Mathias Weske, Judith Michael (Hrsg.): Modellierung 2024,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2024 197

Model-Driven Engineering for Machine Learning Code Generation using SysML

Simon Rädler[1], Matthias Rupp[2], Eugen Rigger[3], Stefanie Rinderle-Ma[4]

Abstract: The complexity of engineering products increases due to more functions, components, and the number of involved disciplines. In this respect, Data-Driven Engineering (DDE) aims to integrate machine learning to support product development and help manage the increasing complexity of engineered systems. Still, the potential and opportunities of DDE are not entirely reflected in practice, which among others originate from the rarely available machine learning experts on the market and the effort for the implementation in practice. In this respect, this work depicts an approach based on model-driven engineering, allowing to automatically derive executable machine learning code based on machine learning task formalization using the general-purpose modeling language SysML. The main focus of the approach is on the generality of the model transformation using templates so that extensions and changes to the code generation can be integrated without requiring profound modifications to the code generator. The approach is evaluated in a use case in the domain of Cyber-Physical Systems, i.e., weather forecast prediction based on data from a Cyber-Physical weather system. The derived executable code promises to reduce the time for the implementation and supports the standardization of machine learning implementations within a company due to templates.

Keywords: Model-Driven Engineering; Machine Learning; Model Transformation; SysML

1 Introduction

Engineering systems are getting more complex due to the number of functions, components and the involvement of various engineering disciplines, e.g. involvement of software, electronics and mechanical engineering subsystems in Cyber-Physical (Production) Systems (CPS) [Be14]. To manage the knowledge of various disciplines during development, systems engineering and in particular, model-based systems engineering methodologies have been proposed, promising to increase development performance [HS21, HS19]. Besides the increased development performance, model-based methodologies are intended to create an authoritative source of truth, aiming to ensure the credibility and coherence of a digital artifact that its creators share with a variety of stakeholders[5]. To support engineers in design decisions and to allow the improvement of a product or production line, use-case-oriented

[1] Technical University of Munich, Germany; TUM School of Computation, Information and Technology; Department of Computer Science simon.raedler@tum.de
[2] Vorarlberg University of Applied Sciences
[3] Zumtobel Lighting GmbH
[4] Technical University of Munich, Germany; TUM School of Computation, Information and Technology; Department of Computer Science stefanie.rinderle-ma@tum.de
[5] https://www.omgwiki.org/MBSE/doku.php?id=mbse:authoritative_source_of_truth

collection of data from the product lifecycle and the utilization of data-driven algorithms have been defined as data-driven engineering, recently [Tr20]. We have recently proposed an approach to integrate data-driven algorithms and the substeps of the implementation into a model-based systems engineering approach [Rä22]. Although this approach supports the formalization of machine learning tasks using SysML, there is still a gap between the formalized knowledge within the SysML model and the actual implementation in dedicated programming languages such as Python. Fig. 1 depicts a sample implementation cycle of a real-world CPS, e.g. a robot or manufacturing machine. The real-world CPS is abstracted in the first step so that it can be represented as a model using the general-purpose language SysML, e.g. to describe structural, behavioral or functional components. Next, the formalized system is enhanced with data science formalization, e.g. requirements, data transformation or machine learning algorithms. Actually, the formalized knowledge must be manually implemented using dedicated programming languages such as Python to gain insights and improve a real-world CPS or product. In this respect, this work aims to introduce an automatic generation of the implementation to reduce the necessity to implement it manually.

