Code Generation for Machine Learning using Model-Driven Engineering and SysML

1st Simon Raedler ⬤
Department of Computer Science
TUM School of Computation, Information and Technology
Technical University of Munich, Germany
simon.raedler@tum.de

2nd Matthias Rupp
Department of Computer Science
University of Applied Sciences Vorarlberg
Dornbirn, Austria

3rd Eugen Rigger
Department of Digital Engineering
Zumtobel Lighting GmbH
Dornbirn, Austria

4th Stefanie Rinderle-Ma ⬤
Department of Computer Science
TUM School of Computation, Information and Technology
Technical University of Munich, Germany
stefanie.rinderle-ma@tum.de

Abstract—Data-driven engineering refers to systematic data collection and processing using machine learning to improve engineering systems. Currently, the implementation of data-driven engineering relies on fundamental data science and software engineering skills. At the same time, model-based engineering is gaining relevance for the engineering of complex systems. In previous work, a model-based engineering approach integrating the formalization of machine learning tasks using the general-purpose modeling language SysML is presented. However, formalized machine learning tasks still require the implementation in a specialized programming languages like Python. Therefore, this work aims to facilitate the implementation of data-driven engineering in practice by extending the previous work of formalizing machine learning tasks by integrating model transformation to generate executable code. The method focuses on the modifiability and maintainability of the model transformation so that extensions and changes to the code generation can be integrated without requiring modifications to the code generator. The presented method is evaluated for feasibility in a case study to predict weather forecasts. Based thereon, quality attributes of model transformations are assessed and discussed. Results demonstrate the flexibility and the simplicity of the method reducing efforts for implementation. Further, the work builds a theoretical basis for standardizing data-driven engineering implementation in practice.

Index Terms—Model-Driven Engineering, Machine Learning, Model Transformation, SysML

I. INTRODUCTION

The attractiveness of machine learning and data mining in engineering has been increasing for years, as seen in the number of publications on machine learning and data mining [1]. In technical product development, the application of machine learning for making informed decisions is called data-driven engineering [2]. The complexity of technical product development is increasing due to the number of components, functions, and interactions of systems. This in turn leads to an increasing need for Model-Based (Systems) Engineering

(MBE) techniques, which are promising to manage the complexity due to different system modeling methods proven in practice [3]–[5]. However, MBE techniques focus on formalizing knowledge rather than processing data to generate valuable insights. Therefore, efforts are required to integrate data-driven engineering into technical product development and support to make informed decisions. Consequently, the formalized integration of data-driven engineering or machine learning into MBE approaches is necessary. The authors previous work introduced a model-based formalization of machine learning tasks based on the systems modeling language SysML [6]. Although this approach supports the formalization of machine learning tasks using SysML, a gap exists between the formalized knowledge within the SysML model and the actual implementation in dedicated programming languages such as Python. In this respect, this work aims to introduce a method for the automatic generation of machine learning code to reduce the duplication of effort for formalizing and implementing machine learning tasks and to extend the model-based approach to a model-driven approach. Consequently, the following research questions are elaborated in this work:

RQ1 Which model properties can be used in the context of model-driven engineering to automatically derive a machine learning model?

RQ2 What means of software engineering allows to extend and maintain the machine learning code derivation without changes in the model transformation?

As a result, this work presents a method that facilitates the implementation of machine learning code by deriving the formalization of machine learning tasks in SysML using a mapping mechanism that completes code snippets with properties and contexts from SysML and introduced stereotypes. From a more general point of view, this work contributes by improving the efficiency and effectiveness [7] of the development of machine learning in the context of systems engineering. The method is implemented and assessed for feasibility based on

a use case involving an online dataset for weather forecasting including sensor data. The source of the implementation and evaluation is available online. Furthermore, an evaluation and justification with regard to the quality characteristics of the model transformation [8] is conducted. The reminder of the paper is as follows. First, relevant background on MBE and the previously introduced approach to model machine learning concerns using SysML is depicted. Second, a method is introduced, allowing to derive machine learning code using model transformation techniques. Next, an evaluation based on an open dataset is presented. Finally, the results are discussed, future work is highlighted, and a conclusion is presented.

II. BACKGROUND

In the following, relevant background concerning Model-Based Engineering (MBE), SysML, and a basic understanding of the machine learning modeling method published in [6] are presented. Additionally, related work is discussed.

A. Model-Based Engineering & Model Transformation

The core of MBE includes the pillar concepts of models, metamodels, and model transformation [9]. Depending on the application domain, the involved engineering concepts, e.g., software or hardware, and the degree of automation, various acronyms are typically used for the concepts of MBE[1]. Model transformation can be characterized as the mapping between one or multiple input and output models. The mapping itself is defined on metamodels and not on the actual instances of a metamodel (model) to allow for reuse and generality. Model transformation aims to achieve the highest degree of automation by mapping artifacts [9]. The transformation can either be programmed manually using any programming language or using appropriate languages provided by the model-driven software engineering domain, e.g., ATL[2], Epsilon[2], etc. Model transformations can be classified as model-to-model or model-to-text transformations, depending on whether the transformation output is a model or text/code [9].

