Language for Test Specification of Machine Learning Models

Boqi Chen 260727855
Neeraj Katiyar 260726511

April 5, 2022

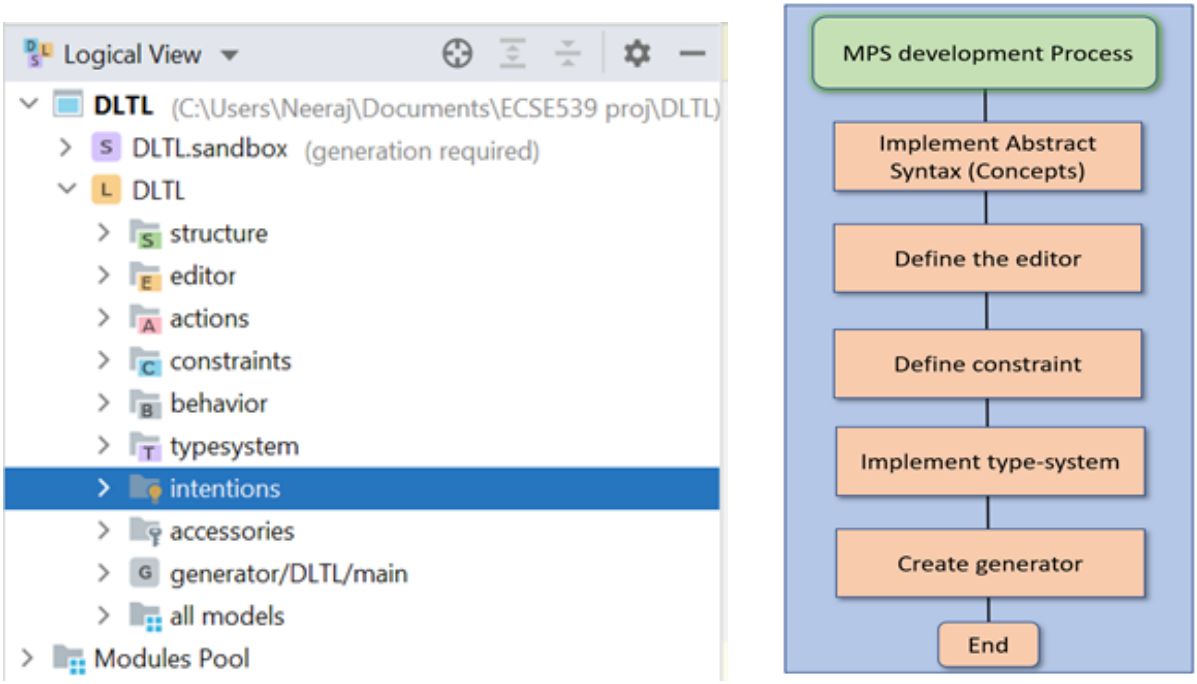
1 Introduction

1.1 MPS: The Language and Transformation Environment

Meta Programming System (MPS)[1] is a language workbench developed by JetBrains. MPS is a tool to design domain-specific language. It uses projectional editing, which allows users to overcome the limits of language parser and build domain-specific language editor. MPS supports composable language definitions. This feature helps to extend and embed a language, and these extensions can be further used in the same program in MPS. Figure 1a shows the snap of the MPS models, each of these models covers some aspect of the definition of language. We'll discuss each of them in the following section:

- The structure aspect defines the abstract syntax - the logical building blocks of a language. So basically, the concepts of the language.
- The editor aspect helps to define the notations of the language also called concrete syntax. There are multiple notations being supported by MPS, such as textual, symbolic, tabular, and graphical. For our project, we have used textual notation. The task of the MPS editor is to create the Abstract Syntax Tree in a user-friendly way and provide the means for editing it effectively. MPS uses a positional DSL to place "cells" on the canvas and bind them to the properties of the concept, which helps users define the concrete syntax for the language. MPS also helps to determine the layout in the editor.
- The Type-system aspect helps users to define checking rules and quick fixes. It is similar to adding constraints (e.g. Notice that users are not allowed to set the value of the desired accuracy to non-positive values).
- Intentions helps define suggestions on automated changes to the code, i.e. a hint to the users. At the same time, the Accessories Aspect helps to create the internal libraries of the system in the system.
- The Generator helps to combine DSL and implementation. The generator aspect of language definition in MPS is responsible for setting the meaning of code by transforming it into another target language with templates.

