FH Vorarlberg
University of Applied Sciences

Model-driven Machine Learning Based on the Systems Modeling Language

Using Template-based Code Generation and a Mapping Configuration

Master Thesis
Submitted in Fulfillment of the Degree
Master of Science

Vorarlberg University of Applied Sciences
Computer Science Master

Submitted to:
Simon Rädler, MSc

Handed in by:
Matthias Rupp, BSc

Dornbirn, 10.07.2022

Abstract

Systems are constantly increasing in complexity. This poses challenges to managing and using system knowledge. The Systems Modeling Language (SysML) is a modeling language specifically for systems, while Machine Learning (ML) is a tool to tackle complex problems. Currently, no bridge between systems modeled in SysML and ML regarding said systems has been proposed in literature. This thesis presents an approach that uses Model-driven Software Engineering (MDSE) and Template-based Code Generation (TBCG) to generate a ML IPython Notebook (IPYNB) from a SysML model. A mapping configuration using JavaScript Object Notation (JSON) allows the definition of mappings between SysML elements and template variables, enabling configuration and user-supplied templates. To test the approach, a SysML model describing ML to predict the weather based on data is created. Python ML templates are supplied and template variables mapped with the JSON mapping configuration are proposed in the thesis. The outcome is an executable IPYNB that contains all information from the SysML model and follows the modeled workflow. The findings of the work show that model-driven ML using SysML as a modeling language is beneficial due to the representation of ML knowledge in a general-purpose modeling language and the reusability of SysML model elements. It further shows that TBCG and a mapping configuration allow for more flexible code generation without changing the source implementation.

Kurzreferat

Systeme nehmen konstant an Komplexität zu. Dies führt zu Herausforderungen bezüglich Verwaltung und Verwendung von System-Wissen. Die Systems Modeling Language (SysML) ist eine Modellierungssprache speziell für Systeme, während Maschinelles Lernen (ML) ein Werkzeug zum Angehen komplexer Probleme ist. Zurzeit wurde keine Brücke zwischen in SysML modellierten Systemen und ML für besagte Systeme in der Literatur vorgeschlagen. Diese Arbeit präsentiert einen Ansatz, der Modellgetriebene Softwareentwicklung (MDSE) und Vorlagenbasierte Codegenerierung (TBCG) verwendet um ein ML IPython Notebook (IPYNB) aus einem SysML Modell zu generieren. Eine Mapping-Konfiguration, welche JavaScript Object Notation (JSON) verwendet, erlaubt die Definition von Mappings zwischen SysML Elementen und Variablen von Vorlagen, was Konfiguration und von Benutzenden bereitgestellte Vorlagen ermöglicht. Um den Ansatz zu testen wird ein SysML Modell erstellt, welches ML zur Vorhersage des Wetters basierend auf Daten beschreibt. Python ML Vorlagen werden bereitgestellt und Variablen der Vorlagen, abgebildet mit der JSON Mapping-Konfiguration, werden in der Arbeit vorgeschlagen. Das Resultat is ein ausführbares IPYNB, welches alle Informationen aus dem SysML Modell enthält und dem modellierten Arbeitsablauf folgt. Die Ergebnisse der Arbeit zeigen, dass modellgetriebenes ML mit SysML als Modellierungssprache vorteilhaft ist wegen der Repräsentation von ML Wissen in einer Allzweck-Modellierungssprache und der Wiederverwendbarkeit von Elementen des SysML Modells. Weiters zeigt die Arbeit, dass TBCG und eine Mapping-Konfiguration eine flexiblere Codegenerierung ohne Änderung des Quellcodes erlauben.

Acknowledgments

I would like to thank V-Research GmbH for supporting me during the work on this thesis.

Parts of the work were funded by the Austrian COMET Program (Project InTribology, no. 872176) and carried out at the “Excellence Centre of Tribology” (AC2T research GmbH) in collaboration with V-Research GmbH".

This work has been partially supported and funded by the Austrian Research Promotion Agency (FFG) via the “Austrian Competence Center for Digital Production” (CDP) no. 881843, the K2 centre InTribology, no. 872176.

I am also grateful to the Vorarlberg University of Applied Sciences. Both the staff and my colleagues made my studies both enjoyable and worthwile.

Furthermore, I am extremely grateful to my supervisor, Simon Rädler, for the invaluable feedback, which helped me greatly increase the quality of this thesis.

Finally, I would like to express my gratitude to my family and friends for their continued support and patience throughout my studies and the work on this thesis.

Contents

List of Figures

List of Code Fragments

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CPS	Cyber Physical Systems
CSV	Comma separated values
DAML	Data Analytics and Machine Learning
DSML	Domain-Specific Modeling Language
IoT	Internet of Things
IPYNB	IPython Notebook
JSON	JavaScript Object Notation
M2M	Model-to-Model
M2T	Model-to-Text
MDA	Model-driven Architecture
MDSE	Model-driven Software Engineering
ML	Machine Learning
PoC	Proof of Concept
SE	Systems Engineering
SysML	Systems Modeling Language
TBCG	Template-based Code Generation
UML	Unified Modeling Language

1. Introduction

Systems are growing more complex, and the needs they must respond to grow increasingly more diverse. This requires the systems of the future to make use of the evolving body of technology and more sophisticated analysis tools. (Beihoff et al., 2014) One technology that is suited to analyze complex problems is Machine Learning (ML) (Géron, 2019). However, to apply ML, the relevant information about the system needs to be available in a structured way (Beyerer, Maier, & Niggemann, 2021). Systems Modeling Language (SysML) is a general-purpose modeling language, specifically meant to deal with Systems Engineering (SE) problems, that provides a formalized structure (Arnould et al., 2019). This thesis aims to bridge the gap between a SysML model, providing structured knowledge, and ML, which uses this knowledge to draw conclusions.
To this end, techniques from Model-driven Software Engineering (MDSE) are used. The ML information is integrated into the SysML model, using its profile mechanism (Arnould et al., 2019). Model transformations and a custom metamodel are used to condense the information stored in the SysML model (Brambilla, Cabot, & Wimmer, 2017). Using this condensed information, Template-based Code Generation (TBCG) is used to generate executable ML code in the form of an IPython Notebook (IPYNB). To allow configuration of how the information from the SysML model is to be mapped to the template variables, a mapping configuration is introduced. A concept to propagate knowledge back from the executed output to the SysML model is proposed, to keep the knowledge gained through ML in the formalized structure that is the SysML model.
The resulting Proof of Concept (PoC) therefore supports the analysis of systems knowledge with ML and the management of systems knowledge with SysML.
In this chapter, concepts and terminology that are important to the understanding of the thesis will be introduced first in section 1.1. Currently existing literature and research will be presented and the research gaps identified in section 1.2. The research questions and limitations of this thesis in section 1.3 will conclude the chapter.

1.1. Concepts

Models are simplified abstractions of systems. They were mainly used for documentation, but have taken a more central role in software engineering in recent years (Rodrigues da Silva, 2015).

Unified Modeling Language (UML) is a graphical modeling language. It is primarily meant for software development. UML allows the modeling of software-based systems. This helps developers design, understand and implement systems. Primitive types are data types without any substructure. Primitive types in UML are Integer, Boolean, String, UnlimitedNatural and Real

(Cook et al., 2017). While designed to describe software-based systems, UML can also model other domains, like logistics processes (Czuchra, 2010), for example.