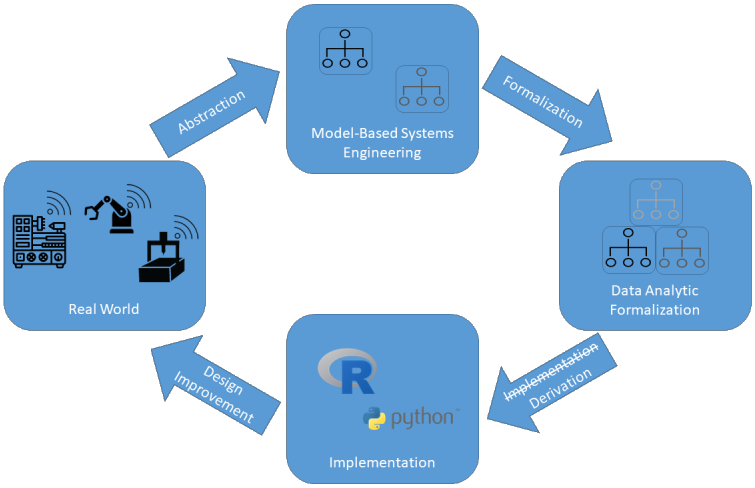


Fig. 1: Running Sample of Code Generation Cycle

With the automatic derivation, effort reduction for machine learning programming is expected, which contributes to an increasing development performance and reduces the number of data scientists, which are barely available on the market [RR22].
Concerning this, the following research questions are elaborated in this work: Given a system model representing a product's design and manufacturing environment within an enterprise: 1) What model characteristics can be used to automatically derive machine learning models to enable model-driven engineering? 2) What means of software engineering allows to extend and maintain the machine learning derivation without profound software changes? Therefore, in this work, a method is developed allowing to automatically derive executable

machine learning code based on machine learning task formalization using SysML. The code derivation is based on model transformation, supported by small and generic code chunks that facilitate the transformation's extensibility and maintainability. From a more general point of view, this work contributes by improving the efficiency and effectiveness [Du05] of the development of machine learning in the context of systems engineering. Additionally, the implementation effort for machine learning experts is reduced, supporting the integration in the industry due to the rarely available experts on the market [RR22]. This method promises to be beneficial for communication in an interdisciplinary field and supports the integration of machine learning in the early development of complex systems to build faster and better products. The method is evaluated with a smart weather station, allowing prediction of weather forecasts based on collected weather data. Future work aims to integrate a closed-loop process that automatically propagates information back from the derived and possibly changed programming code. The closed-loop process is promising to enable traceability, reproducibility and an authoritative source of truth for machine learning task definition in SysML.

2 Background

In the following, relevant background concerning Model-Based Engineering (MBE), SysML and a basic understanding of the machine learning modeling method published in [Rä22]. Additionally, comparable approaches are discussed.

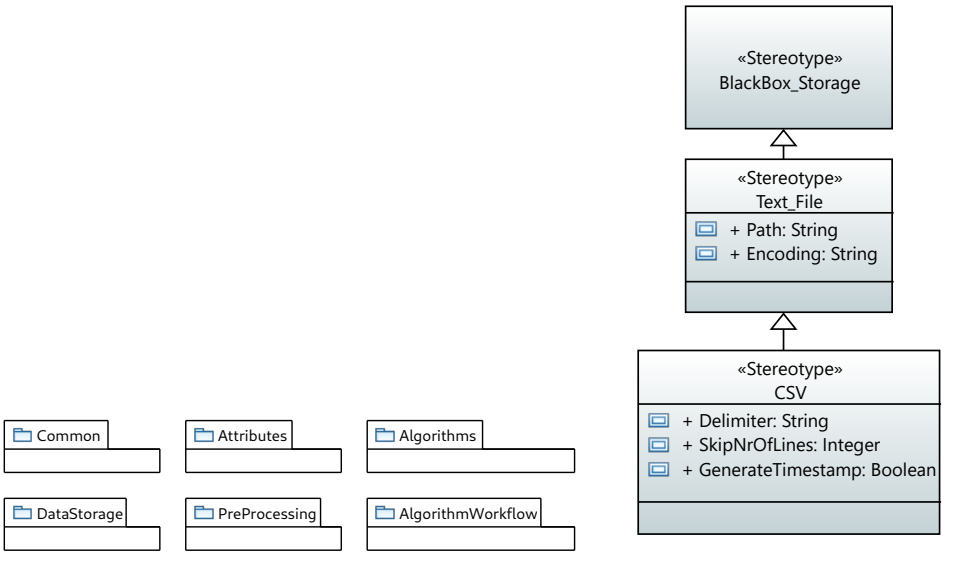
2.1 Model-Based Engineering & Model Transformation

The core of Model-Based Engineering includes the pillar concepts of models, metamodels, and model transformation [BCW17]. Depending on the applied domain, the involved engineering concepts (e.g. software, hardware, systems engineering) and the degree of automation, various acronyms are typically used for model-based engineering[6]. Model transformation can be characterized as the mapping between an input and one or multiple output models. Particularly, the mapping is defined on the metamodels, not on the actual instances of a model to allow reuse and generality. Model transformation aims to achieve the highest degree of automation by mapping artifacts [BCW17]. 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[7], Epsilon[7], etc. Model transformations can be classified as model-to-model or model-to-text transformations, depending on whether the transformation output is another model(s) or text [BCW17].