[1] `https://www.jetbrains.com/mps/`



(a) MPS models

(b) MPS development workflow

Figure 1: 1(a): Various models under MPS, 1(b): Development life-cycle of MPS

Figure 1b outlines the life cycle of the development for a MPS project into five major categories: implementation of Abstract Syntax, Defining the editor, Defining the constraint, implementing the type-system and the generation part. In further sections, We have discussed each step relevant to our project in detail.

1.2 Advantages of MPS

The main reason behind this technology choice is its integrated generator for model transformation, which we have used for the code generation part of our language. This integration can significantly reduce the setup and configuration time at the beginning of the project. Simultaneously, its separation of abstract (Structure) and concrete (Editor) syntax and powerful editor support will also help us develop the language. Good documentation with plenty of examples also helps to get started with the tool with minimal effort. The user only needs minimum knowledge of the theory of Syntax analysis to begin the use of the tool.

1.3 Drawbacks of MPS

Despite having a strong generator and editor part, there are a few downsides of MPS, as discussed in the following points:

- **Source control:** The version control of MPS projects are complex. Because of the projectional editor, the source files are not stored as text files but rather XML files. This file format increases the difficulty during version control, especially when merging. To alleviate this difficulty, we choose not to work on the same file at the same time.

- **Projectional editor:** The projectional editor is also complex during implementation. For example, if we want to call a method, we must use code completion to select the method from a list. However, once we choose a method, other methods with the same name but different parameters can not be used (method overloading) due to the reference-binding nature of the editor. We define the method call again, which reduces the productivity during editing.
- **Lack of Multi-Language Support:** MPS is limited to only IntelliJ ecosystem, so the generator only provide syntax analysis if the target language is Java. For all other languages, the outputs are treated as plain text.
- **Lack of Visualization:** MPS does not support central visualization of the project's meta-model and the graphical modeling languages as input.

2 Language Definition

2.1 DLTL: Deep Learning Testing Language

Motivation Domain specific language (DSL) come in handy to fluently express complex business and working assumptions, in a language that reads like English. In industry and academia, usually developers and testers spent a good amount of time in the development and testing of ML models. And sometimes they are required to repeat the same set of operations even with a slight change in the requirement from client end. This problem motivated us to develop a language for test generation of ML models which would lead to reduced turnaround time related to the development and testing of ML models.

Proposed solution As briefly shown in the figure 2, DLTL would help the clients with very little knowledge of Machine Learning (ML) concepts would be able to define model metrics and test sets with our language. This solution would then allow them to create test cases and generate a test suite in Python. On the other hand, the ML developers would be able to evaluate their AI models against provided test cases and would be able to see test results.

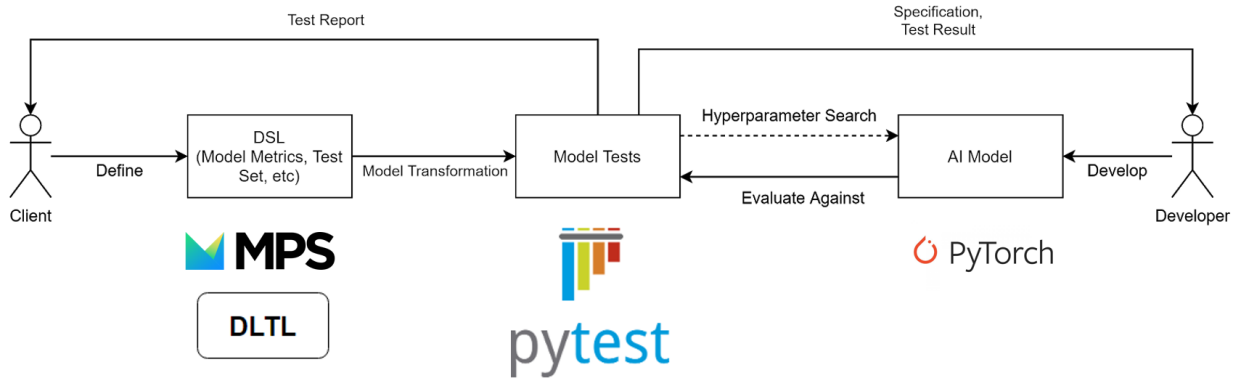


Figure 2: Solution Workflow for DLTL

Scope There are currently multiple ML frameworks existing for different programming languages. Each of them may have its API naming and set up. ML models for different kinds of tasks may require different sets of metrics apart from the other frameworks. To restrict the scope of our

project, we decide to focus on the evaluation of PyTorch models for classification tasks. But the language model would also cover the evaluation of other frameworks in other tasks providing minor modification of the code generation process.