Sillitto et al. (2019, p. 3) define SE as "...a transdisciplinary and integrative approach to enable the successful realization, use, and retirement of engineered systems, using systems principles and concepts, and scientific, technological, and management methods." The basic premise of SE is that everything has the characteristics of a system and can be described (Winzer, 2016).

SysML, as specified in Arnould et al. (2019, p. 1), is "...a general-purpose modeling language for systems engineering." which also "...reuses a subset of UML 2.5 and provides additional extensions...". Meant as support for systems engineers, interest in SysML has also been on the rise in the field of software engineering. While well suited to working with complex systems, SysML is often customized with profiles to better fit the system domain (Wolny, Mazak, Carpella, Geist, & Wimmer, 2020). Examples would be extensions for simulation modeling capabilities (Bock, Barbau, Matei, & Dadfarnia, 2017) or for modeling a manifacturing system (Vogel-Heuser, Schütz, Frank, & Legat, 2014). The profile mechanism is consistent with OMG (2016). The primary mechanism for the creation of profiles is the stereotype. Stereotypes extend metaclasses and can be applied to model elements that conform to said metaclass. To describe systems with SysML, so called blocks are used. A block is a general-purpose modular unit, that can model the kinds of components, connections between them, etc. SysML blocks are based on UML classes. The standard state machines are imported from UML. They model states and the transitions between them, representing the behaviour of an object. Several new diagrams are also introduced, including the block definition diagram, which is based on the UML class diagram. Said diagram is used to capture features of and relationships between blocks (Arnould et al., 2019). Block definition diagrams and state machine diagrams are the two types used in this thesis. A tool for modeling SysML with a graphical framework is Eclipse Papyrus[1], which will be used in this thesis (Nyamsi, 2020).

Metamodels are models that describe models. They are an abstraction of models, just like models are an abstraction of reality. Metamodels can be used to define new and extend existing modeling languages (Jeusfeld, 2009). When all elements of a model can be expressed with instances of the respective meta-elements, a model conforms to a meta-model. A metamodel of a metamodel would be a meta-metamodel. This could be continued indefinitely, but it usually does not make sense to go beyond meta-metamodels (Brambilla et al., 2017). This leads to a four-layer architecture for metamodeling.

1. M_0 layer: meta-metamodel
2. M_1 layer: metamodel
3. M_2 layer: model
4. M_3 layer: semantic artifacts like code

[1]https://www.eclipse.org/papyrus/

Layers M_0 to M_2 define the layers that follow them, while M_2 abstracts M_3 (Giese, Karsai, Lee, Rumpe, & Schätz, 2010).

Model transformations take models as input and produce models as output according to specified rules (Sendall & Kozaczynski, 2003). They are defined at the metamodel level. One metamodel is the source of the transformation, one is the target. Rules on how to map elements of one metamodel to elements of the other are defined. The model transformations can then be executed on instances of said metamodels (Brambilla et al., 2017).

A template is a blueprint describing an output. It consists of a static and a dynamic part. The static part will appear in the output unchanged. The dynamic part contains generation logic. In TBCG, a template engine is used to execute templates and replace the dynamic part with input provided at run-time (Syriani, Luhunu, & Sahraoui, 2018).

In MDSE, software is derived from models and their transformations. Everything is a model in MDSE, with transformations being models of operations on models. A modeling language that specifies the rules models must follow is needed. Such modeling languages can be designed to be specific to a certain context (Domain-Specific Languages) or to be more general in nature (General-Purpose Modeling Languages) (Brambilla et al., 2017). Code generation also plays a part in MDSE. It allows the automatic generation of an implementation from a model, making models primary artifacts of software development, thus "model-driven" (Jörges, 2013). Since the process of solving a problem in software development is a form of modeling and program code can also be considered as a model, the model-driven approach is well suited to software development (France & Rumpe, 2014).

Regular expressions, often shortened to "regex", are a tool to parse and manipulate text. They have their own notation which allows the definition of patterns that the regular expression will match. Matches in a text can then be found and processed further. Support for regular expressions is built into many programming languages, like Java, for example (Friedl, 2006).

ML is a field of Artificial Intelligence (AI) that focuses on a system learning from data. ML generally works by training a model on a set of data. Based on said trained model, future scenarios are then predicted. It is already used in everyday applications, like web searches (Awad & Khanna, 2015). It has also seen use in more complex fields, like geoscience (Bhattacharya, 2021) or Cyber Physical Systems (CPS) (Beyerer et al., 2021). It is well suited for tackling complex problems and large amounts of data. According to Géron (2019), ML can be roughly separated into:

- Supervised learning: the data is labeled (the desired solutions are known)
- Unsupervised learning: the data is not labeled
- Semi-supervised learning: the data is partially labeled

- Reinforcement learning: the system can perform actions and learns the best strategy based on rewards and penalties for said actions

Usually, the ML workflow follows the steps as described by Raschka and Mirjalili (2021):

- Preprocessing: raw data is made useable
- Learning: a model is selected and trained on training data
- Evaluation: the trained model is evaluated based on unseen test data
- Prediction: the trained and evaluated model is used to make predictions based on new data

1.2. Literature study

Brambilla et al. (2017) provide a comprehensive guide to MDSE for readers of all skill levels. This includes, but is not limited to, the basic principles behind MDSE, Model-driven Architecture (MDA), Model-to-Model (M2M) and Model-to-Text (M2T) transformations, how to integrate MDSE into the development process, etc.

Morin, Fleurey, Husa, and Barais (2016) showcase a generative middleware. Their focus is on abstracting away different programming languages, instead offering a custom language that can be compiled into various goal languages. Their custom language is based on ports and messages, which are used to define a model of the system to generate. This happens via a specific textual representation.
A custom, textual language is used instead of a general-purpose modeling language like SysML. The generation cannot be configured without changing the source code.

As presented by Harrand, Fleurey, Morin, and Husa (2016), ThingML is a Domain-Specific Modeling Language (DSML) targeted at distributed systems. With said DSML, users can create a model (also graphically) of Things (software components) and Configurations (their interconnections). A family of compilers can then generate code from this model, with various target languages available. ThingML is targeted at software developers and architects, who also have the knowledge to extend the DSML.
A custom DSML is used instead of a modeling language standard like SysML. Furthermore, no additional configuration options for the code generation are provided. The source code needs to be changed to adapt it. The same applies if the user wants to supply their own templates for code generation.

Based on ThingML, Moin, Badii, and Günnemann (2021) offer an extension that focuses on MDSE for Data Analytics and Machine Learning (DAML). Besides things and messages, ML can now also be modeled. The modeling happens via a DSML in textual form. Code can then be generated based on the model. Any extensions for the ML capabilities would require extending the DSML.

As it is based on the work done by Harrand et al. (2016), the same weaknesses apply: a custom DSML instead of a modeling language standard and no ways to configure the code generation or for the user to supply their own code to generate without changing the source code.

Stratum, as presented by Bhattacharjee et al. (2019), offers a custom DSML to compose an ML pipeline. It is built on top of WebGME[2], thus offering a graphical interface. Stratum is geared towards ML experts and focuses on supporting the building of ML models. Code generation is also supported, and various ML frameworks are integrated. Further extensions can be programmed in.
As with Harrand et al. (2016) and Moin et al. (2021), a custom DSML instead of a modeling language standard is used for modeling. No options to configure the code generation are provided, and user extensions require programming.