[6] See https://modeling-languages.com/clarifying-concepts-mbe-vs-mde-vs-mdd-vs-mda/ for a discussion.
[7] https://www.eclipse.org/atl/

2.2 Machine Learning Task Formalization using SysML

SysML is a general-purpose modeling language allowing to describe a system of interest with machine-readable artifacts. Its metamodel consists of definitions to describe a system from various (engineering) viewpoints and enables to represent the knowledge of various disciplines. SysML is a standardized language, being one of the core modeling languages in the context of model-based systems engineering and allows the description of functional, behavioral, and structural aspects of a system [OM07]. In preliminary work, the concept of machine learning task definition is depicted based on extensions of the SysML metamodel and the usage of core concepts of associations and generalization of SysML. In the following, the ML modeling approach's core concept is depicted according to [Rä22]. A stereotype is a concept allowing to extend the semantics of a metamodel. Each stereotype describes a specific function used in the machine learning task definition, e.g. loading a data file such as CSV depicted in Fig. 2b. Based on the steps of the CRISP-DM methodology [Ch00], a package structure is introduced in SysML to organize the machine learning stereotypes, depicted in Fig. 2a.



(a) Package Structure based on CRISP-DM Methodology

(b) Sample Stereotype Describing the Text and CSV Data

Fig. 2: Package Structure with Hierarchical Stereotypes

The stereotypes within the packages are hierarchically organized, allowing to extend the approach readily and enable to inherit from high-level stereotypes, e.g. the stereotype *CSV* in Fig. 2b inherits the attributes of the *Text_File* and *BlackBox_Storage* stereotypes, respectively. The advantage of inheritance is to allow to define specific attributes, valid for multiple stereotypes, only once. Similarly, a data type can be stereotyped, allowing to

describe an attribute in more detail. In the referenced approach, the stereotypes for the data type description are added to depict details of an input value for the machine learning. An example is shown in Fig. 3, where the *date* attribute in the *CSV_1* block on top left has a stereotype *Datetime*. The actual attribute type is *String*, describing that the attribute is represented as *String* in the *CSV* file and mapped during the import in the implementation to a *Datetime* format. For the implementation, the format of the *Datetime* is specified within the stereotype.

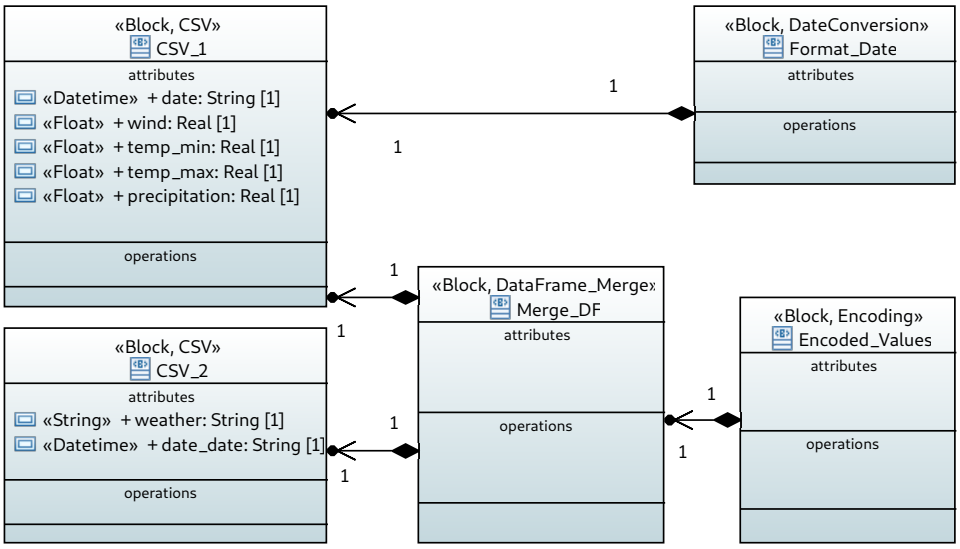


Fig. 3: Implementation of Pre-Processing Steps

Fig. 3 additionally depicts associations between blocks. With the associations, the structure of a function is described on a modular level, e.g., the *Format_Date* Block consists of the *CSV_1* block, describing that the function requires the *CSV_1* formalization to be defined in detail. For modeling the execution order of subtasks of a machine learning algorithm, state diagrams are used. A sample state diagram with stereotypes and relations to the modeled subtasks is depicted in Fig. 4.