B. Machine Learning Task Formalization using SysML

SysML is a general-purpose modeling language allowing to describe a system of interest with machine-readable artifacts. In previous work, we introduced the concept of machine learning task definition based on an extension of the SysML metamodel using stereotypes and the definition of semantics to interpret the model [6]. A stereotype is a concept that allows the semantics of a metamodel to be extended to include specific properties suitable for a particular purpose and to which any block applying the stereotype must conform. A block in SysML represents a specific system, abstraction of a part of a system, or components, among others. The particular purpose of a block in the preliminary work is the representation of a specific task or subtask of a machine learning definition, e.g., data preprocessing or more specific,
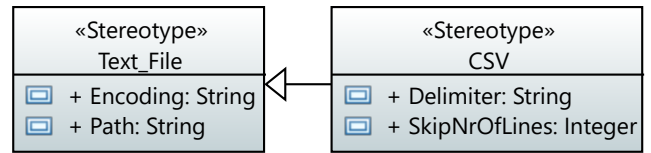

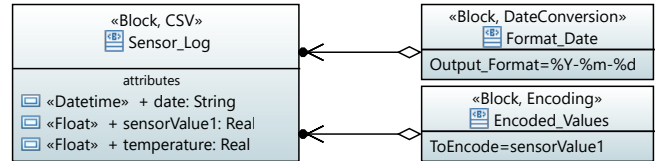Fig. 1. Sample stereotype defining a text and CSV data source.

Fig. 2. Sample application of stereotypes and semantic integration.

datetime conversion. In this respect, each stereotype abstracts a specific function or set of functions applied to a specific input value, e.g., the loading of a CSV file is abstracted using a stereotype defined in Figure 1. The properties of a stereotype are the mandatory parameters of the abstracted function and have to be defined. Additionally, properties of a stereotype can be inherited, allowing to define specific attributes only once, e.g., *Path* attribute is valid for a text and CSV file.

Figure 2 depicts the application of the defined stereotypes. The block association indicates that a specific block that is *part of* another block can be interpreted as input. Date conversion, for example, is applied to the *Sensor_Log* data source of type *CSV* with a defined *Output_Format*. Since only the *date* attribute is suitable for the date conversion, no further details are required. Still, if further details are necessary, the modeling can be extended by adding additional attributes, e.g., selecting the correct input value, etc.

The modeling of the specific machine learning tasks is based on block definition diagrams, which are a means of structural modeling. To specify an execution order for a set of functions abstracted behind a block, behavior diagrams, especially state diagrams, are used. More precisely, each state of a state diagram is connected to a previously-defined machine learning block. The connection is established using a custom stereotype. With the connection of blocks to the state diagram, the execution order of the method is defined: For this, each task is specified with a sequential execution order, allowing implementation or deriving machine learning code.

C. Related Work and Research Gaps

The concept of model-driven software engineering with a special focus on machine learning concerns can be found in literature [10]–[12].

In [10], an extension of the CPS modeling framework ThingML [13] called ThingML+ is proposed. The extension ThingML+ allows to model machine learning artifacts using a textual domain-specific language. The extension focuses on

---

[1] See https://modeling-languages.com/clarifying-concepts-mbe-vs-mde-vs-mdd-vs-mda/ for a discussion.

[2] https://www.eclipse.org/atl/

modeling supervised machine learning, and the Xtext-based transformation generates both Java and Python code. Instead of a standardized general-purpose modeling language like SysML, a custom domain-specific modeling language is used. Customization and extension of the code generation or adding additional machine learning algorithms require extending the source code by adding specific algorithms.

In [11], a platform supporting the integration of machine learning in a cloud application by experts called "Stratum" is proposed. The domain-specific modeling language allows the modeling of machine learning pipelines and models. The models and functions can be enriched with parameters, such as hyper-parameters for the learning method. Various machine learning frameworks are integrated and code can be generated. A graphical modeling interface is available by using WebGME[3] as a base. The extension and customization of the code generation require code extensions. A shortcoming of the method is the stiffness of the code snippets for the code generation, making it hard to use the generator for approaches other than the proposed case study.

In [12], textual modeling is used to describe neural networks. The approach mainly focuses on artificial neural networks. We have observed that a considerable amount of work is required to add more functions. Additionally, the integration of various data sources is limited.

Apart from related scientific work, frameworks such as KNIME[4] or RapidMiner[5] can be considered as related approaches. However, these approaches solely rely on machine learning concerns while their embedding into product development, such as MBE approaches, is not provided. Furthermore, the higher degree of freedom to formalize and document specific machine learning concerns requires less rigidity in terms of extensibility and adaptability.

Summarizing the analysis of existing literature, machine learning code generation based on model-driven methods is under development and state of the art [14]. The actual approaches mainly rely on custom domain-specific languages that define machine learning tasks using models. However, the given approaches are stiff regarding extensions due to the encapsulation of the machine learning algorithms in the source code of the code generation. Additionally, the integration of knowledge from intersecting domains is not given, making it hard to synchronize changes or to transfer knowledge. Last but not least, the approaches often propose new modelling languages, which in turn might limit users to the already scarce Data Scientists [15], instead of extending the modelling languages used in other fields. Accordingly, using a modeling language that is already known may make sense.

