An Internal DSL for Implementing Complex Training Workflows for Model Predictive Contollers

Stijn Bellis*, Joachim Denil*, Ramesh Krishnamurthy† and Guillermo A. Pérez†
* *Cosys-Lab*, † *Ansymo*
University Of Antwerp, Flanders Make@UAntwerpen
Antwerp, Belgium
{Stijn.Bellis, ramesh.krishnamurthy, guillermo.perez, Joachim.Denil}@uantwerpen.be

***Abstract*—In this paper, we introduce a domain-specific language that helps with creating complex workflows that are used when training model predictive controllers. The running example used is learning to imitate a controller for an inverted cart pole. The language is implemented as an internal domain-specific language in Python. First, the requirements of the domain-specific language are listed. Then to show the effectiveness of the domain-specific language, the domain-specific language is used to implement three methods: Dagger, NDI, and Curriculum Learning. Using the results of these implementations, it is shown that the domain-specific language fulfills the requirements. Lastly, we describe what could be done in the future to improve the domain-specific language.**

***Index Terms*—Artificial Intelligence, Data manipulation languages, Model predictive controllers**

I. Introduction

Model Predictive Controllers (MPC) solve very complex control problems using an executable model of the system, and optimization. However, the disadvantage is that model predictive controllers have a high computational cost. One way to deal with this high computational cost is to imitate the MPC using machine learning. There are several benefits of a machine learning MPC over the classical MPC. For example, machine learning models are often computationally more efficient than classical models. Furthermore, machine learning models are amenable to certain validation techniques to check the validity of the model [6].

However, these benefits of machine learning are only obtained once the model is properly trained. The training of a controller is especially difficult as the controller influences the next states it will visit during its operation. To properly train the controller we need to include all the states a controller will visit in the training set, but to include these states we need the trained controller to know which states the controller will visit. When the full "state space" is not taken into account, the controller will accumulate error at each timestep as the controller reaches states it was not trained on. Because this is a critical factor in data-driven controllers, several techniques are devised to solve these issues: e.g., Dagger, Polish, and NDI. These methods use iterative processes of training a controller. In each iteration, the method evaluates the controller and adapts the training data to train a new and hopefully better controller. This leads to complex workflows that can be hard to set up and change. One of the challenges is data manipulation. We want to manipulate data more abstractly when implementing a method like Dagger but more concretely when implementing the functions (for example to train the controller).

Flanders Make SBO - Deterministic and Inexpensive Realizations of Advanced Control - DIRAC

In this paper, we introduce a Domain-Specific Language (DSL) that allows control engineers to focus on complex training workflows. A DSL was chosen because we want the user to be able to abstractly define the method and more concretely define the techniques (like the controller training) in a modular fashion. We argue that creating a DSL is appropriate for this purpose. The DSL we introduce is an internal DSL. An internal DSL means that the DSL is built on top of another language and therefore can use the development tools of this language. It can also work together with already existing libraries for the host language [4]. The host language we implement the DSL in is Python. In section 2 the running example is introduced. In section 3, we discuss the background and related work. In section 4 we introduce the requirements for the DSL. In section 5 the DSL is explained and an example is given. In section 6 we validate if the DSL meets the requirements and lastly, in section 7 we discuss future work.

II. Running Example

The running example we use in this paper is a controller for an inverted pendulum. The controller is created using the Rockit (Rapid optimal control kit) framework [3]. The Rockit framework is used to solve optimal control problems. Rockit creates an expert policy to gather training data from. Rockit also gives us a function to simulate the system to get the next state.

The inverted pendulum use case consists of a cart with a pendulum attached to it. The cart can move only horizontally in one dimension and the pendulum is allowed to freely swing around. The goal of the controller is to keep the pole upright and move the cart to position zero. The input variables that the controller gets are:

- POS: The position of the cart in the horizontal dimension.
- THETA: The angle of the pendulum when measured from a line straight up from the cart.
- DPOS: The speed of the cart in the horizontal dimension.
- DTHETA: The angular velocity of the pendulum.

The control variable that the controller has is F a force on the cart in the horizontal direction. The force is limited between -2 N and 2 N. The sign of the force signifies the direction of the force. The cost function that the Rockit created controller is minimizing is $F^2 + 100 * POS^2$.

III. Background & Related Work

A. Model Predictive Control

Model predictive control is a control technique that uses a model to predict the future state of the system and adapts the control actions accordingly. It uses a cost function to evaluate which control actions to take. A basic model predictive controller consists of two parts: an optimizer and a model. The model predicts the future state of the system given the current state and control action. The optimizer will try to find an optimal set of control actions to minimize a cost function for a certain amount of steps in the future. The optimizer also takes any possible constraints of the optimal control problem into account. These constraints can be on variables in the system or the constraints can be on the control actions.