2.3 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. In [Mo22], an extension of the CPS modeling framework ThingML [Ha16] is proposed. The extension ThingML+ allows to model machine learning aspects using a textual domain-specific language. The extension focuses on modeling supervised machine learning, and the xtext-based transformation generates both Java and Python code. Still, unsupervised and semi-supervised are possible. The Python code is used for machine learning and the Java code puts the Python machine learning application in an

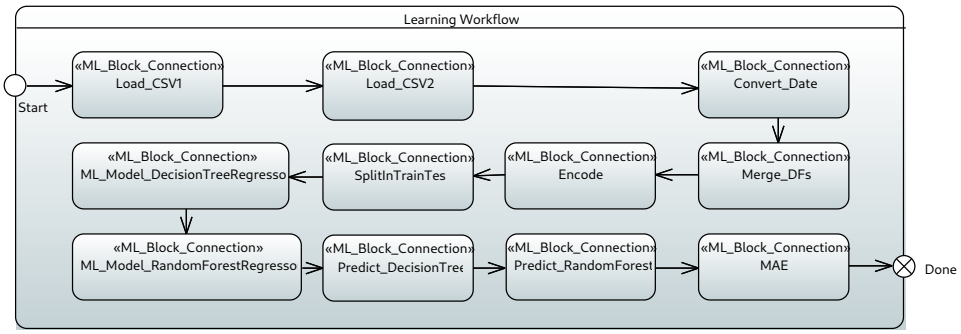


Fig. 4: Workflow Defining the Execution Order of a Machine Learning Algorithm

IoT-specific application. 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 [Bh19], 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 approach. Various machine learning frameworks are integrated and code can be generated. By using WebGME[8] as a base, a graphical modeling interface is available. The extension and customization of the code generation require code extensions. A shortcoming of the approach is the stiffness of the templates for the code generation, making it hard to use the generator for approaches other than the proposed case study.

In [Ku19], textual modeling is used to describe a neural network. The approach mainly focuses on artificial neural networks. Other modeling approaches of the umbrella-framework MontiAnna are used to describe the components&connectors. The approach requires considerable effort to add new algorithms. Additionally, mechanical or electrical engineering artifacts are separated from machine learning, making it hard to synchronize changes among the disciplines.

Summarizing the literature, machine learning code generation based on model-driven approaches is actually under development and state of the art. The 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 transfer knowledge. In this respect, in Sect. 3, a method is proposed

[8] `https://webgme.org/`

allowing to define model transformation from the SysML to any programming language of choice that shall be used to implement the machine learning solution.

3 Method

A preliminary work defined a method to describe all relevant information for implementing a machine learning approach using SysML [Rä22]. 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. More details can be found in related literature [Rä22] and Sect. 2.2.

To enable the decomposition of the SysML model, the elaborated approach relies on templates, defined as code snippets in a dedicated programming language and a mapping from a stereotype to a template. In this respect, each task of the machine learning algorithm, such as date conversion, requires a custom stereotype. The stereotype is used to identify the correct template, which is a code snippet equipped with arbitrary placeholders, defined as a named variable exchanged during the model transformation using values of stereotype attributes.

In the template, it is possible to set default values if an attribute is not set in the formalization. If no default value is given, the attribute is mandatory. 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 the instance of a block without being defined in the stereotype. The additional properties are added to a specific position in the template indicated by an anchor-indicator such as ***kwargs*. So that an additional property is usable for the algorithm, the additional property name is required to be similar to the parameter name of the dedicated programming language function but with two tailing stars, e.g. if a parameter of a diagram printing function in Python calls *X-Axis Name*, the attribute in the block must be named ***X-Axis Name*.

In case a a new function needs to be defined, a new stereotype can be added to the metamodel. To add a template for the newly defined stereotype, the model transformation does not require any changes. The transformation is generic enough that a mapping between the stereotype and the template can be defined using the JSON data format. The definition of the JSON mapping is depicted in Listing 1.