Hartmann, Moawad, Fouquet, and Le Traon (2017) propose an approach focused on micro learning, meaning a focus on small learning units that compose ML. To this end, they introduce a textual editor that allows the creation of models adhering to their own metamodel, an extension of UML focusing on said small learning units. They further explicitly define the connection between learning units and domain knowledge. It is intended to be used in Internet of Things (IoT) environments where performance is of the essence.
The modeling is currently only textual, though the authors state their intention to propose an additional graphical modeling language. It is also strictly targeted at ML experts.

1.3. Research questions and limitations

As shown in the literature study, the main research gaps are:

1. Not using a modeling language with a formalized specification and existing graphical tools like SysML. This hampers reusability of models and knowledge transfer to other contexts.

2. No configuration of the output generation outside of changes to the model or the source code. This reduces adaptability and extendability of the output generation.

3. No user supplied templates for code generation, which reduces extendability.

This thesis will close these gaps and answer the following research questions.

1. Is it possible to use MDSE to go from a SysML model to a ML IPYNB?

2. Can MDSE be combined with TBCG in such a way that the generation of output can be adapted without changing the input model or the underlying code?

3. Can a user supply templates for TBCG with the approach mentioned above?

The following limitations apply to keep the scope reasonable.

1. The input model is fixed as a SysML model created in Papyrus.

[2]https://webgme.org/

2. Allowing user supplied metamodels and transformations is out of scope.
3. The output will be fixed as an IPYNB.
4. The knowledge back-propagation will be conceptualized, but not implemented due to time constraints.

2. Methods

This chapter will first present the general architecture in section 2.1. It will then go into more detail on the individual parts. Namely the SysML model in section 2.2, the model transformations in section 2.3, the mapping configuration in section 2.4, how templates and mappings are handled in section 2.5, the output generation in section 2.6 and finally the concept for knowledge back-propagation in section 2.7.

2.1. General architecture

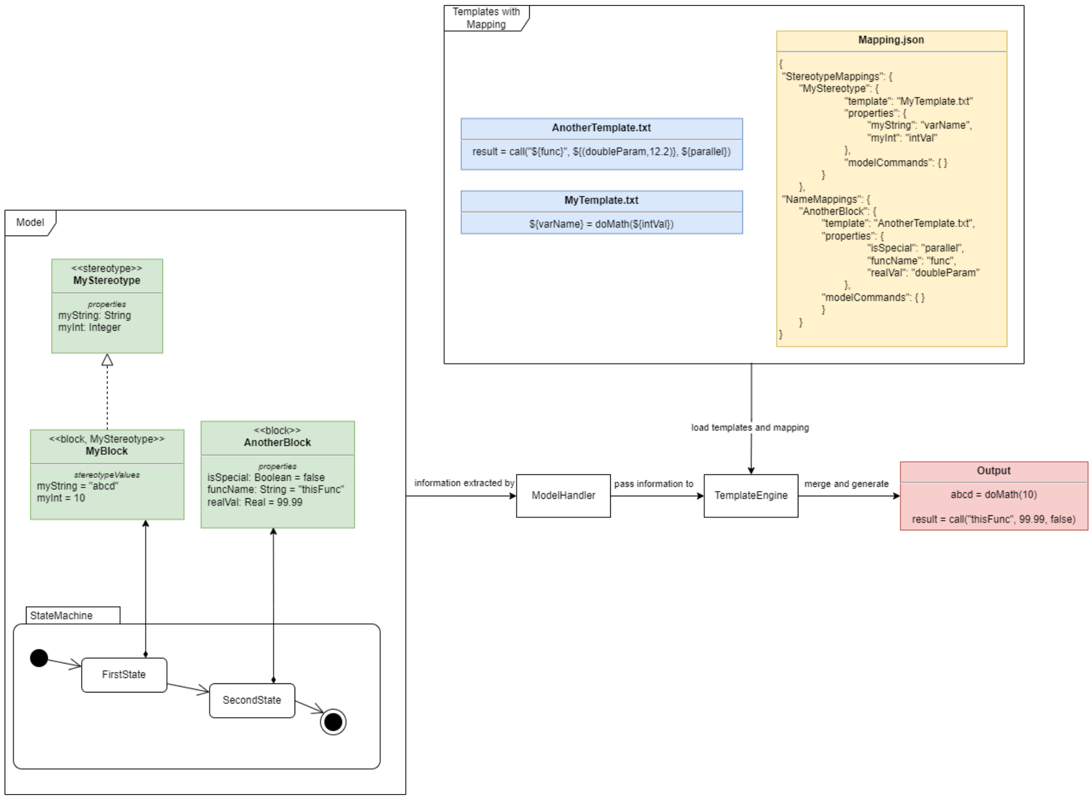
Figure 2.1 shows the architecture for this thesis. The starting point of the approach is the SysML model. In said model, an expert models the needed ML methods using stereotypes provided by a custom profile. Then, a mapping configuration in JavaScript Object Notation (JSON) format is created, with mappings for the stereotypes and individual blocks. Following that, the information needed for the generation of the ML code is extracted from the model using model transformations. The information gained from the model and the information provided by the JSON mapping configuration are combined with templates and processed. The dynamic parts of the templates are replaced with the extracted model information. From this, output is generated. This can be seen in Figure 2.1, where, for example, "${varname}" is mapped to the stereotype attribute "myString" and therefore replaced with "abcd".

2.2. SysML Model

The information in the SysML model can be roughly separated into three categories relevant for this thesis.

1. The modeled system
2. The modeled ML information for that system
3. State machine diagrams representing an ordered collection of ML information (e.g. an IPYNB)

The general system is modeled with no regard to ML. Application of ML techniques can then be done by someone with knowledge in the field, reusing information from the already modeled system. An example could be reusing the information collected by a sensor as fields of a Comma separated values (CSV) file. After the ML information is added to the model, one or several state machine diagrams are created to combine modeled ML tasks into an ordered workflow. Each state machine diagram represents a desired IPYNB output.



```
result = call("${func}", ${(doubleParam,12.2)}, ${parallel})
```

```
${varName} = doMath(${intVal})
```

```
{
 "StereotypeMappings": {
      "MyStereotype": {
                 "template": "MyTemplate.txt"
                 "properties": {
                        "myString": "varName",
                        "myInt": "intVal"
                 },
                 "modelCommands": { }
           }
      },
 "NameMappings": {
      "AnotherBlock": {
           "template": "AnotherTemplate.txt",
           "properties": {
                        "isSpecial": "parallel",
                        "funcName": "func",
                        "realVal": "doubleParam"
                 },
           "modelCommands": { }
           }
      }
}
```

```
abcd = doMath(10)

result = call("thisFunc", 99.99, false)
```

Figure 2.1.: General architecture overview

Information in the model regarding ML tasks can be separated into four categories.