Model predictive controllers are useful because they are able to solve very complex control problems. The disadvantage of model predictive controllers is that they are often very costly and computationally more expensive than other more classical controllers (e.g., PID controllers). As such, it is interesting to imitate model predictive controllers using machine learning to reduce the computational load of model predictive controllers [5, 10].

B. Useful Terms

In this section, we define some terms to keep our language clear in the rest of the paper. The term *policy* is used to describe any mapping between the input and output. The mapping can be constructed using machine learning or other methods. The term *expert policy* describes a policy that is used to label the data. The expert policy will often use classical methods to map the input to the output e.g., a simulation. The term *trainable policy* refers to a policy that is constructed using machine learning. We use the term *method* to refer to the complex workflows that we want to implement. In the next section, we define the workflows we show in this paper. We use the term *technique* or *function* to refer to the building blocks of these methods like the machine learning functions that give trainable policies.

C. Workflows

Several workflows have been proposed to solve the accumulation of error described in the introduction. In this paper, we look at two of these workflows: Dagger and NDI. To show a more general use of our DSL we also implement a curriculum learning approach.

1) Dagger: The Dagger or Dataset Aggregation method was first introduced in [9]. The Dagger method consists of a loop that is run several times. First, it takes a linear combination of the initially trained policy and the expert policy. It uses the linear combination of these policies to generate trajectories. The states in these trajectories are then collected into a dataset and each state is assigned an action from the expert policy. This dataset with new points is then aggregated together with the dataset that was used to train the policy. This aggregated dataset is used to train a new policy. This loop is done several times to get a more robust policy.

2) NDI: NDI was first introduced in [1]. The idea behind NDI is the policy will only start learning the next step in the trajectory until the policy can do the current step. The NDI algorithm consists of a single loop. First, the starting points of trajectories are sampled and gathered with their corresponding action given by the expert policy. The trajectories in this dataset then are extended using the ExteND algorithm until the policy gives a bad output. The states of these trajectories are put into a dataset and this dataset is then used to train a new policy. The longest version of each trajectory is then saved to use in the next iteration.

3) Curriculum Learning: Curriculum learning is a set of workflows with the same general idea. The idea is to first train machine learning policies on easier examples of the problem and then gradually introduce harder examples. This should hopefully improve the performance of the machine learning policy. The Curriculum Learning algorithm that is used in this paper goes as follows. First, the dataset is split into a number of datasets that go from easy to hard. The datapoints are placed in these datasets depending on how hard the datapoint is to learn for the machine learning policy. For the running example, we have chosen to use the distance to position zero as a measure of hard a datapoint is to control. Other measures of difficulty could be used. Then the policy is recursively trained for each loop adding a more difficult dataset to the training dataset until all datasets are added [11].

D. Dynamic Management

Baumann. et al presented a Dynamic data management framework into the MontiAnna framework for Machine Learning [2]. The authors focus their efforts on making it easier to extend datasets efficiently and to (automatically) retrain a model when an extension to a dataset occurs. The workflow the authors focused on is in the context of DevOps. The workflows we are looking at in this paper are often of a more complex nature.

E. Auto Machine Learning

Auto Machine Learning is a group of tools that has the goal to automate the process of creating a machine learning policy. Auto machine learning systems can automate the training of the policy, selection, and exploration of model architectures. Some automation tools also exist for feature engineering and prediction engineering. Feature engineering is when more relevant features are constructed from the data to improve the accuracy of the policies that are trained on these features. Prediction engineering is labeling the data and creating meaningful training and testing sets. Most of auto machine learning tools operate with the idea that there is only one dataset that is being used and that this dataset is static.

This is not the case for the complex workflows that we are looking at. Auto ML still can be used with the DSL as the choice of how to construct the policy is free for the user [7].

F. Pandas

Pandas is a Python library that gives data structures and tools to work with these data structures. Pandas is a versatile library and gives many options to work with the data structures it has. Pandas is a very powerful tool but it is also built very generally. This allows for customization and flexibility. Pandas for example allows adding data that has elements missing. This is not something we want in the data structure of the DSL because adding non-existing data could lead to problems down the line. The data structures that are used in the DSL are built on the pandas dataframes to keep the powerful tools that the pandas library has but the DSL data structures also give more structure. The DSL data structures also have tools to make it easier to use on a more abstract level [8].