Particularly, the JSON is defined as follows: First, the mapping allows to define whether empty lines shall be trimmed during the generation of the Jupyter Notebook or not (Line 2 in Listing 1). Second, the definition of constant values allows reusing specific strings as static

```
{
  "trimEmptyLines": <true||false>,
  "constants": {
    "<TemplateVariableName>": "<ConstantValue>",
    ...
  },
  "stereotypeMappings": {
    "<StereotypeName>": {
      "template": "<TemplateName>",
      "properties": {
        "<stereotypeAttributeName>": "<TemplateVariableName>",
        ...
      },
      "modelCommands": {
        "<ModelCommandKeywordCombination>": "<TemplateVariableName>",
        ...
      }
    },
  "nameMappings": {
    "<BlockName>": {
      "template": "<TemplateName>",
      "properties": {
        "<PropertyOrStereotypeAttributeName>": "<TemplateVariableName>",
        ...
      },
      "modelCommands": {
        "<ModelCommandKeywordCombination>": "<TemplateVariableName>",
        ...
      }
    }
  }
}
```

List. 1: JSON Mapping Structure

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. Within the stereotype mapping (Line 10-13 in Listing 1), the mapping of stereotype properties to template variables is defined. 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
2. **CONNECTED[Name=, Nr=0, StereotypeName=, AttributeValue=Ättribute-Name": , OUTPUT_Name=]**: the information can be found on an associated block based on a search query, e.g. CONNECTED[Name="CSV_1"] for *Format_Date* in Fig. 3
3. **BLOCK**: the information is stored on the block directly
4. **STEREOTYPE[SStereotypeName"]**: 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
7. **STEREOTYPEofATTRIBUTE[ÄttributeName"]**: the information is stored in a data stereotype of an attribute, e.g. *Datetime* stereotype of the *date* attribute of the *CSV_1 block* in Fig. 3
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*, it is possible to use either *ATTRIBUTES* or *STEREOTYPEofATTRIBUTE* again, to dig deeper into specific information. The *OUTPUT* value is one of the most important values to connect a code block with the result of a previous one. Similar to stereotype mapping, name mapping is available, enabling to specify a mapping for a specific block via the name. The only difference is that properties defined in the mapping can also be properties of the block itself, not just the stereotype. Name mappings take precedence over stereotype mappings if both apply for a block.

With the JSON description, all necessary pre-conditions are depicted. The execution of the model transformation itself is as follows: First, each state of the state diagram is collected

and ordered ascending concerning the order of execution. Second, the connected blocks of each state with stereotypes and attributes are collected. Next, information is extracted via a model-to-model transformation to merge the state diagram and the blocks in a single representation with all information in one place. The source metamodel is SysML and the target metamodel 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. Second, a list of rich-text that can be rendered as text before a code block, modeled as owned comments in the SysML model. Third, references to connected block contexts based on the qualified name, which is a unique identifier for named SysML elements. Due to the uniqueness of the qualified name, it can be used as an identifier for attributes, blocks, etc. 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 during the mapping. Finally, an integer represents the execution order in the state diagram. The transformation is executed for each block connected to a state and each block connected to such a block. Care is taken not to execute the transformation for the same block more than once.
The list of block contexts created in the previous step is then iterated. The information is combined with the template specified by the mapping configuration and the block context. Rich-text information is directly converted to a rich-text cell, while the template variables are replaced as defined in the mapping configuration, based on the name (for properties defined in the mapping) or the collected information from the model commands. Template-based code generation produces the output code, which is then put into a source-code cell. Each block context from the state machine gets one source code cell and, optionally, one rich-text cell. 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 notebook syntax is validated, so the execution is ensured. The validation for semantics is considered out of scope.

4 Use Case

The approach is evaluated based on an open dataset[9] concerning weather prediction based on data collected with a smart weather station, which is from the domain of Cyber-Physical Systems. In the following, the use case is introduced from a general point of view. Next, an excerpt of the modeling is depicted, which is available online as Proof of Concept implementation[10].

4.1 Scenario - Data-Driven Weather Station

A weather station allows for observing weather conditions and meteorological data. Weather and meteorological data are typically recorded using a set of sensors. The combination

[9] https://www.kaggle.com/datasets/ananthr1/weather-prediction
[10] https://github.com/sraedler/MDE_for_ML_Generation

of multiple sensors is a complex CPS. Each sensor or system has at least one interface to store or report the data to an online station. Additionally, online data might be collected to complete the data set.

In this sample, a weather system consists of three offline and one online sensor, an application programming interface (API) for weather forecasts. One of the offline sensors contains a subsystem, which is depicted but not of special interest for the use case. The CPS is formalized in SysML with the output of each subsystem from a data perspective, as depicted in Fig. 5. The system itself is also formalized in the SysML model, but will not be depicted here, because it is not of special interest.