The language we developed will be transformed as a model-to-text transformation to a set Python script consisting of the generated test cases and logic to set up the test. Ideally, the user can use the command line to directly execute the script by providing the ML model artifact and datasets path. We defined the abstract syntax to cover various metrics for measuring an ML model, supporting some logic operations to create conditional evaluations. Furthermore, since it is not possible to develop a comprehensive list of ML model metrics, we also made the language extensible to define their evaluation of the ML models. As for the concrete syntax, we used textual representation to make the development in our language similar to using simple descriptive languages.

2.2 Editor Example

MPS provides an internal editor for the language defined with the system. One only needs to provide a set of editor layout rules to define the language's concrete syntax. To start the editor of DLTL, one needs to load the DLTL language into MPS and create a **Solution**. One can also use the provided example sample solution in the accompanying package with the report.

As MPS and our language features are very complex, it is difficult to use one example that shows all the features of our language. This section will first provide a basic example model that covers many aspects of DLTL, followed by some specific features that are not covered by the example.

A Basic Example Figure 3 shows the example editors for datasets, models and metrics. Notice that these are three separated editors, but they are combined into one for illustration purposes. Words in bold are text constant that the user cannot modify, while words in blue are the keywords of DLTL. For datasets and models, the user needs to specify the name, the path to the method that provides the dataset/model, and some parameters. The parameters are checked with a regular expression to ensure that its value matches its type (e.g. int variables can only contain numerical values). For metrics, the user can define the number of models, the number of datasets, a list of inputs the metric needs to take and the output type. The output type is used later as type checks in the test cases. Then, the user can provide the implementation for the metric. Currently, there is only a check to make sure that the name of the implementation method must match the name of the metric. We defer other static checks in the implementation as future works.

Figure 4 shows the editor for defining test suite. The user first specifies the name and a set of imported packages for the test suite. Then, the user can define the test cases as logic expressions with the datasets, metrics and models defined previously. As an illustrative example, the first test case shows the features of the logic part of DLTL. It can be interpreted as $(1+2=3) \implies (5+1=4)$. The test cases that follow are similar but used metric references other than numerical constants. For the set of supported operations in the logic language definition, please check the metamodel in Figure 6.

Other Features The editor for DLTL also contains other features that help the user build the model. We summarize these features as follows.

```
Dataset: CIFAR10 pagkage dataset getter cifar10 Model: Resnet18 pagkage resnet getter resnet18
  boolean train = False                                  << ... >>

Metric Time_Second
  number of models: 1
  datasets: 1
  Inputs: device : string
  Output: float

  Implementation:
  def time_second(model, test_set, device):
      model.eval()
      model.to(device)
      batch_size = 32
      test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size, shuffle=False)
      start = torch.cuda.Event(enable_timing=True)
      end = torch.cuda.Event(enable_timing=True)

      start.record()
      with torch.no_grad():
          for images, labels in test_loader:
              images = images.to(device)
              labels = labels.to(device)
              outputs = model(images)
      end.record()
      torch.cuda.synchronize()

      return start.elapsed_time(end) / 1000
```

Figure 3: Example Editor for Models, Datasets and Metrics

```
Test: demo
Imports:
import torch
TestCases:
  demo test
    1 plus 2 equal 3  implies 5 minus 1 equal 4

  accuracy test
    The Accuracy value of models VGG11 with inputs device = cuda:0 on dataset CIFAR10  larger than or equal 0.9

  accuracy test impossible
    The Accuracy value of models VGG11 with inputs device = cuda:0 on dataset CIFAR10  larger than 1.0

  precision recall treadoff
    The Precision value of models VGG11 with inputs label = 1, device = cuda:0 on dataset CIFAR10  larger than or equal
        0.95  or The Recall value of models VGG11 with inputs label = 1, device = cuda:0 on dataset CIFAR10
        larger than or equal 0.95

  model comparason
    The Accuracy value of models VGG11 with inputs device = cuda:0 on dataset CIFAR10  larger than
        The Accuracy value of models Resnet18 with inputs device = cuda:0 on dataset CIFAR10  minus 0.05  and
        The Time_Second value of models VGG11 with inputs device = cuda:0 on dataset CIFAR10  smaller than
        The Time_Second value of models Resnet18 with inputs device = cuda:0 on dataset CIFAR10  minus 0.5
```