1. Information of stereotype attributes
2. Information of block properties
3. Information of connected blocks
4. Information stored in comments

After applying a stereotype to a block, the attributes of the stereotype for that block can be set, with an example seen in Figure 2.2. The stereotype in question is the "CSV" stereotype, applied to the block "CSV_1". The shown attributes are:

- Delimiter: with which character the rows of the CSV are separated
- SkipNrOfLines: how many rows of the CSV are to be skipped
- GenerateTimestamp: whether a timestamp for entries is to be generated or not
- Encoding: the format the file is encoded in
- Path: the path to the CSV file
- VariableName: the name of the variable the loaded CSV is to be assigned to

Information can also be stored on properties of a block itself, as Figure 2.3 shows. Stored information can be attributes or association ends. Attributes can have types and default values of

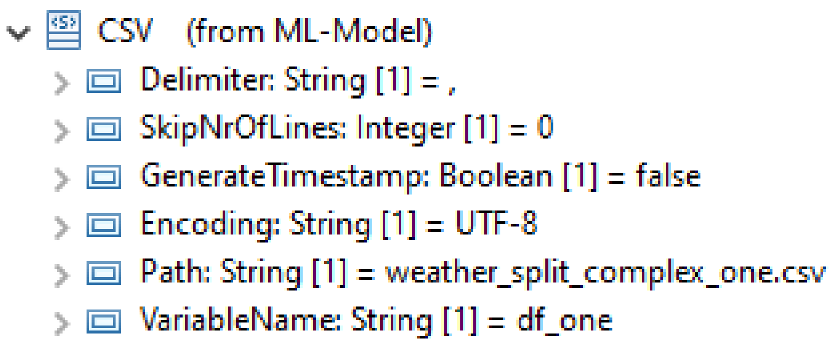

CSV (from ML-Model)
Delimiter: String [1] = ,
SkipNrOfLines: Integer [1] = 0
GenerateTimestamp: Boolean [1] = false
Encoding: String [1] = UTF-8
Path: String [1] = weather_split_complex_one.csv
VariableName: String [1] = df_one

Figure 2.2.: Stereotype attributes for a block

said type (Cook et al., 2017). In this case, the properties are attributes and show the fields saved in the CSV file represented by this block. Connections between blocks also contain information on how blocks relate to each other, a system hierarchy, for example (Arnould et al., 2019). In Figure 2.4, such a connection is shown. The connection in question is a shared association, meaning it has shared aggregration semantics. Aggregration refers to instances being grouped together by another instance. What exactly a shared aggregration means can vary (Cook et al., 2017). In this thesis, shared associations represent groupings of instances where the parts can exist without the whole. The information contained in this connection is which date shall be formatted as specified in the “DateConversion” stereotype applied to the “Format_Date” block. The connected “CSV_1” block has a “date” property, and the connection signals that this “date” property is to be converted.
Any desired markdown is added as a comment to a block, which will lead to a result as seen in Figure 2.5. Comments are converted to markdown cells that are placed before the block they belong to. In the example shown, the code cell for the date conversion is preceded by a markdown cell with the content being the same as the shown comment.
Finally, state machine diagrams collect ML tasks into a workflow. To this end, a custom stereotype is offered, which allows a state in a state machine diagram to be connected to a ML block. With this, the execution order of the generated python code is determined, with an example shown in Figure 2.6. Beginning from the entry point labelled “Start”, the workflow follows a

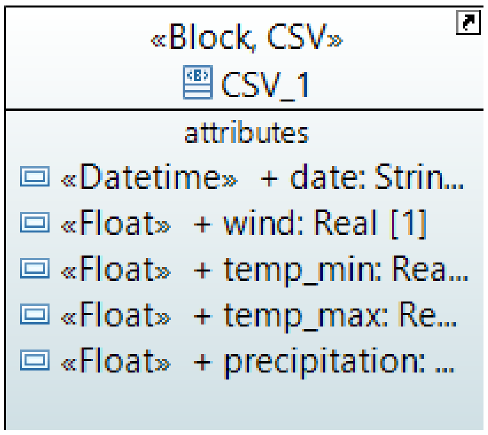

«Block, CSV»
CSV_1
attributes
«Datetime» + date: Strin...
«Float» + wind: Real [1]
«Float» + temp_min: Rea...
«Float» + temp_max: Re...
«Float» + precipitation: ...

Figure 2.3.: Properties of a block

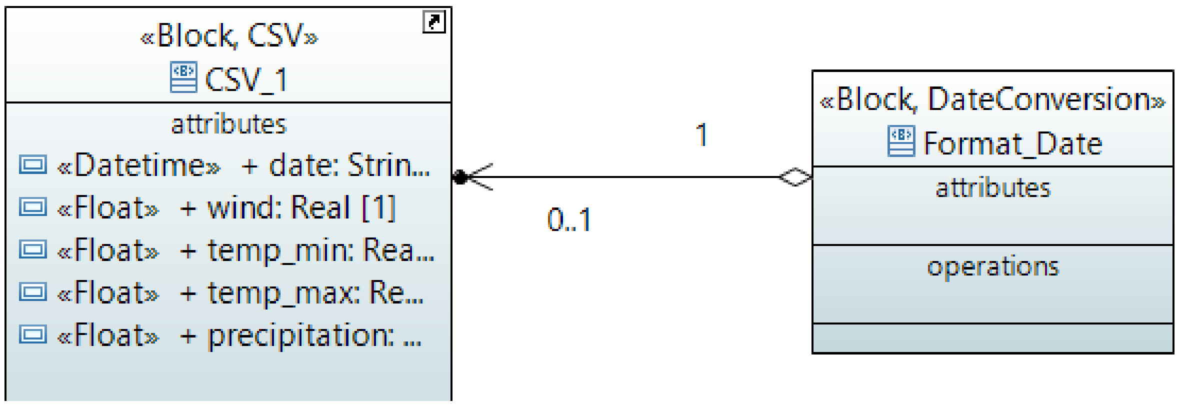
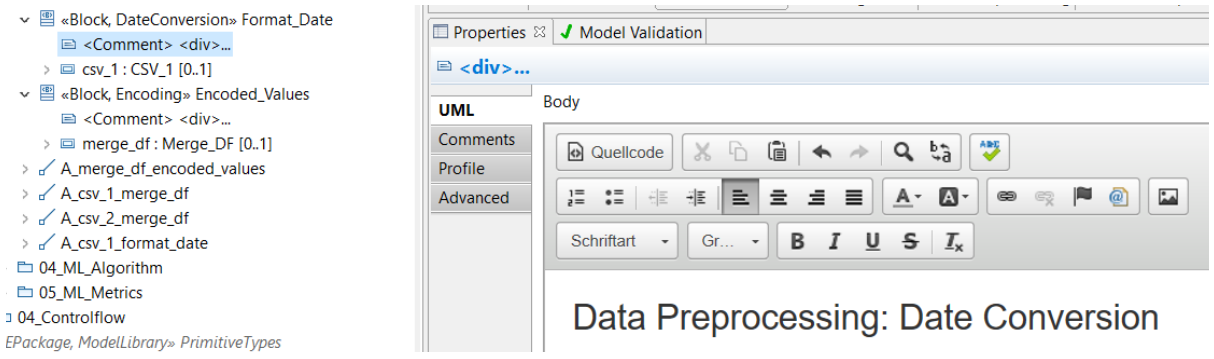


Figure 2.4.: A connection between blocks



Data Preprocessing: Date Conversion

Figure 2.5.: Weather comment example

single line of transitions, with the order of the states and their connected blocks representing the desired execution order for the output IPYNB file. The exit point, labelled “Done”, marks the end of the workflow.

2.3. Model transformation

To be able to process the information contained in the model further, the information needed for the ML code generation is extracted. A M2M transformation is used for this. The source metamodel is SysML, the target metamodel is a custom metamodel which will be described in this section.