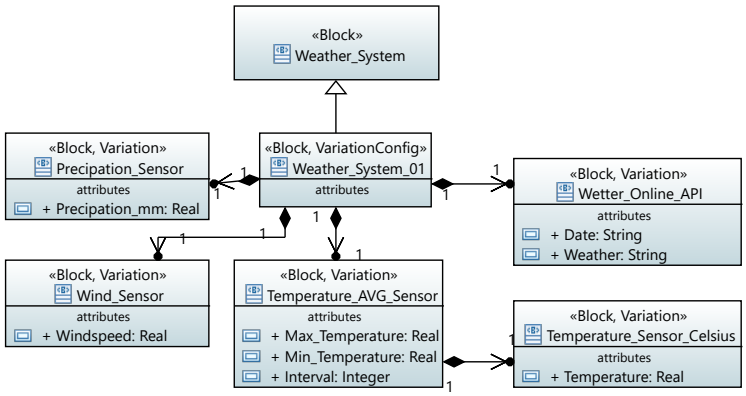


Fig. 5: The Formalized Weather System with Output Data as Attributes

The purpose of the use case is to build an application capable of learning to forecast the weather condition based on the collected data of the weather system.

4.2 Excerpt of the Model Transformation

Fig. 6 depicts a sample model transformation in an activity-like diagram. The workflow describes the model transformation of the loading of a CSV file formalized in SysML into an iPython Notebook. Starting with a SysML block, indicated by (1) on the left of Fig. 6: The SysML block contains a stereotype *CSV*, whose properties are annotated on top of the block. After the ML block is identified, ML properties are read from the Model. Each attribute of a block has a value describing a characteristic required for loading the file. The values of the model have various data types, such as boolean, integer, string or a list of strings, depending on the required type in the derived code chunk. The attribute *Encoding* has no value, meaning the default value for encoding is used, as described later. The *DateTimeColumn* is a list of column names with the specified format of DateTime. The value could also be automatically gathered from the defined attributes and the related attribute stereotype. However, with the automatic collection, custom code is required, making the approach

less generic since it only applies to the collection of DateTime attributes. Below the CSV stereotype attributes, the value attributes of the CSV file are depicted, describing the values in the file. Each attribute consists of an attribute type representing the value type in the original file. The attribute stereotype depicts the data format in the implemented Jupyter Notebook file with additional characteristics, such as the format of the DateTime value. The mapping between the block stereotypes and the template is indicated with (2) in Fig. 6. The hierarchical composition of the stereotypes in Fig. 2b is described, allowing to map from the model to the template. Due to the hierarchical structure, values of the CSV stereotype, such as *Encoding* in line 14, are not defined in the CSV mapping but in the parent stereotype *Text_File*. In Line 25, the keyword *THIS.BLOCK.NAME* is used to get the block's name and map it to the template value *varname*. Further details on the mapping in Sec. 3. (3) in Fig. 6 illustrates the template for loading a CSV file. Each variable is highlighted with the marker ${ }. If one value is given, the model must define the attribute value. A default value is available if two values are given, e.g. UTF-8 for the encoding. The tailing ***kwargs* indicates that additional properties defined in the block attributes with no reference in the stereotype and without an attribute stereotype are rendered with the format $attribute\_name = attribute\_value$. The result of the model transformation is depicted in Fig. 6 indicated by (4). Each attribute value of the model is mapped to the template, and the default value for the encoding is taken from the template due to the missing specification in the SysML model.

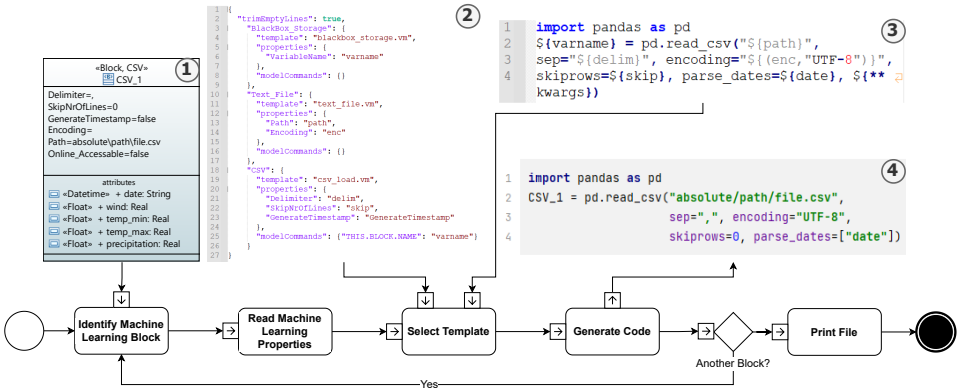
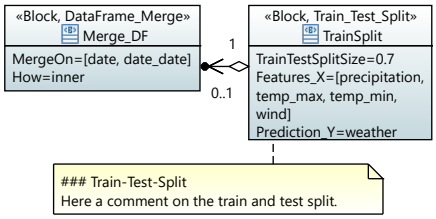