III. METHOD

In preliminary work, a method to describe all relevant information for implementing a machine learning approach using SysML is defined [6]. Particularly, the model represents all information concerning the composition of various relevant systems, their related data collection and the formalization of relevant data transformation and machine learning-related tasks on a single step (subtask) level. Additionally, the execution order of the machine learning tasks in the implementation is formalized using state diagrams. Each state of the diagram describes a set of sub-activities, e.g., a sequence of python functions with a dedicated purpose, such as the transformation of *Datetime* into another format.

To enable the decomposition of the defined SysML model, the here presented method relies on templates, defined as code snippets in a dedicated programming language, such as Python, and a mapping configuration that allows to identify a template based on a stereotype. The purpose of the template-based approach is to enable extendability and maintainability without the necessity to make changes in the model transformation. Additionally, an exchange of the template can be used to derive code within another programming language, such as JAVA or R. Figure 3 depicts the generic method to generate machine learning code based on templates in a flow-diagram aligned workflow, aligned with a sample model transformation depicted as images on top of the figure. The transformation applies the following subsequent steps:

1) A state diagram is provided as input, referencing each machine learning subtask formalized using stereotypes and blocks
2) For each of the states, which are provided in ascending order, the machine learning blocks are identified.
3) Based on the unique stereotype name, a template is selected.
4) Stereotype and block attributes ① are mapped to the template ③ using a mapping configuration ② to generate a code snippet ④ (see Figure 3
5) A file is generated representing the executable code snippets in the correct execution order. In the actual prototype implementation, a Jupyter Notebook is generated.

The model transformation in Figure 3 is slightly simplified and omits the step of an intermediate transformation. In the following, this intermediate transformation is introduced in Section III-A. Next, the composition of the templates with placeholders is introduced. Finally, the mapping configuration is introduced, focusing on model commands.

A. Intermediate Model

The purpose of the intermediate model is to extract information from the SysML model and to merge the state diagram information with the linked blocks. The source metamodel is a SysML model, and the target metamodel is a custom one, referred to as "block context" in the following. The block context consists of the following parts:

First, a reference to the original block in the SysML model to allow change tracking and to potentially enable synchronizing changes in the generated code with the original model.

Second, a list of rich-text blocks that can be rendered as text before a code block, modeled as so-called owned

```
«Block, CSV»
    CSV_1                                        ①
Delimiter=,
SkipNrOfLines=0
GenerateTimestamp=false
Encoding=
Path=absolute\path\file.csv
Online_Accessable=false
                 attributes
 «Datetime» + date: String
 «Float»  + wind: Real
 «Float»  + temp_min: Real
 «Float»  + temp_max: Real
 «Float»  + precipitation: Real
```

```
 1  {                                                    ②
 2    "trimEmptyLines": true,
 3    "BlackBox_Storage": {
 4       "template": "blackbox_storage.vm",
 5       "properties": {
 6          "VariableName": "varname"
 7       },
 8       "modelCommands": {}
 9    },
10    "Text_File": {
11       "template": "text_file.vm",
12       "properties": {
13          "Path": "path",
14          "Encoding": "enc"
15       },
16       "modelCommands": {}
17    },
18    "CSV": {
19       "template": "csv_load.vm",
20       "properties": {
21          "Delimiter": "delim",
22          "SkipNrOfLines": "skip",
23          "GenerateTimestamp": "GenerateTimestamp"
24       },
25       "modelCommands": {"THIS.BLOCK.NAME": "varname"}
26    }
27  }
```

```
 1  import pandas as pd                                   ③
 2  ${varname} = pd.read_csv("${path}",
 3  sep="${delim}", encoding="${(enc,"UTF-8")}",
 4  skiprows=${skip}, parse_dates=${date}, ${**
    kwargs})
```

```
 1  import pandas as pd                                   ④
 2  CSV_1 = pd.read_csv("absolute/path/file.csv",
 3                      sep=",", encoding="UTF-8",
 4                      skiprows=0, parse_dates=["date"])
```