The goal of the transformation is to decompose the SysML structure into a simpler representation, seen in Figure 2.7. For each block linked to a state in a state machine diagram, a BlockContext is created. Said BlockContext contains the qualified name of the underlying UML class of the block as well as the class itself. It also contains an integer for sorting the BlockContexts called “executionOrder”. This will be based on the order of the connected states. The BlockContext connected to the first state after the entry point will have executionOrder 0, the one connected to the following state 1, etc. BlockContexts which are not connected to a state themselves, but which are connected to one or more blocks that are connected to a state in the state machine diagram, receive an execution order of -1. A BlockContext can also contain

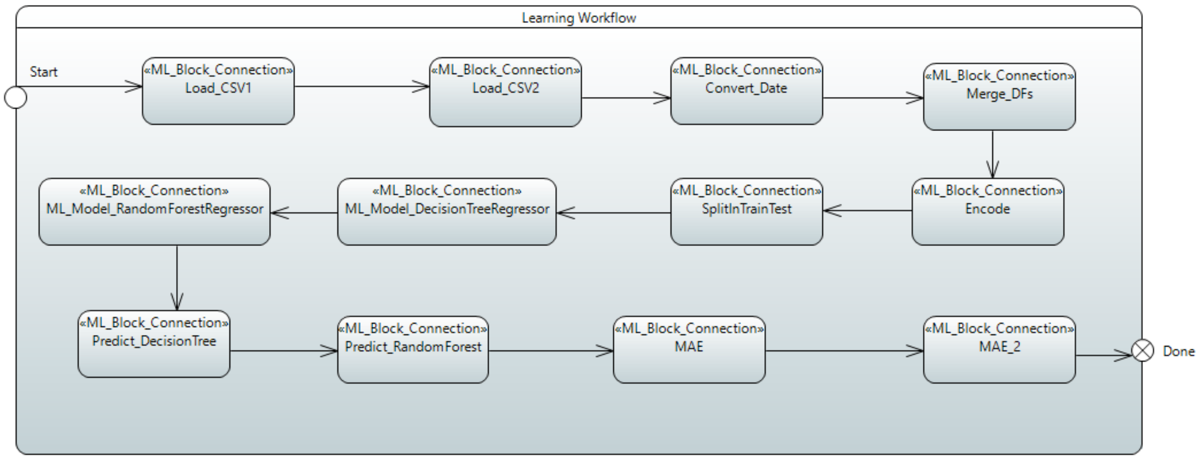

Learning Workflow
Start
«ML_Block_Connection» Load_CSV1
«ML_Block_Connection» Load_CSV2
«ML_Block_Connection» Convert_Date
«ML_Block_Connection» Merge_DFs
«ML_Block_Connection» Encode
«ML_Block_Connection» SplitInTrainTest
«ML_Block_Connection» ML_Model_DecisionTreeRegressor
«ML_Block_Connection» ML_Model_RandomForestRegressor
«ML_Block_Connection» Predict_DecisionTree
«ML_Block_Connection» Predict_RandomForest
«ML_Block_Connection» MAE
«ML_Block_Connection» MAE_2
Done

Figure 2.6.: State machine diagram as workflow

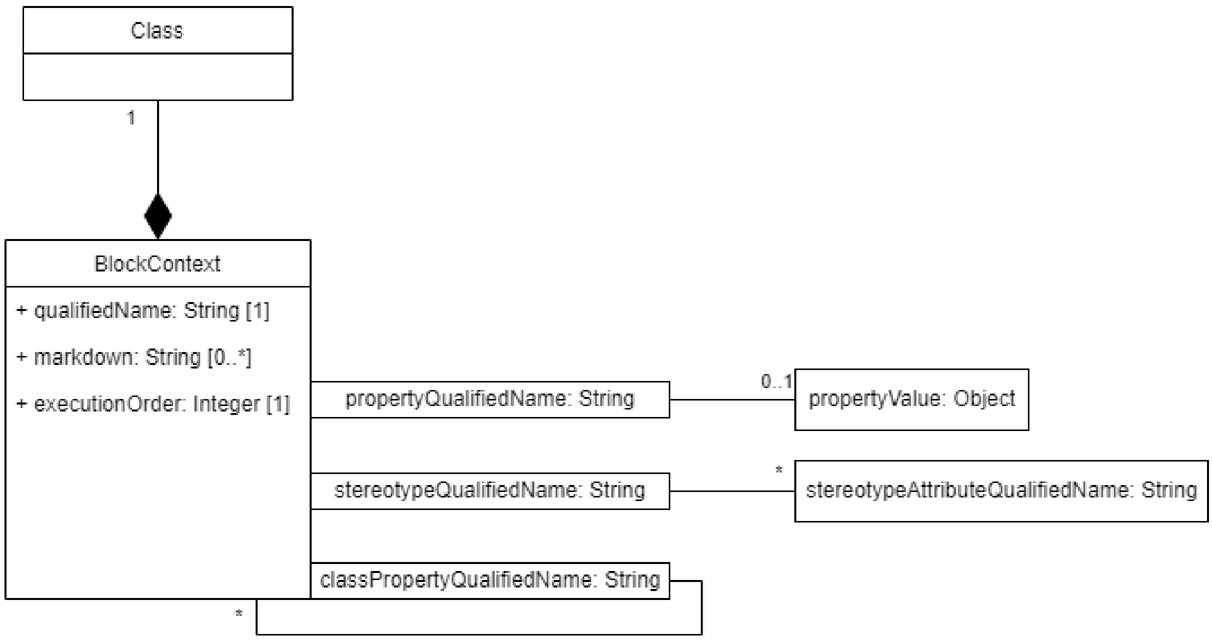
several markdown strings. It furthermore contains several maps.

1. A map between the qualified name of a stereotype and the qualified name of the attributes of said stereotype
2. A map between the qualified name of a property and its value
3. A map between the qualified name of a class property and the BlockContexts said property links to

Concrete instances of the source metamodel are SysML models, while concrete instances of the target metamodel are Java classes. To get from the source to the target metamodel, the transformation rules:

1. Decompose all properties of a primitive type to the mapping between the qualified name of the property and its value
2. Create a new BlockContext (if it does not already exist) for any property of type class and map the qualified name of the property to the newly created or already existing BlockContext or BlockContexts
3. Perform the same decomposition for the attributes of all applied stereotypes while also mapping the qualified name of the stereotype and the qualified names of its attributes

As the starting point of the transformation, a state machine diagram is used. Beginning from the entry point, the transitions are followed and for each state, a BlockContext is created for the connected block according to the laid out rules until the exit point is reached. This will transform only the part of the model specifying the ML information. After the transformation is finished, the BlockContexts are sorted by their execution order, so subsequent steps preserve the order specified in the state machine diagram. BlockContexts with an executionOrder of -1 are not directly considered during code generation, although the information they provide to the BlockContexts linked to them is. The end result of the M2M transformation is a map where the key is the UML class from the source SysML model and the value is the BlockContext created during the transformation. The map is sorted according to the execution order.