G. Credibility, DSL

In this paper, [12], Zucker. et al introduced Arbiter. Arbiter is a domain-specific language for ethical machine learning. Arbiter is a declarative programming language based on SQL. For ethical machine learning Arbiter focuses on fairness, accountability, transparency, and reproducibility. Arbiter is built to train machine learning policies from a singular and static dataset in an ethical way. While these are interesting and important concepts and these concepts could be looked at in the future. They fall outside of the scope of our current project.

IV. Requirements for DSL

The requirements for the DSL are as follows.

Requirement 1: Modularity The DSL should be constructed in a modular way. This makes it easier to implement and reuse code. Because the method is less intertwined with the techniques used in that method. This should improve the ability of the user to edit the code and it makes it easier for the user to explore other techniques without having to overhaul the code.

Requirement 2: Readability/understandibility When methods are implemented from scratch the structure of the method often gets lost through the implementation of the techniques. This is in contrast with the pseudocode of the method illustrated in the literature. The DSL should be able to retain this high-level view so that the method can be understood without having to look at the paper where the methods were introduced.

Requirement 3: Data management The DSL should make data management easier. The data should be able to be manipulated on an abstract level in the methods and at a more concrete level when used in a technique. The data container that is used should also be able to be passed between the method and techniques as easily as possible. The data container should also be easily transformed into common data types to make it easier to work with when implementing the techniques.

Requirement 4: Reliable The DSL should not affect the outcome of the method and should perform as well as when it was coded from scratch.

V. DSL

A. Overview DSL

As mentioned before the idea behind the DSL is to make it easier to create the complex workflows that come with methods like Dagger, NDI, and curriculum learning. The DSL we implemented is an internal DSL that is coded in Python. The reason for this choice is that Python is often used in data science and machine learning applications. We decided on an internal DSL to make the DSL more flexible and general so that more algorithms can be implemented using the DSL. The DSL consists of several classes. The DSL classes are split into three groups: the data structure classes, the policy class, and the function classes. In subsection V-B the structure of a program when implemented in the DSL is explained. In section V-C the data structure classes are explained. In section V-D the policy class is explained. In section V-D the Function classes are explained. Lastly, in section V-F the Dagger method implemented in the DSL is shown.

B. Structure of the DSL

Figure 1 shows the structure of the DSL. The method is implemented using the DSL functions and DSL policies. The user implements the functions and policies how they want. Furthermore, the DSL is then used to initialize the DSL functions and DSL policies with their implemented functions. Once initialized, the DSL functions and policies can be used in the method. The DSL functions and policies always transfer data using the DSL data structures. The parameters for functions and policies are passed using data classes that are specified when implementing the functions. Lastly, the operating script is the place where the parameters are specified and DSL functions and policies are initialized to run the method. This is done to keep all the changeable parameters in one place and minimize the amount of change to the code that has to be done when switching functions or policies.

C. Data structures

One of the things that complicates writing code for complex workflows is data management. To simplify data management the DSL implements its own data structures. There are three data structures that can be used: datapoint, dataset, trace dataset.

The datapoint is the simplest data structure it consists of a single datapoint. It is initialized using dictionaries or pandas dataframes. It has an input and output dataframe to keep track of the input data and the output data and it also keeps track of the name of the input and the output data.

The next data structure is the dataset. It can be seen as a group of datapoints. It is still saved as an input and output pandas dataframe for ease of working with them when implementing functions. The dataset is initialized with a dataframe or from a csv file. The dataset can also be appended with data

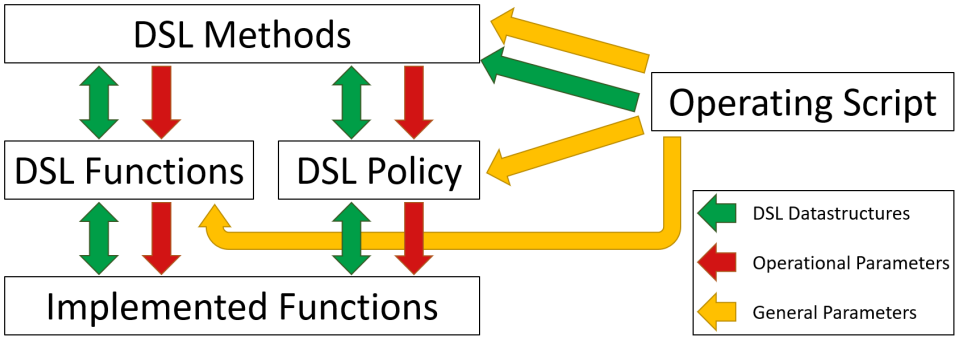

Fig. 1. The structure of the DSL

from a datapoint, dataset, or trace dataset. The input and output parts of the dataset are set and retieved by simply using .input or .output appendices. The dataset can be iterated over which gives the datapoints in the order they are stored. The dataset can be shuffled and split using custom functions. Lastly the dataset can also be saved to a csv file.