Figure 4: Example Editor for Test Suites

Intentions are a set of hints provided to the user at edit time so that the user can use them to quickly develop the model (activated with *alt + enter*). There are currently two intentions in the language; the first one helps with copy/paste plain text for implementing the metric (Normal copy/past for plain text does not work due to MPS' projectional editor). The second one is in the metric reference, where the user can use the hint to infer a list of needed parameters to execute the metric.

The **type-check system** of MPS is capable of expressing complex constraints that are both type-related or non-type-related. Using the type check system, we created a list of type inference rules for the logic expressions to check for their correctness. For example, the left side and right side of any binary comparator should have the same type, and the evaluation result of a test case should be a boolean type. Similarly, we also created standard constraint checking rules such as the value of a variable must obey its type.

Source Files Overall, these files we manually developed for the editor features are located under the editor, intentions and type system folder of the language project in MPS. A set of accessories are also defined for basic machine learning metrics so that the user does not need to define them manually. Last, the example models are stored in the DLTL.sandbox project. All corresponding directories are shown in Figure 1a. Notice MPS does not store the source code as plain text but rather XML files. One needs to install MPS to check the formatted source code.

2.3 Metamodel

In this section, we demonstrate the metamodel representing the abstract syntax of DLTL. As shown in Figure 5 and Figure 6, the metamodel of DLTL is divided into two parts. The first metamodel mainly represents concepts necessary for creating a test suite for the machine learning models (shown in white color). The second part is mainly about creating specific test cases with logic expressions (in red color). Two other types of concepts are rootable concepts, which can be directly instantiated by the user from the editor and contains other parts of concepts, and MPS internal concepts, which are the concepts defined in the abstract syntax of MPS' base language. We use these internal concepts to simplify the abstract syntax of our language.

MPS does not come with default visualization for the language metamodel; thus, we manually drew the two figures representing the metamodel. In order to keep the metamodel readable, we simplified several parts of the metamodel drawing. First, the multiplicity of the composition end is omitted, as all the compositions are optional. This option is necessary as the composition relationship in MPS is directional (parent to children) and there may be multiple composition relationships for one class (e.g. an instance of *Parameter* class may be either contained in *Dataset* class or *Model* class). Second, all relationship names are omitted, except when there are multiple relationships between two same classes (e.g. Expression and Binary Expression). Last, the reference classes (*xxRef*) are necessary due to the inherent constraints of MPS that associations between two classes can only have multiplicity up to 1. To support more than one multiplicity, we use the reference classes as a workaround (composition can have many multiplicity).

Starting from the test part metamodel (Figure 5), three concepts are rootable: *TestWorkbench*, *Dataset*, *Model* and *Metric*. *TestWorkbench* contains a list of test cases as *Predicate*, which is connected with *Expression* in the logic part of the language. *Dataset*, *Model* and *Metric* are defined as root concepts so that the user can potentially use the same datasets, models and metrics in multiple test suites to increase reusability. We applied abstraction occurrence pattern for the *Metric*, while *Metric* is the definition of a metric and *MetricRef* is the specific call to a metric. We

integrate *MetricRef* with the logic part by make it as a subconcept of *SingleValue*. This inheritance allows user to use metric methods in their test case definition.