Class
1
BlockContext
+ qualifiedName: String [1]
+ markdown: String [0..*]
+ executionOrder: Integer [1]
propertyQualifiedName: String
0..1
propertyValue: Object
stereotypeQualifiedName: String
*
stereotypeAttributeQualifiedName: String
classPropertyQualifiedName: String
*

Figure 2.7.: Target Metamodel

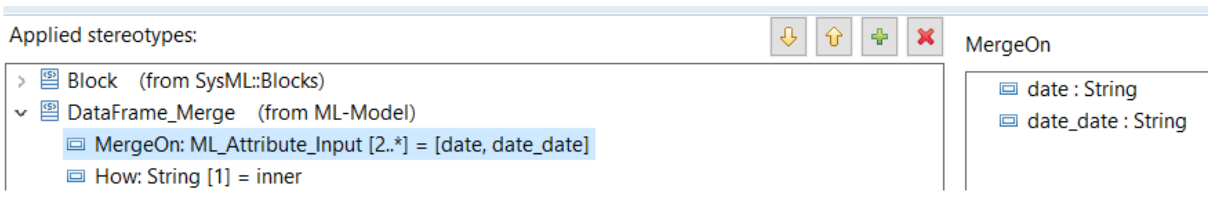
2.4. Mapping configuration

In order to allow the user to configure the provided templates and to add own templates, as well as allowing extensions in modeling, a mapping configuration is introduced. Said mapping configuration is in a JSON format and defines how attributes of stereotypes and other modeled information is mapped to template variable names. The basic structure of the mapping configuration can be seen in Code Fragment 2.1. “trimEmptyLines” is a boolean flag. If true, lines in templates that are empty or consist only of whitespaces and/or newlines are trimmed during code generation. The “blockedStereotypes” section is for configuring which stereotypes not to generate code for. Information extraction will still happen, so the information in the BlockContext will be available to other linked BlockContexts, but no code will be generated for the specified stereotypes. This will affect all blocks with the specified stereotype applied. To allow blocking the code generation for only specific blocks, the “blockedNames” section can be used. The principle is the same as with the “blockedStereotypes” section, but specific blocks can be targeted instead of stereotypes. The “constants” section allows the configuration of constants that are available to all templates.

“stereotypeMappings” is the section for defining how stereotypes should be mapped into templates. The name of the stereotype specifies which stereotype the mapping is for. The mapping contains the template name, which defines which template to use. Currently, a stereotype can only have one template assigned to it. How to map stereotype attributes to the variables of the template based on their names is defined in the “properties” section. Here, the key (the value on the left) is the original name, meaning the name the attribute of the stereotype has. The value (the value on the right) is the name of the template variable the attribute is to be mapped to. Mappings in the “properties” section are defined on the stereotype the attribute belongs to and are inherited by specializations of said stereotype without having to be mapped again. An example would be a stereotype “Text_File” that has an attribute “path”. A stereotype “CSV” is a specialization of “Text_File” and has the attribute “delimiter” and inherits “path”. The mapping for “path” is defined in the “properties” section for the stereotype mapping of “Text_File”, while the mapping for “delimiter” is defined in the “properties” section of the stereotype mapping of “CSV”.

A special case is if the stereotype attribute is a list. In this case, elements of the list can be accessed by using the name of the list, square brackets and the index of the desired element, e.g. MergeOn[0]. Note that indices start with zero. Another special feature for properties is the OWNER keyword. It allows the access of properties from the element that owns the property the OWNER keyword is applied to. Keywords are separated by periods.
An example stereotype mapping can be seen in Code Fragment 2.2. The mapping is for a stereotype called “DataFrame_Merge”. In the “properties” section, an example for both list access and the use of the OWNER keyword can be seen. “MergeOn” is an attribute of the “DataFrame_Merge” stereotype. It specifies on which attribute a merge between dataframes is to take place. It holds a list of properties marked with a custom property stereotype.
Figure 2.8 shows how this information looks like in the model. From a list of properties, elements can be accessed via indices, with the starting index being 0. In this example, “MergeOn[0]” refers to the property named “date”, while “MergeOn[1]” refers to the property named “date_date”.
With the OWNER keyword, the blocks holding these properties can be gotten.
In Figure 2.9, the owners can be seen in the SysML model. “CSV_1” and “CSV_2” are the respective owners.
The final part of the mapping example with the OWNER keyword from Code Fragment 2.2 is the name of the attribute of the owner whose value is to be mapped to the new name. “VariableName” is an attribute of the stereotype “CSV”. The “VariableName” of “CSV_1” will be mapped to the template variable named “one”, that of “CSV_2” to “two”.

The final part of a stereotype mapping is the “modelCommands” section. This section is for mappings that do not directly concern attributes of the stereotype, but other model information. Keywords are used to specify where in a model information is located. The available keywords and their meanings are described on page 25.


Applied stereotypes:
Block (from SysML::Blocks)
DataFrame_Merge (from ML-Model)
MergeOn: ML_Attribute_Input [2..*] = [date, date_date]
How: String [1] = inner

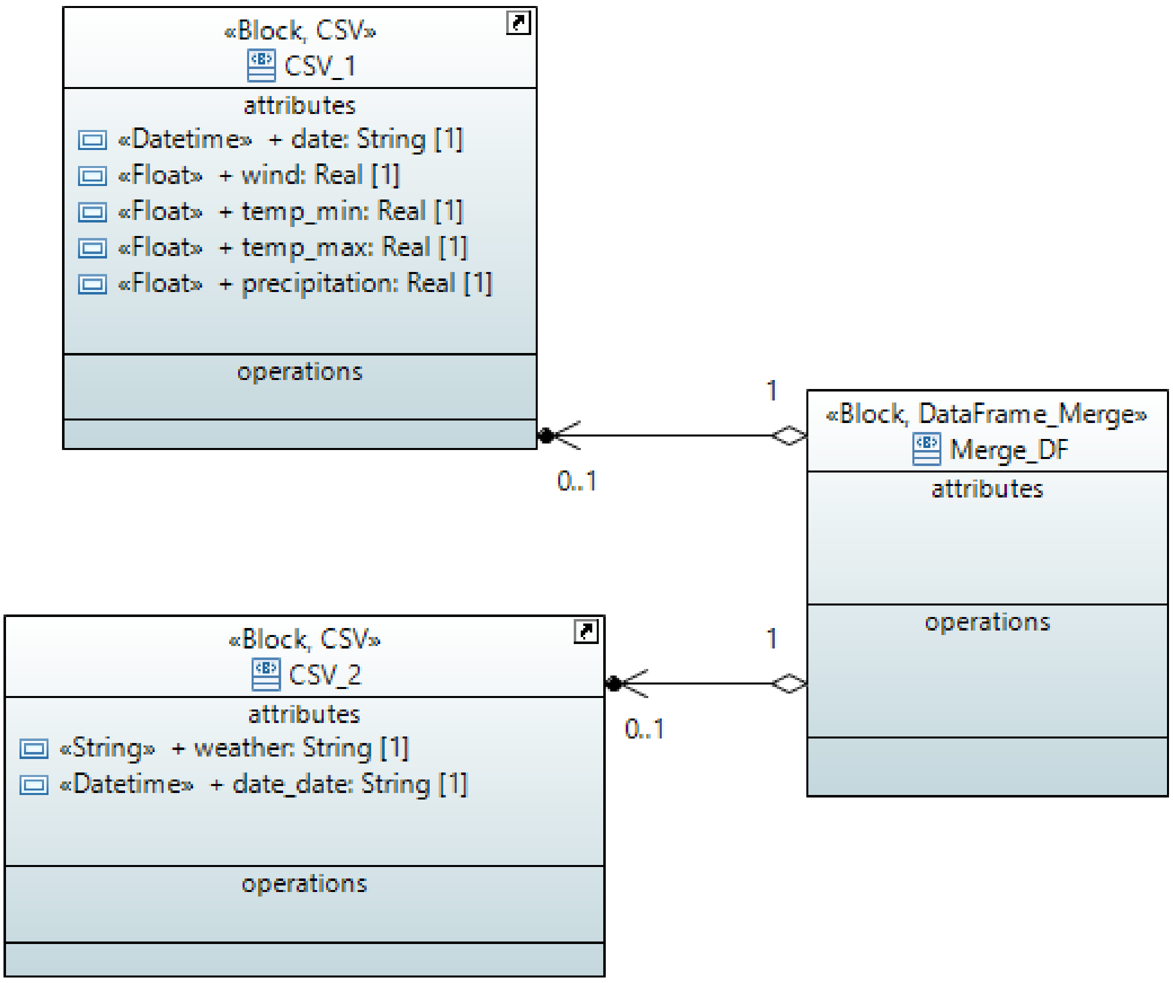
MergeOn
date : String
date_date : String