Select State Diagram → Identify Machine Learning Block → Read Machine Learning Properties → Select Template → Generate Code → Another Block? → Print File
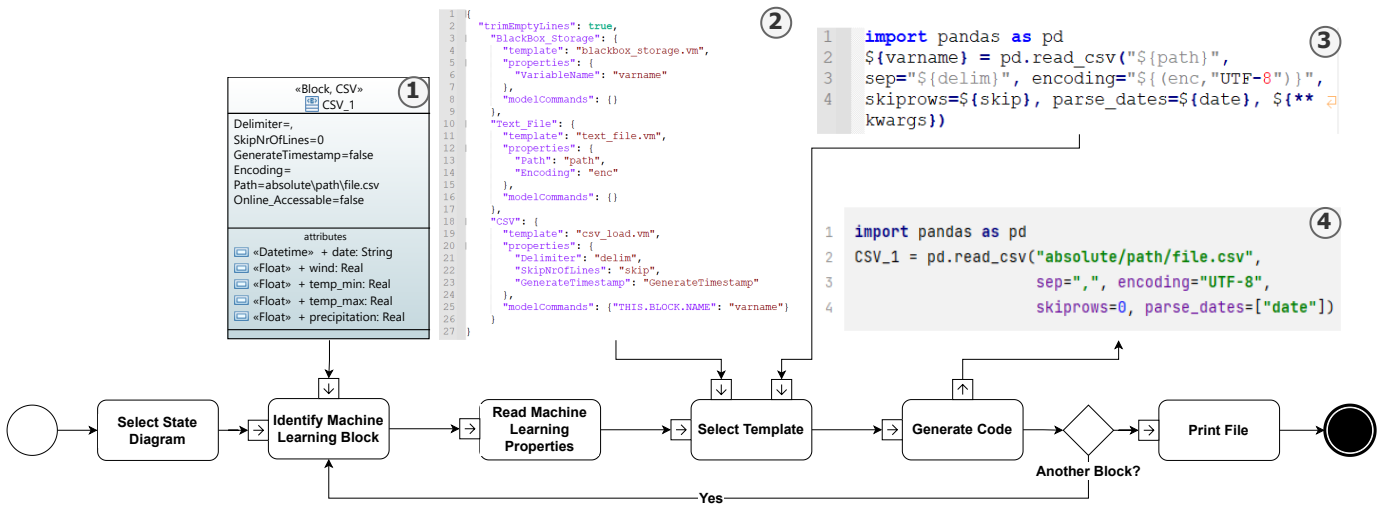
Yes

Fig. 3. A sample model transformation to load a CSV File.

comments in the SysML model. Note that rich-text annotations are represented as text block cells in the actual implementation. A dedicated format for Jupyter Notebooks and must be considered separately for other environments or programming languages.

Third, references to connected block contexts based on the qualified name, a unique identifier for named SysML elements. Due to the uniqueness of the qualified name, it can be used as an identifier for attributes or blocks.

Fourth, a list of block and stereotype attributes with their values. If a value is a primitive type, the value is used. Otherwise, the qualified name is stored and translated to a value when assigned.

Finally, an integer represents the execution order in the state diagram. The transformation is executed for each block connected to a state and for each block connected to such a block. Care is taken to prevent the multiple execution of the transformation for the same block more than once by tracing the unique identifiers of a block.

B. Code Snippet Template Definition

The templates defining code snippets are defined in textual editors. Particularly, a template consists of formatted plain-text with various placeholders filled with property values from the stereotypes during the code generation. The marker ③ in Figure 3 depicts a sample of a template with all possible types of variables, which are:

1) Standard variables are highlighted with *${variable name}*. In this case, the attribute is mandatory and has to be set in the model.
2) An optional variable that alternatively is set with a default value, indicated with a default value in the variable definition *${(variable name, default value)}*.
3) Arbitrary other attributes.

Since a function in a code snippet can have countless attributes, not all attributes can be defined in a stereotype,

and it would not make sense due to the complexity for the user. Therefore, additional properties can be added to a block instance without being defined in the stereotype. These additional properties are added to a specific position in the template indicated by an anchor-indicator ***kwargs*. For an additional property to be used for the template function, the name of the additional property must be similar to the parameter name of the corresponding programming language function, but with two asterisks after it, e.g., if a parameter of a chart printing function in Python calls *X-Axis Name*, the attribute in the block must be named ***X-Axis Name*. The double-stared unforeseen attributes are rendered to the template in the following format $attribute\_name = attribute\_value$ without the double-stars. If a template requires two ***kwargs*, the transformation must be adapted, or two sub-stereotypes must be used.

C. Mapping Configuration

A mapping configuration in ③ in Figure 3 illustrates the content of a mapping between a stereotype and a code snippet template using JSON file format. The definition of JSON mapping is depicted in Listing 1.

The mapping configuration is defined as follows:

First, the mapping allows defining whether empty lines shall be trimmed during the generation of the Jupyter Notebook (Line 2 in Listing 1).

Second, the definition of constant values allows reusing specific strings as static text, e.g., as a global variable for all templates (Line 3-6 in Listing 1).

The stereotype mapping (Line 7-18 in Listing 1) allows specifying which template to use for a stereotype. The stereotype mapping (Line 10-13 in Listing 1) defines the mapping of stereotype properties to template variables.

A command can be defined (Line 14-17 in Listing 1) and mapped to a variable by using the following keywords to collect information:

1) **THIS**: the information can be found on the block with the stereotype

```json
1  {
2    "trimEmptyLines": <true||false>,
3    "constants": {
4      "<TemplateVariableName>": "<
       ↪ ConstantValue>",
5      ...
6    },
7    "stereotypeMappings": {
8      "<StereotypeName>": {
9        "template": "<TemplateName>",
10       "properties": {
11         "<stereotypeAttributeName>": "<
         ↪ TemplateVariableName>",
12         ...
13       },
14       "modelCommands": {
15         "<ModelCommandKeywordCombination>
         ↪ ": "<TemplateVariableName>",
16         ...
17       }
18     },
19   "nameMappings": {
20     "<BlockName>": {
21       "template": "<TemplateName>",
22       "properties": {
23         "<
         ↪ PropertyOrStereotypeAttributeName>"
         ↪ : "<TemplateVariableName>",
24         ...
25       },
26       "modelCommands": {
27         "<ModelCommandKeywordCombination>
         ↪ ": "<TemplateVariableName>",
28         ...
29       }
30     }
31   }
32 }
```