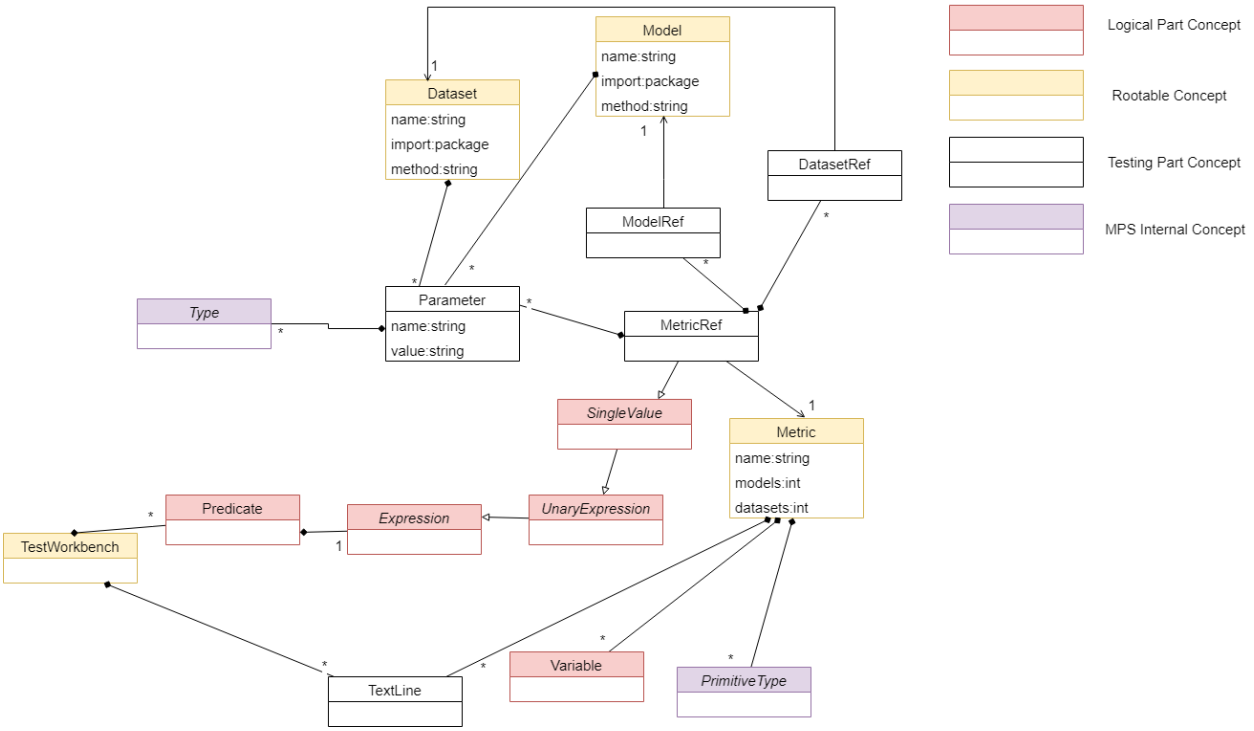


Figure 5: Metamodel for Test Part

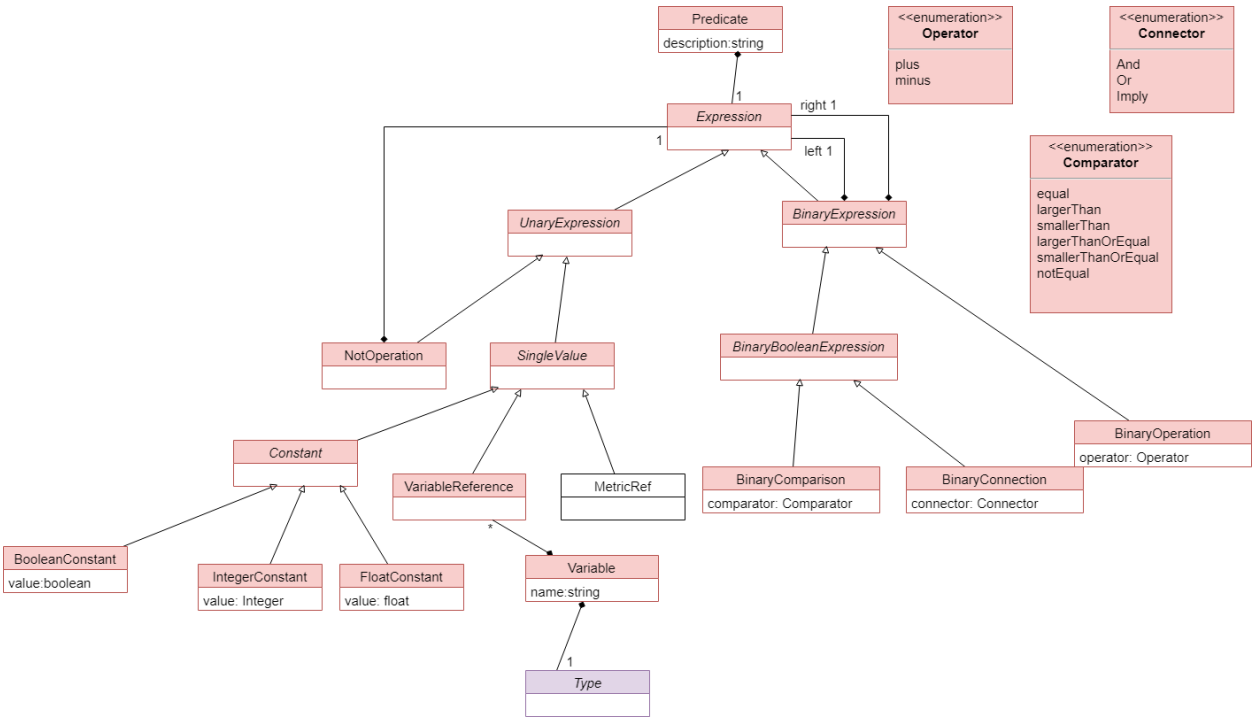


Figure 6: Metamodel for Logic Part

The main reason to define a logic part (Figure 6) in the abstract syntax is to support diverse logic expressions in the test cases. As a basic example, a user can use an instance of *Metric* to evaluate the machine learning model with the desired dataset. A metric for a machine learning model normally returns a numerical value. However, the result of a test case is normally binary (True/False). Thus, one must support at least the comparison operation for metric evaluation results and some numerical constant (i.e. *BinaryComparason* and *IntegerConstant*). Starting with this intuition, we gradually added more operations and built an expressive logic language for defining test cases.

3 Transformation

3.1 Target Language and Frameworks

To facilitate the test environment, we used Pytest[2] as the chosen test framework for the target language Python. The following section discusses the reasons for choosing Pytest over other frameworks.

Flexibility Because of the scope-based nature of Pytest fixtures, it is easy to define how the tests should be executed. We can create custom fixtures so that they can be easily re-used by each new test package, module or method we add. We take this reusability advantage extensively during the generation. Once we have defined the scope of a fixture, a module does not need a specific setup and tear-down, which are normally required for other testing libraries.

Fixtures Parametrization of fixtures is another feature that helps us send parameters used during fixture execution – this is helpful for avoiding the boilerplate and assert as many scenarios as possible by using only a single test method. We also use parameterized fixtures extensively in our generated test files.