The last data structure is the trace dataset. A trace dataset is a group of traces. A trace is a dataset that has the property called 'is_trace' set to True. If a dataset is a trace it is protected against shuffling and splitting. Being a trace makes it possible for a dataset to be appended to a trace dataset. The trace dataset is initialized using a csv file. It can also append data from a dataset (if it is a trace) and from another trace dataset. The trace dataset can also append data from a dataset pointwise meaning each datapoint in the dataset goes to another trace in the trace dataset. The pointwise appending makes it easier when simulating each timestep for different traces at the same time. The trace dataset can also be written to a csv file. Lastly, the trace dataset can also be iterated over and will then give the datasets in the order they are stored.

All the DSL data structures are protected against wrong actions like merging an output dataset to an input dataset with fewer datapoints. As mentioned before the DSL data structures are built on pandas dataframe to make them easy to work with when implementing functions.

D. Policies

In complex workflows often there are several policies trained and used. The policies that are used do not always have the same structure. For example, the expert policy can be a simulation and the trained policy can be a neural network. The differences in structure make it hard to use these policies together and swap policies if wanted. The DSL policy solves this problem by being a wrapper that goes around any policy that can be used. The DSL policy makes sure that the input and output of the policy is a DSL datapoint so that the policies can be constructed in a modular way. The DSL policy will accept a dataset or a datapoint with only input values and return a dataset or a datapoint with input and output values.

One of the other features of the DSL policy is that new policies can be constructed using simple arithmetic on DSL policies. The output of the constructed policy is equal to the output of the arithmetic function. To calculate the output the arithmetic function swaps the policies it has as input with the output of these policies. DSL policies can be added, subtracted, multiplied, and divided with other DSL policies and numbers. This feature is used in methods like Dagger where a linear combination of the expert policy and the trained policy is asked for.

E. Functions

In a similar way as for the policies, there are a lot of functions that get used in complex workflows. The difference in kind between functions sometimes makes it hard to code in a general way and to swap them if another function is needed. For example, when you want to use another policy, the training function also has to change. The DSL includes wrappers to give training and other functions a general structure and protect the input and output of these functions. The parameters for implemented functions are passed using data classes. There are two types of parameters that are passed: (a) general parameters that are set when the function is initialized and, (b) operational parameters that are set when the function is run. In general, most of the parameters should be given as general parameters because this makes the code more general and easier to change the functions that are used. Now a short overview of the functions that exist.

Train Policy: This function trains a policy. The input for the train policy function is a dataset that has input and output values and the output of the train policy function is a DSL policy.

Simulate system: This function simulates a system to get to the next time step. The input for the simulate system function is a dataset or a datapoint with an input value and output value the output of the simulate system function is a dataset or a datapoint with only an input.

Simulate system traces: This function simulates traces given a policy and starting states for a certain number of steps. The input for the simulate system traces function is a policy, a dataset with the starting states (so only input values) for each trace and the number of steps the trace has to be. The output of the simulate system traces function is a trace dataset.

Validate dataset: This function compares two datasets. The validate dataset function is meant to compare a dataset gotten

from a policy with a dataset gotten from an expert policy but it also can be used for other purposes. The validate dataset function takes as input two datasets of equal length with the same input and output names. The output of the validate dataset function is not restricted.

Validate trace datasets: This function compares two trace datasets. The validate trace dataset function is meant to compare a trace dataset gotten from a policy with a dataset gotten from an expert policy but it also can be used for other purposes. The validate trace dataset function takes as input two trace dataset of equal length with the trace datasets having the same input and output names. The output of the validate trace dataset function is not restricted.

F. Dagger Example

In this section, we look at an implementation of the Dagger method. First, an implementation of the Dagger algorithm as described in the paper is shown. Then a more complex Dagger algorithm is shown. The more complex Dagger method is a more realistic type of Dagger algorithm that also tests the policy and saves results.