Listing 1. JSON Mapping Structure

7) **STEREOTYPEofATTRIBUTE["AttributeName"]**:
the information is stored in a data stereotype of an attribute, e.g. *Datetime* stereotype of the *date* attribute of the *Sensor_Log* block in Fig. 2
8) **OUTPUT**: the information is the last declared variable name of the template, which refers to the block specified by the preceding keywords

The command's syntax consists of at least three keywords, separated by a period. The first keyword is either *THIS* or *CONNECTED* with a selector to choose the correct connected block. The second keyword is either *BLOCK* if the information is directly stored on the block or *STEREOTYPE* with a parameter specified for the stereotype name if it does not belong to the block itself. The third parameter is depicted in the enumeration list of keywords above with the item numbers 5-8. After the last keyword, it is always possible to select a value if the result is a list using square selector $[Nr.]$. After the *ATTRIBUTES* and *STEREOTYPEofATTRIBUTE*, optionally *ATTRIBUTES* or *STEREOTYPEofATTRIBUTE* can be defined again to dig deeper into specific information. The *OUTPUT* value is one of the essential values to connect a code block with the result of a previous one.

If a specific mapping is only applied to a specific block, name mapping can be used (Line 19-31 in Listing 1). Name mapping is similar to stereotype mapping, but it specifies the input model block via the block name instead of the stereotype name. The only difference is that properties can also be defined on the block without being defined on the stereotype. Name mappings take precedence over stereotype mappings if both apply for a block.

D. Composition of Code Snippets

Based on the generated code snippets and the defined execution order of the snippets, an executable file can be generated. The method presented in this work is implemented for Jupyter Notebook. For this, the following steps for composition are conducted:

1) Rich-text information modeled as owned or applied comment is directly converted to a Jupyter rich-text cell.
2) The generated templates are put in a source-code cell. Each block context (intermediate model) from the state machine gets one source code cell and, optionally, one rich-text cell.

The code snippets are analyzed for *"from ... import ..."* or *"import ..."* lines of code to increase the readability and reduce potential errors due to multiple inputs of modules required. These lines are cut out and inserted in the first code cell on top of the Jupyter Notebook file.

After all block contexts are iterated over, the cells are put together as a single file, leading to an executable Jupyter Notebook file. Finally, the syntax is validated, so the execution is ensured. If the syntax is incorrect, the user is notified, but the task is still defined as completed. The validation for semantics is considered out of scope.

2) **CONNECTED[Name="", Nr=0, Stereotype-Name="", AttributeValue="AttributeName": "", OUTPUT_Name=""]**: the information can be found on an associated block based on a search query, e.g. CONNECTED[Name="Sensor_Log"] for *Format_Date* in Fig. 2
3) **BLOCK**: the information is stored on the block directly
4) **STEREOTYPE["StereotypeName"]**: the information is stored on a specifically applied stereotype (blocks can inherit from multiple stereotypes)
5) **NAME**: the information is the name of the block specified by the preceding keywords
6) **ATTRIBUTES**: the information is a list of attributes defined in a specific block

IV. EVALUATION

The evaluation of the presented method aims to assess the feasibility and applicability of the method for generating executable machine learning code.

In the following, we present the case study used for the evaluation with the used artifacts from an open dataset. Additionally, an excerpt of the generated artifact is presented. The comprehensive results and generation of code is available online[6].

A. Case Study and Artifacts

As of [9], two approaches can be followed to implement a model transformation, 1) using current high-level programming languages, APIs, and frameworks or 2) relying on MDE principles and dedicated languages such as ATL[2] and Epsilon[2]. This evaluation uses traditional programming paradigms and the well-known high-level programming language JAVA.

The dataset for the evaluation is based on an open dataset[7] to predict weather forecasts based on sensor data from a weather station. The scenario of a weather forecast based on weather station data is suitable for application in the engineering domain because the data comprises multiple sensors with different timestamps and sampling rates. Additionally, the use of temperature or humidity sensors is also relevant in manufacturing specific components and the resulting quality. The model transformation concept is wholly decoupled from data-driven engineering and could therefore be evaluated for any machine learning problems.

The modeling of the machine learning tasks is depicted in a previous publication [6], extended with various comments to support the readability of the generated code. Further details, such as a representation of the model as images or the transformation result of the evaluation, are shown in the artifacts available online[8].

B. Results

This section depicts the results from the model transformation applied to the model in [6].

Figure 4 to Figure 7 depict the four parts of the developed model transformation.

Figure 4 depicts two blocks with stereotype properties defined and a block comment connected to a block, which is further used in the final Jupyter Notebook as Rich-Text Cell. The *TrainSplit* block is defined only by stereotype attributes. Additional attributes for hyper-parameter tuning, etc. are not defined. The composition indicates that the *Merge_DF* block is an input value for the *TrainSplit* function. Therefore, it is accessible through the *modelCommand* functionality defined in Listing 1.