Fig. 6: A Sample Model Transformation to Load a CSV File.

Fig. 4 shows the execution order of the workflow steps of the presented approach. In Fig. 6, the first state of the workflow is depicted. In the following, the train-test-split is described, which is step number 6 in the sample workflow.

In Fig. 7a, two blocks connected with a composition and a comment are modeled. This sample shows how previous transformations and compositions are propagated among the execution workflow and the integration of comments and related rendering in the notebook depicted in Fig. 7d. In the block for the train-test-split, no additional attributes for hyper-parameter tuning, etc. are given. With the connection to the block *Merge_DF*,

the input data frame for the split is defined as shown in line 1 of Fig. 7d. Fig. 7b defines two *modelCommands*, the first to get the name of the actual block and the second one to collect the output of the previous block. Particularly, it collects the name of the new data frame in the block *Merge_DF*. In this specific sample, it would also be possible to reference the name of the block *Merge_DF*, since the name is used in its template as the output variable. However, this makes the approach less generic and leans on mapping the merging template. Fig. 7c depicts the template for the train-test-split, showing that a variable defined in the mapping can be used multiple times and filled with the same value. The result of the transformation in Fig. 7d has the same format as the template. Additionally, the markdown comment connected to the block in the SysML model is inserted before the code block.



(a) Train 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"
  }
},
```

(b) Train Mapping