In figure 2, the comparison between the pseudo-code of the dagger method and the DSL implementation can be seen. The implementation is close to the implementation in the pseudo-code. A few things are different: (a) we ask to give an initial policy because the second line of the pseudo code is not very concrete. (b) We return all the trained policies because asking for the best policy is not well-defined. The rest of the code is almost analogous to the pseudo-code. Some extra lines are necessary to initialize a dataset and to do some data management. In the DSL code, we see that thanks to the DSL very complicated procedures like taking the linear combination of an expert policy and a trained policy or sampling T-set trajectories are expressed as a single line of code.

In figure 3, a complex version of the Dagger algorithm is shown. The complex Dagger algorithm has more features of traditional machine learning workflows like splitting the dataset in a training and testing dataset and saving data to see the performance of the policy and for later analysis. First, we see that the datasets are initialized in the Dagger method itself. This initialization makes it so that the operational script is cleaner. The operational script being cleaner helps with having a better overview of the parameters that can change. Then in the loop, an iteration output map is generated and the iteration output map is used in operational parameter data classes. The iteration output map is used to save results from training and validation in a separate folder for each iteration so that the policies and process can be better analyzed. Then the dataset is split into a training and test dataset and the training dataset is used to train the first policy. The splitting of the data in two datasets is standard practice in machine learning and allows for validation of the trained policy. Instead of asking for an initial policy, the policy is trained with the initial dataset this change keeps all the training of policies in the method and makes the operational script cleaner. This change also helps with a better overview. Then the output is gotten from the trained policy for the test dataset and the trained policy output is compared with the output from the expert policy. The results of this comparison are saved in the output folder and can be later used for analysis and to inform decision-making about the policy and the machine learning workflow. Then like before the traces are generated and these traces are compared with the traces from the expert policy. For this example, we load them from a pre-saved dataset because they don't change but the traces from the trained policy could also be obtained by the use of the simulate_system_traces function. The results of this comparison are also saved so that they can be analyzed later. Then the loop dataset is generated and appended to the total dataset like before. The last difference is that the total dataset gets saved so that it can be analyzed or used later.

The complex Dagger algorithm is coded on a very high abstraction level letting the user worry about the details of the implementation later. As shown, the DSL is very flexible in its use and easily allows alliterations to the code to soothe whatever the user wants to do. By making it an internal DSL it also allows custom functions and the use of familiar coding practices like the for loop. So that the implementation process of the method can be as frictionless as possible.

VI. Validation

In this section, we discuss if the DSL meets the requirements that were set out in Section IV.

The first requirement that we discuss is **Reuse/Modularity**. We want the DSL to be constructed in such a way that switching functions or methods should not require that much coding. This is mostly done by constructing the methods using standard function formats so that when a function is switched it does not affect the method. These DSL functions should also enable easier switching of methods because each method should be constructed using these standard functions and will only require to code custom functions if they are asked for in the method.

To check if the requirement of reuse/modularity is met, we show how switching the policy type affects the code. In our running example, the policy type used was neural networks. For this example, the policy is switched to random trees. To change the training functions only three things needs adaptations. In the method code, the operational parameters of the training function was switched from the neural network training function operational parameters to the random tree training function operational parameters. In the operational script, the same had to happen but now the general parameters for the training function have to change from the neural network training function general parameters to the random tree training function general parameters. Lastly, the actual training function passed to the DSL training function was swapped from the neural network training function to the random tree training function. For these adaptations to happen only a couple of lines of code have to change. The number of lines of code that have to change could increase for more complex workflows but still here the modularity should still help. A benefit that comes from this modularity is that the

Dagger Algorithm

1. Initialize $D \leftarrow \emptyset$.
2. Initialize $\hat{\pi}_1$ to any policy in Π.
3. for i = 1 to N do
4. Let $\pi_i = \beta_i \pi^* + (1 - \beta_i) * \hat{\pi}_i$.
5. Sample T-step trajectories using π_i.
6. Get dataset $D_i = \{(s, \pi^*(s))\}$ of visited states by π_i
7. and actions given by expert.
8. Aggregate datasets: $D \leftarrow D \cup D_i$.
9. Train classifier $\hat{\pi}_{i+1}$ on D
end for
10. Return best $\hat{\pi}_i$ on validation.

```
Dagger(train_NN, simulate_system_traces, expert_policy, init_policy, Dagger_loops, p,
start_point_dataset, trace_length):
1.  total_dataset = DSL_Data_Set()
2.  policies_list = [init_policy]

3.  for idx in range(Dagger_loops):
4.      beta = math.pow(p, idx)
4.      loop_policy = beta*expert_policy + (1-beta)*policies_list[idx]

5.      trace_dataset = simulate_system_traces.simulate_system_traces(loop_policy,
                    start_point_dataset, trace_length)

6.      loop_dataset = DSL_Data_Set()
6.      loop_dataset.append_trace(trace_dataset)
7.      loop_dataset.output = expert_policy.give_output(loop_dataset.input)

8.      total_dataset.append_dataset(loop_dataset)

9.      trained_policy = train_NN.train_policy(total_dataset)
9.      policies_list.append(trained_policy)

10. return policies_list
```