Figure 2.8.: MergeOn Attribute Example

```
{
  "trimEmptyLines": <true||false>,
  "blockedStereotypes": [
    "<blockedStereotypeName>",
    ...
  ],
  "blockedNames": [
    "<blockedBlockName>",
    ...
  ],
  "constants": {
    "<TemplateVariableName>": "<ConstantValue>",
    ...
  },
  "stereotypeMappings": {
    "<StereotypeName>": {
      "template": "<TemplateName>",
      "properties": {
        "<stereotypeAttributeName>": "<TemplateVariableName>",
        ...
      },
      "modelCommands": {
        "<ModelCommandKeywordCombination>": "<TemplateVariableName
   ↪ >",
        ...
      }
    },
  "nameMappings": {
    "<BlockName>": {
      "template": "<TemplateName>",
      "properties": {
        "<PropertyOrStereotypeAttributeName>": "<
   ↪ TemplateVariableName>",
        ...
      },
      "modelCommands": {
        "<ModelCommandKeywordCombination>": "<TemplateVariableName
   ↪ >",
        ...
      }
    }
  }
}
```

Code Fragment 2.1: Mapping Configuration Structure

```
{
    "DataFrame_Merge": {
      "template": "dataframe_merge.vm",
      "properties": {
        "MergeOn[0].OWNER.VariableName": "one",
        "MergeOn[1].OWNER.VariableName": "two",
        "MergeOn[0]": "left",
        "MergeOn[1]": "right",
        "How": "how"
      },
      "modelCommands": {
        "THIS.BLOCK.NAME": "new_name"
      }
    }
}
```
Code Fragment 2.2: Example mapping using OWNER keyword


«Block, CSV»
CSV_1
attributes
«Datetime» + date: String [1]
«Float» + wind: Real [1]
«Float» + temp_min: Real [1]
«Float» + temp_max: Real [1]
«Float» + precipitation: Real [1]
operations
1
0..1
«Block, DataFrame_Merge»
Merge_DF
attributes
operations
1
0..1
«Block, CSV»
CSV_2
attributes
«String» + weather: String [1]
«Datetime» + date_date: String [1]
operations
Figure 2.9.: MergeOn owners in model

- THIS: the information can be found on the block the stereotype is applied to
- CONNECTED: the information can be found on a block that is connected to the block the stereotype is applied to
- BLOCK: the information is stored on the block directly (currently only the name of the block is used)
- NAME: the information is the name of the block specified by the preceding keywords
- PROPNAME: the information is the name of the property specified by the preceding keywords (only meant for properties with stereotypes applied to them)

THIS and CONNECTED come first, followed by either BLOCK or the name of a stereotype. The last part of the key is the attribute which is to be mapped to the template variable name, the NAME keyword or the PROPNAME keyword. In Code Fragment 2.2, a model command can already be seen. The key consists of the three keywords THIS, BLOCK and NAME, separated by a period. This means the value to be mapped is the name of the block the stereotype "DataFrame_Merge" is applied to. It is to be mapped to the template variable "new_name".

Finally, the configuration contains a section for name mappings. These are mappings that only affect specific blocks, not whole stereotypes. The basic structure and available keywords follow the same pattern as the stereotype mappings. Name mappings block stereotype mappings. This means that if a block has a name mapping and a stereotype with a stereotype mapping defined, only the name mapping applies. Code Fragment 2.3 shows a name mapping example for a block called "CSV_1". The structure is the same as a stereotype mapping. For the rest of this thesis, the key of both "properties" and "modelCommands" mappings will be referred to as "original name", the value as "remapped name". Using Code Fragment 2.3 as an example, "VariableName" is the original name and "varname" is the remapped name.

2.5. Template and mapping handling

To generate code, the BlockContexts created via the M2M transformation and the mappings provided with the mapping configuration JSON are combined with templates. The templates used are Apache Velocity templates[1]. Custom functionality for default values was added, with an example template using a default value seen in Code Fragment 2.4. The syntax for a template variable with a default value is to use the formal reference notation[2] of Apache Velocity, but put the variable name and the desired default value in parentheses, separated by a comma. In Code Fragment 2.4, the template variable "enc" has a default value string of "UTF-8". Such default values are only used if the value mapped from the BlockContext is null, otherwise the provided value takes precedence over the default value. Note that default values need only be provided once per template, even if a variable appears multiple times.

[1] https://velocity.apache.org/engine/devel/user-guide.html#velocity-template-language-vtl-an-introduction

[2] https://velocity.apache.org/engine/devel/user-guide.html#formal-reference-notation

```
{
    "CSV_1": {
      "template": "name_csv.vm",
      "properties": {
        "VariableName": "varname",
        "Path": "p",
        "Encoding": "e",
        "Delimiter": "d",
        "SkipNrOfLines": "s",
        "GenerateTimestamp": "GenerateTimestamp"
      },
      "modelCommands": {}
    }
}
```

Code Fragment 2.3: Name mapping example

```
import pandas as pd

${varname} = pd.read_csv("${path}", sep="${delim}",
encoding="${(enc,"UTF-8")}", skiprows=${skip})
```

Code Fragment 2.4: Template with default value

Should one variable have multiple different default values defined, only the first default value specified is taken into account. Default values will be used even if they are defined after the variable in question has been used before. An example can be seen in Code Fragment 2.5. The variable test is used twice before a default value is defined, and another default value is defined later.
This will generate the output seen in Code Fragment 2.6. As can be seen, the first default value specified was used for all occurrences of the template variable.
The Apache Velocity Engine[3] is being used for TBCG. Variables for the templates are entered into a Velocity context as key-value pairs. This is where the BlockContexts and the mapping configurations are used. Template handling starts with a map of UML classes mapped to their BlockContexts, and a Java representation of the JSON mapping configuration file called MappingWrapper.

Pseudocode demonstrating the handling of the TBCG can be seen in Code Fragment 2.7. Each BlockContext is first converted into a VelocityContext and then, based on the stereotypes applied

[3] https://velocity.apache.org/engine/

```
${test}
This is a test with default variable ${test}.
${(test,12)}
Now, does the value change, ${(test,"abcdefg")}?
${test}
```

Code Fragment 2.5: Default value order example template

```
12
This is a test with default variable 12.
12
Now, does the value change, 12?
12
```

Code Fragment 2.6: Default value order example output

to the block in the SysML model, the stereotype mappings are applied. Following those, the name mappings are applied. The result of each mapping is a cell in the notebook, with the final result being a notebook consisting of said cells. Care is also taken to not execute templates multiple times.