To enable the mapping from the input model in Figure 4 to the output in Figure 7, a mapping configuration as defined in Figure 5 and a template as depicted in Figure 6 is required. The
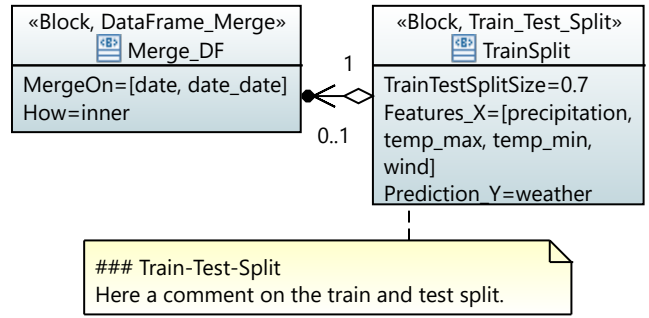

Fig. 4. Sample input model.

```
"Train_Test_Split": {
  "template": "train_test_split.vm",
  "properties": {
    "Features_X": "feat_x",
    "Prediction_Y": "pred_y",
    "TrainTestSplitSize": "split"
  },
  "modelCommands": {
    "THIS.BLOCK.NAME": "split_name",
    "CONNECTED[0].BLOCK.OUTPUT": "new_name"
  }
},
```

Fig. 5. Mapping configuration.

mapping configuration assigns a stereotype *Train_Test_Split* to a template with a name and, potentially, a path if sub-folders are used in the given structure. Each stereotype property is defined within the template's properties, whereas the left side of the assignment is the original variable in the stereotype and the right side is the placeholder in the template. The mapping defines two *modelCommands*, i.e., the first to get the name of the actual block and the second one to collect the output variable of the first connected block.

Figure 6 illustrates a sample code snippet for a machine learning function, more precisely, a template for the train-test-split. Within each template, necessary imports must be defined, and arranged at the end of the code generation, as defined in Section III-D.

Figure 7 depicts the generated code based on the template and the input model attributes. As it can be seen, the formatting is aligned with the template in Figure 6.

```
1  from sklearn.model_selection import train_test_split
2  X=${new_name}[${feat_x}]
3  y=${new_name}.${pred_y}
4  ${split_name}_train_X, ${split_name}_test_X,\
5  ${split_name}_train_y, ${split_name}_test_y = \
6      train_test_split(X, y,random_state = 0, train_size=${split})
```

Fig. 6. Template for the Train_Test_Split stereotype.

Train-Test-Split

Here a comment on the train and test splitting.

```
1  from sklearn.model_selection import train_test_split
2  X=Merge_DF[["precipitation", "temp_max", "temp_min", "wind"]]
3  y=Merge_DF.weather
4  TrainSplit_train_X, TrainSplit_test_X, \
5  TrainSplit_train_y, TrainSplit_test_y = \
6     train_test_split(X, y,random_state = 0, train_size=0.7)
```

Fig. 7. Result of the code generation.

V. DISCUSSION

This section discusses the introduced code generation method for machine learning based on model transformation and SysML. First, general advantages and disadvantages are discussed. Next, quality attributes of model transformation are discussed to allow an assessment of code generation. Finally, potential future work is presented.

A. Advantages and Disadvantages

Using model transformation to decompose formalized machine learning tasks is beneficial in several ways. First, it reduces the programming effort required for machine learning and consequently reduces the effort for rarely available data scientists [15]. In addition, it allows the formalized knowledge in the model to be validated from an implementation point of view. Validated knowledge enables the creation of a proven machine learning model library, leading to standardization of machine learning implementation within an organization's infrastructure. This potentially favors the creation of machine learning tasks without profound programming knowledge.

Nevertheless, the method can be costly for small programs and too complex and cumbersome for large-scale problems. One reason is the initial effort required to create and validate templates. However, the resulting templates lead to standardization and can thus be reused in multiple projects, which becomes an advantage in future projects. Another reason is that traceability can suffer with larger and more complex data preprocessing steps, requiring additional documentation. However, these problems are more of the nature of formalization than model transformation. Finally, it should be mentioned that the transformation is currently only directional, which does not allow changes in the generation code to be synchronized with the model.

B. Quality Attributes of Model Transformation

Quality attributes of model transformation can be distinguished in direct assessment, which is the actual assessment of the model transformation and its properties, and indirect by analyzing the input and output artifacts, e.g., metamodels [8], [16]. Furthermore, a distinction is made between internal quality, which focuses on development and maintenance, and external quality, which focuses on compliance with requirements and performance [8], [16]. In the following, direct internal quality attributes are discussed. Although various metrics are available to assess these quality dimensions, a qualitative discussion is chosen. The metrics are adapted to the transformation language used, which is not applicable here as the implementation is not based on a transformation language but on a traditional programming language [17].

1) Understandability: The effort required to understand the purpose of the model transformation [18].

The model transformation is easy to understand because a high-level programming language is used for the implementation, which can be adopted by most programmers. In contrast, the use of specific transformation languages such as ATL or Epsilon is less common and therefore the concept needs to be learned and understood. Moreover, the overall concept of mapping model artifacts using a configuration in JSON file format is a simple technique with typical concepts known from programming. A possible argument for weaknesses is that the programming of the JAVA code could be more complex to understand than MDE techniques for transformation. However, these problems are more related to software engineering than to model transformation.