Fig. 2. The Dagger pseudo code and the Dagger implementation with the DSL [9]

```
def Dagger(train_NN, simulate_system_traces, validate_datasets, validate_trace_datasets, expert_policy, start_dataset_csv,
start_point_dataset_csv, expert_trace_dataset_csv, Dagger_loops, trace_length, p, output_map):
    total_dataset = DSL_Data_Set()
    total_dataset.initialize_from_csv(start_dataset_csv)

    start_point_dataset = DSL_Data_Set()
    start_point_dataset.initialize_from_csv(start_point_dataset_csv)

    expert_trace_dataset = DSL_Trace_Data_Set()
    expert_trace_dataset.initialize_from_csv(expert_trace_dataset_csv)

    for idx in range(Dagger_loops):
        iteration_output_map = output_map+"\iteration_"+str(idx+1)
        os.makedirs(iteration_output_map)

        train_NN_policy_parameters = Train_NN_Policy_parameters(output_map=iteration_output_map)
        validation_parameters = Validation_parameters(output_map=iteration_output_map)
        validation_trace_parameters = Validation_trace_parameters(output_map=iteration_output_map)

        [train_dataset, test_dataset] = total_dataset.split_dataset([0.8, 0.2])
        trained_policy = train_NN.train_policy(train_dataset, train_NN_policy_parameters)

        policy_results = DSL_Data_Set()
        policy_results.input = test_dataset.input
        policy_results.output = trained_policy.give_output(test_dataset.input)

        validate_datasets.validate_datasets(test_dataset, policy_results, validation_parameters)

        beta = math.pow(p, idx)
        loop_policy = beta*expert_policy + (1-beta)*trained_policy

        trace_dataset = simulate_system_traces.simulate_system_traces(loop_policy, start_point_dataset, trace_length)

        validate_trace_datasets.validate_trace_datasets(expert_trace_dataset, trace_dataset, validation_trace_parameters)

        loop_dataset = DSL_Data_Set()
        loop_dataset.append_trace(trace_dataset)
        loop_dataset.output = expert_policy.give_output(loop_dataset.input)

        total_dataset.append_dataset(loop_dataset)
        total_dataset.write_dataset_to_csv(iteration_output_map+"\dataset")
```

Fig. 3. The complex Dagger implementation with the DSL

policies are validated in the same way because the validation function is separate from the training function making it easier to compare different policies.

The next requirement is **Readability/understandability**. For this requirement, we want to check that when the method is implemented using the DSL we still understand the method without having to go look at the paper.

To show that this requirement is met, we look at the comparison between the pseudo code and our implementation. Pseudo-code is often used to explain algorithms in literature. The pseudo-code used is not a particular programming language but uses general concepts like if-else statements and loops. These concepts are then used with general short descriptions of functions and operations to explain an algorithm. In the previous section, we see that the Dagger algorithm is implemented in a very similar way to the pseudo-code. This means that the user can use concepts with a high level of abstraction like seen in pseudo-code to construct the method and do not have to think about the details of the implementation. This makes it also easier to read and understand the code analogous to the reasons why pseudo-code is used. In figures 4 and 5 the comparison between the pseudo-code for NDI and curriculum learning with their implementation in the DSL are shown. We see that the DSL implementation is similar to the pseudo-code in these cases as well. Some extra functions had to be added like the check_out function for the NDI method because if we would just compare the output directly they would practically never match because they are floating point values. A sorting function also had to be constructed for the curriculum learning method. But thanks to our DSL being an

internal DSL implemented in Python these custom functions can be easily abstracted and the readability of the code does not suffer.

The third requirement is **Data management**. For this requirement, we want to show that the DSL can handle manipulating the data abstractly when implementing and concretely when implementing the functions.

To check if the requirement is met we refer to the examples we have shown before. When implementing the methods the user does not have to think about the details of data management and can instead think about more abstract concepts like datapoints, datasets, and trace datasets and use these concepts to construct the method. When implementing the functions the DSL data structures can be easily transformed into pandas dataframes. This allows the user to use the tools of pandas to easily manipulate the data for the implementation of their functions.