```
from sklearn.model_selection import train_test_split
X=${new_name}[${feat_x}]
y=${new_name}.${pred_y}
${split_name}_${TRAIN}_${X}, ${split_name}_${TEST}_${X}, \
${split_name}_${TRAIN}_${Y}, ${split_name}_${TEST}_${Y} = \
    train_test_split(X, y,random_state = 0, train_size=${split})
```

(c) Train Template

Train-Test-Split

Here a comment on the train and test splitting.

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

(d) Train Result

Fig. 7: A Sample Model Transformation to Split a Dataset for the Training and Testing.

5 Discussion

The derivation of machine-readable artifacts, more precisely SysML models, seems promising to reduce the implementation time in programming. Therefore, it supports reducing the effort of machine learning experts, which are rarely available on the market [RR22]. Due to the creation of executable code, the approach allows validating the SysML model from the machine learning point of view. Particularly, with the execution of the machine learning commands, the formalization in SysML can be validated and an authoritative source of truth is enabled. In this respect, the approach allows building a proven model library of machine learning code, which supports reducing the effort for further implementations and standardizing machine learning implementation within a company's infrastructure.

The standardization is additionally fostered by the application of generic templates and the integration in a model-driven approach. The application of templates and small code chunks further supports debugging and fast integration of small new features. Additionally, the formalization acts as a graphical documentation that is beneficial for communication among disciplines, while the Jupyter Notebook helps experts to prove the correctness and additionally allows to add their expert knowledge during product development within a discipline-spanning communication tool. Moreover, fast prototyping can be integrated to validate various scenarios with little programming effort, formalized by non-machine learning experts.
Although the approach is beneficial for standardization and code generation contributes to efficiency and effectiveness, it may not be suitable for approaches with large-scale problems, as pre-processing is too time-consuming. For example, special transformation and the creation of advanced templates might be a complex task requiring similar effort to the programming itself. Therefore, the effort of non-experts might be shifted to the preparation of templates. However, the resulting templates are standardized and reusable within multiple projects, which might lead to a benefit in future. Additionally, the definition of alternative templates and approaches promises to lead to a well-debugged and reusable code basis that supports standardization. The model library potentially adds a foundation for an authoritative source of truth. However, the unidirectional transformation probably leads to changes in the derived file format without synchronization back to the SysML model. In this respect, future work consists of adding an information back-flow from the Jupyter Notebook file to the SysML model capable of tracking changes and supporting the trust in the correctness of modeled information. Additionally, focus is put on optimizing the ML task formalization since it is the foundation for the model transformation and potentially is the root of issues. Finally, the performance of the approach requires to be measured in a user study to enable improvement and evaluate the benefits in practice.

6 Conclusions

This work presented a model-based approach to derive executable machine learning code based on machine learning task definition using the general-purpose modeling language SysML. The derivation is enabled by generic templates providing small code chunks mapped to stereotypes in the SysML formalization. The predefined stereotypes describe key characteristics of the templates and the related programming function(s) behind them. The derived executable code proves the modeling, reduces the time for the programming and the effort for rarely available machine learning experts and supports the standardization of machine learning implementations within a company. The approach is validated in a use case regarding weather prediction using data from a weather system. Future work will include elaborating an information back-flow from the derived and potentially changed code to the SysML model to introduce a truly authoritative source of truth. Further, the approach will be validated in an industrial case study to improve the modeling and strengthen its applicability and usability in practice.

Acknowledgments

This work has been partially supported and funded by the Austrian Research Promotion Agency (FFG) via theÄustrian Competence Center for Digital Production"(CDP) no. 881843

Bibliography

[BCW17] Brambilla, Marco; Cabot, Jordi; Wimmer, Manuel: Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering 4. Morgan & Claypool Publishers, San Rafael, Calif., second edition, 2017.

[Be14] Beihoff, B.; Oster, C.; Friedenthal, S.; Paredis, Christiaan; Kemp, D.; Stoewer, H.; Nichols, D.; Wade, J.: A World in Motion – Systems Engineering Vision 2025. Technical report, INCOSE, San Diego, California, 2014.

[Bh19] Bhattacharjee, Anirban; Barve, Yogesh; Khare, Shweta; Bao, Shunxing; Kang, Zhuangwei; Gokhale, Aniruddha; Damiano, Thomas: STRATUM: A BigData-as-a-Service for Lifecycle Management of IoT Analytics Applications. In: 2019 IEEE International Conference on Big Data (Big Data). IEEE, Los Angeles, CA, USA, pp. 1607–1612, December 2019.

[Ch00] Chapman, Pete; Clinton, Julian; Kerber, Randy; Khabaza, Thomas; Reinartz, Thomas; Shearer, Colin; Wirth, Rüdiger: Step-by-Step Data Mining Guide. SPSS inc., 1.0:76, 2000.

[Du05] Duffy, Alex H. B.: Design Process and Performance. In: Engineering Design-Theory and Practice. A Symposium in Honour of Ken Wallace, Cambridge, U.K., pp. 76–85, 2005.

[Ha16] Harrand, Nicolas; Fleurey, Franck; Morin, Brice; Husa, Knut Eilif: 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. ACM, Saint-malo France, pp. 125–135, October 2016.

[HS19] Huldt, T.; Stenius, I.: State-of-Practice Survey of Model-Based Systems Engineering. Systems Engineering, 22(2):134–145, March 2019.

[HS21] Henderson, Kaitlin; Salado, Alejandro: Value and Benefits of Model-based Systems Engineering (MBSE): Evidence from the Literature. Systems Engineering, 24(1):51–66, January 2021.

[Ku19] Kusmenko, Evgeny; Pavlitskaya, Svetlana; Rumpe, Bernhard; Stuber, Sebastian: On the Engineering of AI-Powered Systems. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW). IEEE, San Diego, CA, USA, pp. 126–133, November 2019.

[Mo22] Moin, Armin; Challenger, Moharram; Badii, Atta; Günnemann, Stephan: 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, January 2022.

[OM07] OMG: , OMG Systems Modeling Language (OMG SysML™, Version 1.0), 2007.

[Rä22] Rädler, Simon; Rigger, Eugen; Mangler, Jürgen; Rinderle-Ma, Stefanie: 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, July 2022.

[RR22] Rädler, S.; Rigger, E.: A Survey on the Challenges Hindering the Application of Data Science, Digital Twins and Design Automation in Engineering Practice. Proceedings of the Design Society, 2:1699–1708, May 2022.

[Tr20] Trauer, Jakob; Schweigert-Recksiek, Sebastian; Onuma Okamoto, Luis; Spreitzer, Karsten; Mörtl, Markus; Zimmermann, Markus: 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.