Test discovery and logging The test case suite (test cases) can be run by using a single command and by defining a test module, method or package of modules to run. Pytest test discovery is smart – it will run everything that is not marked to be skipped and test prefixed methods, classes, modules. This feature makes it easy to define a suite of tests by specifying just a package of test modules to run. One can also select a single test method to run by using the proper flags for the purpose. At the same time, the test report is also informative.

We have also leveraged the PyTorch[3] library to define the metrics and the ML models. Pytorch is a python based scientific computing package. It provides native support for Python, and it is widely used in many deep learning projects had made an obvious choice for us to use it. Pytorch has some powerful and inbuild functions. One such example is: **torch.utils.data.DataLoader** as shown in the figure 7. This method combines a dataset and a sampler and provides iterable access over the given dataset. It also supports both map-style and iterable-style datasets with single- or multi-process loading, customizing loading order and optional automatic batching (collation) and memory pinning.

[2] `https://docs.pytest.org/en/6.2.x/`
[3] https://pytorch.org/

```
Implementation:
def accuracy(model, testset):
  batch_size = 50
  test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size, shuffle=False)
  with torch.no_grad():
    correct = 0
    total = 0
```

Figure 7: Use of pytorch inbuilt method

3.2 Mapping between DLTL and Pytest

DLTL Concepts	Python Concepts
Model	Fixture Method, Command Line Argument
Dataset	Fixture Method, Command Line Argument
Metric	Python Method (using Implementation)
TestWorkbench	Python file containing test cases
Predicate	Test Case Method with Assertion
Expression	Python Expresison