2) Modifiability: The effort required to adopt a model transformation to provide other or additional functions [18].

The effort for modifications is potentially small because 1) the input metamodel can be adapted, and the concept of mapping attributes to a template is simple 2) the mapping configuration is highly customizable and can be adapted without deep programming experience, and 3) the output templates are small code fragments that can be formulated in any programming language. In addition, any functions can be added from the programming perspective by adding additional templates or stereotypes. The mapping already provides *modelCommands*, allowing the collection of specific attributes or related information. Even if more complex extensions are required, such as inserting security-related code to authenticate users, this can be adapted due to the use of the high-level language JAVA.

3) Reusability: The extent to which parts of a model transformation can be reused by other (related) model transformations [18].

Due to the possibility to exchange the output templates, the transformation can be applied to any textual programming language that enables machine learning and can be assembled from small code fragments. Similarly, the concept of transformation can be used for any other model-to-code generation that can be broken down into small code fragments, as it is simply a mapping mechanism between input stereotype and output template.

4) Modularity: The extent in which a model transformation is systematically separated and structured [18].

Modularity is given in two aspects. First, stereotypes can be arbitrarily organized as long as they inherit from the core *ML* stereotype. Second, output templates can be stored in folders to structure templates. However, the method does

not allow defining a mapping only for a specific subset of functions. Therefore, always a single JSON is required to represent the mapping configuration. Nevertheless, extending the method to include the ability to parse multiple JSON files for mapping configuration is possible with little effort, allowing for complete separation and modularization of certain aspects of transformation.

5) Completeness: The extent to which a model transformation is fully developed in relation to the requirements [18].

Completeness is conformance to requirements, which can be separated into functional or non-functional requirements of the model transformation.

The functional requirements for the model transformation can be summarized as the ability to generate executable machine learning code, which is given as of the first evaluation.

From a non-functional perspective, aspects such as generation performance must be evaluated. Due to the early stage of development, performance assessment using Big-O-notation is not reliable, as extensions are potentially required that distort the estimate. For this reason, the non-functional requirements are not yet assessed.

6) Consistency: The extent in which a model transformation is implemented in a uniform manner [18].

Because of the small programming code that compiles the input, mapping configuration, and templates, consistency is not a main quality criterion in this method, as it would be if ATL or Epsilon were used. Therefore, this criterion is not further discussed.

7) Conciseness: The extent to which a model transformation is free of superfluous elements [18].

Due to the high entanglement of the mapping configuration and the code templates, superfluous elements are barely available. Additionally, the functionality to add arbitrary attributes to the generation using **kwargs* reduces the number of superfluous elements. However, elements can be created during the modeling, or unnecessary templates can be defined. However, these expressions are part of the nature of the application rather than a weakness of the model transformation.

C. Future Work

Future work involves implementing improvements and validating the method within user studies to prove its applicability in industrial projects. Additionally, the systematic backflow of results from machine learning to the SysML model requires to be implemented to allow to use the yielded results in further model-based systems engineering methods. Similarly, it is beneficial if changes in the Jupyter Notebook can be traced back to the model so that synchronization and an authoritative source of truth[9] can be achieved. With respect to this, the actual transformation traces the model elements, allowing the identification of the origin, and within the Jupyter Notebook, unique block markers can be used to map the changes to the model elements. However, profound changes require a mechanism to generate further blocks or adapt templates.

---

[9]https://www.omgwiki.org/MBSE/doku.php?id=mbse:authoritative_source_of_truth

VI. CONCLUSION

This work presented a model transformation to facilitate machine learning applications using model-based techniques based on the general-purpose modeling language SysML. The goal of the code generation is to enable standardization of machine learning code within a company and allow to reuse formalized knowledge on machine learning tasks. The code generation is enabled by generic templates providing concise code snippets that are mapped using a mapping configuration defined in JSON file format to stereotypes or specific blocks in the SysML formalization. The generated executable code enables the validation of the formalized machine learning tasks in the SysML model. The method is validated in a case study and artifacts are made available online. Through the study results, RQ1: "What model characteristics can be used to derive machine learning models to enable model-driven engineering automatically?" can be answered by using stereotypes to identify features. Based on this, templates made specifically on stereotypes are selected, and attributes of the stereotypes are inserted into the template. Consistent with the answer to RQ1, RQ2: "What means of software engineering allows to extend and maintain the machine learning code derivation without profound model transformation changes?" can be answered by choosing a code generation that builds on a mapping configuration with templates and stereotyped definition.

By extending stereotypes and adapting or adding new templates, the method can be extended without changes in code generation. Moreover, this allows the target programming language to be selected based on the defined templates without changing the model transformation.

Future work will include elaborating an information backflow from the derived and potentially changed code to the SysML model to introduce a single source of truth. Further, the method will be validated in a user study to improve its applicability in practice and streamline the research agenda of model-driven engineering methods for machine learning.