The last requirement is **Reliability**. For this requirement, we check that the DSL does not negatively affect the outcome of the training and that the policy produced is on par with policies trained from scratch.

To show this, we trained a policy using the more complex Dagger policy shown before. The validation result of this trained policy can be seen in figures 6 and 7. We see that the policy is able to keep the prediction error small for most points in the test dataset.

VII. Discussion & Future Work

In this section, we discuss things that can be done and researched in the future to improve the DSL. First, in this paper, we only look at one use case. This limits the kind of data the DSL was tested on. In the future, more testing with other data types could be done. Although we looked at three methods there are more methods that could be looked at in the future. For example, methods where multiple policies are trained for different purposes. This could show how generalizable the DSL is or if more functions should be added to cover certain needs. Lastly, at the moment there is no empirical data that our DSL helps with the efficiency of coders using the DSL. An empirical study can be done to gather this data. This empirical study could also give more insight into what features could be added or changed to improve the efficiency of the DSL.

VIII. Conclusion

In this paper, we introduced a DSL that helps with coding complex workflows that are used when training machine learning policies for model predictive controllers. This DSL was implemented as an internal DSL in Python. To show its effectiveness the DSL was used to implement three methods: Dagger, NDI and Curriculum Learning. Using the results of these implementations we showed that the DSL fulfilled the requirements that we had defined at the beginning. Lastly, we ended with describing what could be done in the future to improve the DSL.

References

[1] Vahdat Abdelzad et al. “Non-divergent Imitation for Verification of Complex Learned Controllers”. In: *2021 International Joint Conference on Neural Networks (IJCNN)*. 2021, pp. 1–8.

[2] Nils Baumann et al. “Dynamic Data Management for Continuous Retraining”. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS ’22. Montreal, Quebec, Canada: Association for Computing Machinery, 2022, pp. 359–366.

[3] Joris Gillis et al. “Effortless modeling of optimal control problems with rockit”. In: 39th Benelux Meeting on Systems and Control 2020, Elspeet, The Netherlands.

[4] Sebastian Günther. “Development of Internal Domain-Specific Languages: Design Principles and Design Patterns”. In: *Proceedings of the 18th Conference on Pattern Languages of Programs*. PLoP ’11. Portland, Oregon, USA: Association for Computing Machinery, 2011.

[5] A. Hassan and Ahmed Kassem. “SPEED CONTROL DESIGN OF A PMSM BASED ON FUNCTIONAL MODEL PREDICTIVE APPROACH”. In: *JES. Journal of Engineering Sciences* 40 (July 2012), pp. 1121–1135.

[6] Radoslav Ivanov et al. “Verisig: Verifying Safety Properties of Hybrid Systems with Neural Network Controllers”. In: *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*. HSCC ’19. Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 169–178.

[7] Shubhra Kanti Karmaker (“Santu”) et al. “AutoML to Date and Beyond: Challenges and Opportunities”. In: *ACM Comput. Surv.* 54.8 (Oct. 2021).

[8] Wes McKinney. “pandas: a Foundational Python Library for Data Analysis and Statistics”. In: 2011.

[9] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. *A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning*. 2011. arXiv: 1011.0686 [cs.LG].

[10] Max Schwenzer et al. “Review on model predictive control: an engineering perspective”. In: *The International Journal of Advanced Manufacturing Technology* 117 (2021), pp. 1327–1349.

[11] Xin Wang, Yudong Chen, and Wenwu Zhu. *A Survey on Curriculum Learning*. 2021. arXiv: 2010.13166 [cs.LG].

[12] Julian Zucker and Myraeka d’Leeuwen. “Arbiter: A Domain-Specific Language for Ethical Machine Learning”. In: *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*. AIES ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 421–425.

Non-Divergent Imitation (NDI)

1. $\pi_0 \equiv \pi^*$: the oracle policy
2. T ← ∅ /* multiset of paths */

3. for i = 1, . . . , i_{max} do
   /* initialize k new paths */
4. repeat k times

5. sample s ∼ d(·) /* initial dist. */

6. a ← π^*(s)

7. T ← T ∪ {sa}

8. T′ ← ∅ /* multiset of paths */
9. D ← ∅ /* multiset of state-acts. */

10. foreach τ = $s_1 a_1 \cdots s_t a_t$ ∈ T do
11. τ′ ← ExteND(τ,π_{i-1})
12. D ← D ∪ {$\tau'_{n:n}$|1 ≤ n ≤ |τ′|}

13. if |τ′| > t then
14. T′ ← T′ ∪ {τ′}
15. else
16. T′ ← T′ ∪ {τ}

17. π_i ← TrainVerifiablePolicy(D)