Table 1: Mapping from main concepts to Python

Transformation for Main Concepts Table 1 shows the mapping from a subset of concepts to python concepts. The mapping of most other concepts is context-dependent (depends on its parent); thus, creating a mapping illustration for them is difficult. Rather, we focus on giving an overview of the mapping of the main concepts and mentioning the mapping of other elements simultaneously. The generation starts with **TestWorkbench**, and it traverses all the test case expressions to find a set of unique referenced sets of metrics, models and datasets. This step is needed because these three concepts are rootable, and there may be multiple references from the test workbench to one instance of the three concepts. However, we only want to generate one target for each instance. Then, each of the instance from the main concepts is mapped to python concepts as shown below:

- **Model**: models are mapped to a Pytest fixture method which contains a method call to another user-defined method that provides the model. The **import** and *getter* properties are used to import the specific package and method, while the *parameters* are used as input to the method call. Additionally, a command line argument is added to the generated Python file to take the path to the model artifacts.
- **Dataset**: The mapping to dataset is similar to the one for model. All transformed models and datasets are in the same python file.
- **Metric**: The transformation of the metric is, in fact, well-defined by the user when they provide the *implementation* property. Thus, a metric is transformed to a Python method by copying the *implementation* property. All transformed metrics are grouped in the same file.
- **TestWorkBench**: A test workbench is transformed to a python file containing individual test cases. At the beginning of this file, all the *import* copied, along with the import of metrics transformed before.

- **Predicate**: A predicate is transformed to a Pytest test method with the corresponding dataset and model fixture it needs. The method contains a transformation of the expression that the predicate contains as an assertion. Furthermore, method signatures are defined with the same name as the transformed fixture methods for the referenced datasets and models.
- **Expression**: An expression is transformed to the corresponding python expression. A detailed description is discussed next.

DLTL Expression	Python Expression
A plus/minus B (BinaryOperation)	A +/- B
A and/or B (BinaryConnection)	A and/or B
A imply B (BinaryConnectoin)	(not A) or B
A equal / not equal B (BinaryComparison)	A =/!= B
A larger than / smaller than (BinaryComparison)	A >/ <B
A larger than / smaller than or equal B (BinaryComparison)	A >= / <= B
not A	not A
Int / Boolean / Float Constant	Boolean / Number
MetricRef	Method Call to Metric Method

Table 2: Mapping from DLTL Expression to Python Expression

Transformation for Expression The transformation for expressions is recursive as shown in Table 2. The last two rows in italic are the terminal expressions. A and B represent the nested expressions. Most of the transformations keep the original expression structure except implication. The implication symbol is used as a syntactic sugar whose semantic is just "not A and B." MetricRef is transformed into a method called the metric method with the corresponding models, datasets, and parameters.

3.3 Generate Example

To run the generation, one needs to go to the *Solution Project* or sandbox project, right-click and find *Rebuild Project*. The generated file will be put into a user-define directory that can be specified in the project setting. One can also preview the generation result by open the editor for the *TestWorkbench*, right-click and choose *Preview Generated Text*. Notice that the generation needs the plaintextgen [4] plugin of MPS to be installed to execute.

Source Code We developed the transformation logic code as in the Generator folder. To help simplify generation logic, we also developed a set of helper methods for each concept in the behaviour folder. The example model is the same model as shown in Figure 4. All corresponding directories are shown in Figure 1a. The sample generated code can be found in the *code/DLTL/examples/demo/DLTL/sandbox* folder in the accompanying package. To see how the generated code can be executed, check the *examples/demoProject* folder. *run.txt* contains the command to execute the code from the test (Notice CUDA, PyTorch and Pytest are needed to execute the code).

[4] `https://plugins.jetbrains.com/plugin/8444-com-dslfoundry-plaintextgen`

4 Conclusion Future Work

In this project, we implemented a domain-specific language focusing on the test specification for machine learning models. DLTL has a complete pipeline from test specification and test execution. The source project can be found on GitHub[5] as well as in the accompanying package. Future work includes adding more static validations to the language, add training specification support and memorization for metric execution results.

5 Contribution

The authors contributed equally throughout the development of the project. Primarily, Boqi Chen was responsible for defining the abstract syntax, concrete syntax and part of generation logic. Neeraj Katiyar was responsible for developing the internal metrics and the target model. The authors contributed equally to the presentation, writing and editing of the report.

[5] `https://github.com/n-katiyar/DLTL`