developer to access the components in the pipeline code. What's more, by defining the dataflows in the reference model explicitly, type and compatibility checks of the training components can be performed using the standard EmbeddedMontiArc context conditions (for instance, the state type of the actor network has to be the same as the one of the critic network as these two are linked by a connector in the reference model).

The modeling approach presented in this thesis is fully generative, i.e. complete, executable C++ and python code is generated out of the models. Manual code additions are neither required nor recommended. For neural networks the user can choose between several state-of-the-art target platforms including MXNet Gluon, TensorFlow, and Caffe2.