18. T ← T′
19. return SelectBest({π_i|1 ≤ i ≤ i_{max}})

```
NDI(train_NN, simulate_system, expert_policy, start_point_dataset, number_of_paths, max_iteration,
trace_length, max_diff):
1.  policy_list =[expert_policy]
2.  total_traces = DSL_Trace_Data_Set()

3.  for i in range(max_iteration):

4.      for idx in range(number_of_paths):
5.          start_point_dataset.shuffle()
5.          start_point = start_point_dataset.get_iter_datapoint_from_dataset(idx)

6.          k_start_points_dataset = DSL_Data_Set(is_trace=True)
6.          k_start_points_dataset.append_datapoint(start_point)
6.          k_start_points_dataset.output = expert_policy.give_output(k_start_points_dataset.input)

7.          total_traces.append_dataset(k_start_points_dataset)

8.      loop_traces = DSL_Trace_Data_Set()
9.      training_dataset = DSL_Data_Set()

10.     for trace in total_traces:
11.         loop_trace = ExteND(expert_policy, policy_list[i], trace, trace_length, max_diff, simulate_system)
12.         training_dataset.append_dataset(loop_trace)

13.         if loop_trace.length>trace.length:
14.             loop_traces.append_dataset(loop_trace)
15.         else:
16.             loop_traces.append_dataset(trace)

17.     policy = train_NN.train_policy(training_dataset, train_NN_policy_parameters)
17.     policy_list.append(policy)
18.     total_traces = loop_traces
19. return policy_list
```

Fig. 4. The NDI pseudo code and the NDI implementation with the DSL [1]

The Baby Step Training Scheduler
1. D′ = sort(D, C);
1. $\{D_1, D_2, \cdots, D_k\}$ = D′ where C(d_a) < C(d_b), $d_a \in D_i$, $d_b \in D_j$, ∀i < j;

2. D_{train} = ∅;
3. for s = 1 · · · k do

4. D_{train} = D_{train} ∪ D_s;

   while not converged for p epochs do
5.   train(M, D_{train});
   end while
end for

```
CL(train_NN, retrain_NN, total_dataset, k):
1. dataset_list = sort_train_dataset(total_dataset, k)


2. training_dataset = DSL_Data_Set()
3. for idx, dataset in enumerate(dataset_list):

4.      training_dataset.append_dataset(dataset)

5.      if idx == 0:
5.          policy = train_NN.train_policy(training_dataset)
5.      else:
5.          policy = retrain_NN.train_policy(training_dataset)
```

Fig. 5. The curriculum learning pseudo code and the curriculum learning implementation with the DSL [11]

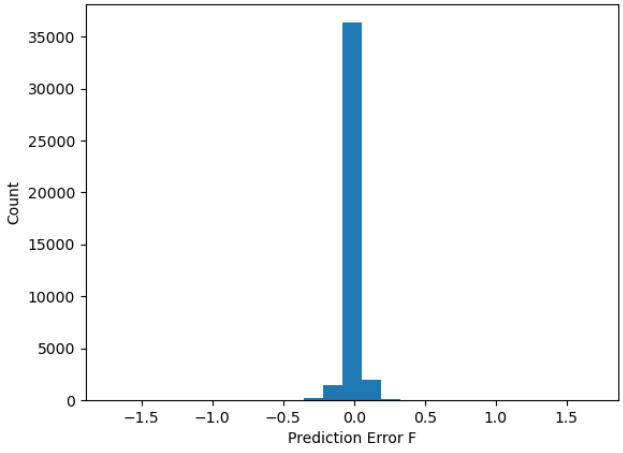

Fig. 6. Histogram of the error between predicted F (from the trained policy) and the True value of F (from the expert policy)

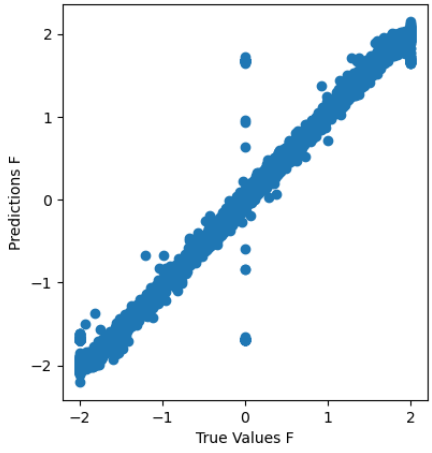

Fig. 7. True values of F (from the expert policy) vs prediction of F (from the trained policy)