REFERENCES

[1] A. Dogan and D. Birant, "Machine learning and data mining in manufacturing," *Expert Systems with Applications*, vol. 166, p. 114060, Mar. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095741742030823X

[2] J. Trauer, S. Schweigert-Recksiek, L. Onuma Okamoto, K. Spreitzer, M. Mörtl, and M. Zimmermann, "Data-Driven Engineering – Definitions and Insights from an Industrial Case Study for a New Approach in Technical Product Development," in *Balancing Innovation and Operation*. The Design Society, 2020. [Online]. Available: https://www.designsociety.org/publication/42546/Data-Driven+Engineering+%E2%80%93+Definitions+and+Insights+from+an+Industrial+Case+Study+for+a+New+Approach+in+Technical+Product+Development

[3] B. Beihoff, C. Oster, S. Friedenthal, C. Paredis, D. Kemp, H. Stoewer, D. Nichols, and J. Wade, "A World in Motion – Systems Engineering Vision 2025," INCOSE, San Diego, California, Tech. Rep., 2014.

[4] T. Huldt and I. Stenius, "State-of-practice survey of model-based systems engineering," *Systems Engineering*, vol. 22, no. 2, pp. 134–145, Mar. 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/sys.21466

[5] A. M. Madni and M. Sievers, "Model-Based Systems Engineering: Motivation, Current Status, and Needed Advances," *Disciplinary Convergence in Systems Engineering Research*, pp. 311–325, 2018.

[6] S. Rädler, E. Rigger, J. Mangler, and S. Rinderle-Ma, "Integration of Machine Learning Task Definition in Model-Based Systems Engineering using SysML," in *2022 IEEE 20th International Conference on Industrial Informatics (INDIN)*, Perth, Australia, Jul. 2022.

[7] A. H. B. Duffy, *Design Process and Performance*, ser. A Symposium in Honour of Ken Wallace, Cambridge, U.K., 2005, pp. 76–85.

[8] M. F. van Amstel, "Assessing and improving the quality of model transformations," Ph.D. dissertation, Technische Universiteit Eindhoven, Eindhoven, 2012.

[9] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, 2nd ed., ser. Synthesis Lectures on Software Engineering. San Rafael, Calif.: Morgan & Claypool Publishers, 2017, no. 4.

[10] A. Moin, M. Challenger, A. Badii, and S. Günnemann, "A Model-Driven approach to machine learning and software modeling for the IoT: Generating full source code for smart Internet of Things (IoT) services and cyber-physical systems (CPS)," *Software and Systems Modeling*, Jan. 2022. [Online]. Available: https://doi.org/10.1007/s10270-021-00967-x

[11] A. Bhattacharjee, Y. Barve, S. Khare, S. Bao, Z. Kang, A. Gokhale, and T. Damiano, "STRATUM: A BigData-as-a-Service for Lifecycle Management of IoT Analytics Applications," in *2019 IEEE International Conference on Big Data (Big Data)*. Los Angeles, CA, USA: IEEE, Dec. 2019, pp. 1607–1612. [Online]. Available: https://ieeexplore.ieee.org/document/9006518/

[12] E. Kusmenko, S. Pavlitskaya, B. Rumpe, and S. Stuber, "On the Engineering of AI-Powered Systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. San Diego, CA, USA: IEEE, Nov. 2019, pp. 126–133. [Online]. Available: https://ieeexplore.ieee.org/document/8967413/

[13] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa, "ThingML: A language and code generation framework for heterogeneous targets," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. Saint-malo France: ACM, Oct. 2016, pp. 125–135. [Online]. Available: https://dl.acm.org/doi/10.1145/2976767.2976812

[14] L. Burgueño, A. Burdusel, S. Gérard, and M. Wimmer, "MDE Intelligence 2019: 1st Workshop on Artificial Intelligence and Model-Driven Engineering," in *Proceedings of the 22nd International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '19. IEEE Press, 2021, pp. 168–169. [Online]. Available: https://doi.org/10.1109/MODELS-C.2019.00028

[15] S. Rädler and E. Rigger, "A Survey on the Challenges Hindering the Application of Data Science, Digital Twins and Design Automation in Engineering Practice," *Proceedings of the Design Society*, vol. 2, pp. 1699–1708, May 2022.

[16] M. F. van Amstel, "The right tool for the right job : Assessing model transformation quality," *Proceedings of the 34th Annual IEEE Computer Software and Applications Conference (COMPSAC, Seoul, Korea, July 19-123, 2010)*, pp. 69–74, 2010.

[17] M. F. van Amstel, v. den Brand, M.G.J., and H. Nguyen, "Metrics for model transformations," *BENEVOL 2010 (9th Belgian-Netherlands Software Evolution Seminar, Lille, France, December 16, 2010. Proceedings of Short Papers)*, pp. 1–5, 2010.

[18] M. F. van Amstel, C. Lange, and M. G. van den Brand, "Metrics for analyzing the quality of model transformations," *Proceedings 12th ECOOP Workshop on Quantitative Approaches on Object Oriented Software Engineering (QAOOSE08, Paphos, Cyprus, July 8, 2008 (co-located with ECOOP 2008))*, pp. 41–51, 2008.