Creating a VelocityContext from a BlockContext is done as shown in Code Fragment 2.8. If the BlockContext was not handled before, each property of the BlockContext is considered. First, the stereotype mapping is retrieved based on the property. This is possible as the properties have the qualified names, which contain the name of the stereotype if they are based on a stereotype attribute. If a stereotype mapping exists, its properties are checked for one that matches the BlockContext property. Should there be a match, it is checked whether the original name from the stereotype mapping contains the keyword OWNER. If it does, it is handled as shown in Code Fragment 2.9. Otherwise, the value of the BlockContext property is put into the VelocityContext with the remapped name from the stereotype mapping as the key. The same process is performed for a possible name mapping. Next, the model commands for both stereotype mappings and name mappings are handled, which is described in Code Fragment 2.10. Afterwards, the whole process is repeated recursively for all linked BlockContexts. The resulting VelocityContexts are merged into the one created before. "Merge" means to put all key-value pairs for keys that do not exist in the VelocityContext created at the start into said VelocityContext. This will result in one VelocityContext for each BlockContext.

Handling the OWNER keyword is done as described in Code Fragment 2.9. First, an empty set is created. The set is needed because it is theoretically possible that multiple owners are found. Currently, all found owners are added to the set and the first one is returned. The OWNER keyword combination also contains the name of the attribute to get. With the qualified name of the property the OWNER keyword is applied to, the owner can be gotten. All BlockContexts linked to the one being currently evaluated are then checked. Should the name of the owner match the name of the class connected to the linked BlockContext, the properties of said BlockContext are checked for a match with the attribute specified in the keyword combination. Should there be a match, the value of that property is added to the set created in the beginning. After checking each linked BlockContext, the first value from the set is returned. Note that currently, the OWNER keyword is limited to directly connected blocks due to the handling not being recursive.

The handling of model commands from Code Fragment 2.8 is shown in Code Fragment 2.10. All applied stereotypes from the class connected to the BlockContext are iterated over. A stereotype mapping is retrieved for each applied stereotype. Every model command from the mapping is checked for the keywords THIS and CONNECTED. Based on which keyword is found, the implementation handles the case and returns a value. This value is put into the VelocityContext

```
for each (Class, BlockContext) do
    create VelocityContext from BlockContext
    if(markdown of BlockContext is not null OR empty) do
        create markdown cell and add to notebook
    end if
    put constants from MappingWrapper into VelocityContext
    for each applied stereotype do
        get stereotype mapping from MappingWrapper
        if a stereotype mapping exists
        AND the stereotype is not blocked
        AND the name of the block is not blocked
        AND the template has not been evaluated yet do
            handle the template from the mapping
            evaluate with the VelocityEngine and VelocityContext
            add template to already evaluated templates
            create python cell and add to notebook
        end if
    end for each
    get name mapping based on class name
    if a name mapping exists
    AND the name is not blocked
    AND the template has not been evaluated yet do
        handle the template from the mapping
        evaluate with the VelocityEngine and VelocityContext
        add template to already evaluated templates
        create python cell and add to notebook
    end if
end for each
return the created Notebook
```

Code Fragment 2.7: Pseudocode for TBCG

```
Create an empty VelocityContext
if BlockContext was not already handled do
    for each property of BlockContext do
        get stereotype mapping from MappingWrapper
        for each property of stereotype mapping do
            if stereotype mapping property matches BlockContext property do
                if orignal name contains keyword OWNER do
                    handle OWNER
                    put return value into VelocityContext with remapped name
                else do
                    put value of BlockContext property into VelocityContext
                    with remapped name
                end if
            end if
        end for
        repeat process with name mapping instead of stereotype mapping
    end for
    handle model Commands
    for each linkedBlockContext of BlockContext do
        create VelocityContext from linkedBlockContext
        merge into beginning VelocityContext
    end for
end if
return VelocityContext
```

Code Fragment 2.8: Pseudocode for creating a VelocityContext from a BlockContext

```
create empty set
get owner name from property via qualified name
for each linked BlockContext do
    if owner name equals name of linked BlockContext do
        for each property of the linked BlockContext do
            if name of attribute to get equals property name
                add property value to set
            end if
        end for
    end if
end for
return first value in set
```

Code Fragment 2.9: Pseudocode for handling OWNER keyword

with the remapped name from the stereotype mapping. The name mapping is then retrieved using the name of the connected class and the process is repeated with the name mapping.
Handling the THIS keyword is shown in Code Fragment 2.11. If the second keyword is BLOCK, a method for handling the rest of the keyword combination is called, shown in Code Fragment 2.13. This means that the THIS keyword can currently only be used together with the BLOCK keyword.
How the CONNECTED keyword is handled is shown in Code Fragment 2.12. An empty set is created, since it is theoretically possible for multiple connected elements fulfilling the keyword combination to be found. Each property of each BlockContext linked to the one being currently evaluated is compared via name to the name given in the keyword combination. If there is a match, a handling method for the rest of the keyword combination, as seen in Code Fragment 2.13, is called. The return value is then added to the set. Afterwards, the linked BlockContext that was just evaluated is added to a list. All BlockContexts that are linked to the linked BlockContext are then recursively handled the same way, unless they are in the list and have therefore already been evaluated. At the end, the first element of the set is returned.
The “handle final keywords” method used before is shown in Code Fragment 2.13. If the final keyword is NAME, the name of the class connected to the BlockContext is returned. Should the keyword be PROPNAME, an empty set is created. Then, for each property of the BlockContext,

```
for each applied stereotype of connected class do
    get stereotype mapping from MappingWrapper
    for each model command in the stereotype mapping do
        if model command contains keyword THIS do
            handle THIS
            put return value into VelocityContext with remapped name
        else if model command contains keyword CONNECTED do
            handle CONNECTED
            put return value into VelocityContext with remapped name
        end if
    end for
end for
get name mapping via the name of the connected class
repeat process for name mapping
```

Code Fragment 2.10: Pseudocode for handling VelocityContext creation for model commands

```
if second keyword equals BLOCK do
    return handle final keywords
end if
```

Code Fragment 2.11: Pseudocode for handling THIS keyword

```
create empty set
for each linked BlockContext do
    for each property of the linked BlockContext do
        if the name equals the name of the connected element do
            handle final keywords
            add return value to set
        end if
    end for
    add linked BlockContext to already checked list
    for each linked BlockContext of the current linked BlockContext do
        if linked BlockContext is not in already checked list do
            handle connected element
            add returned object to set
        end if
    end for
end for
return first object in set
```

Code Fragment 2.12: Pseudocode for handling CONNECTED keyword

a comparison is made between the name of the stereotype of the property, which can be retrieved via the qualified name, and the name of the stereotype the PROPNAME keyword was applied to. If there is a match, the name of the property is added to the set. After the iteration over the properties of the BlockContext, the first element from the set is returned. The third option is no keyword, but the name of the property to get. In this case, an empty set will be created and each property of the BlockContext will be checked for a match. If there is one, the value of the property is added to the set. After checking each property of the BlockContext, the first element in the set is returned.
Next, what exactly "handle the template from the mapping" from Code Fragment 2.7 means is elaborated on in Code Fragment 2.14. After loading the template from the file, each line is processed. For each regex match, the value for the variable name from the template is retrieved from the context. If the value is null, the default value is placed in the context. Since the default values are not part of the Velocity syntax, the combination of variable name and default value is replaced with only the variable name in formal reference notation. The processed string can then directly be evaluated with the VelocityEngine and the VelocityContext with the added default values.

2.6. Output generation

The template generation described in section 2.5 results in a notebook object, the structure of which is described in Figure 2.10. A notebook consists of metadata, cells, and fields providing further information. The cells can be divided into markdown and code cells. The difference between them is that code cells have an additional execution count and outputs. Both cells have

```
if keyword equals NAME do
    return name of class connected to BlockContext
else if keyword equals PROPNAME do
    create empty set
    for each property of BlockContext do
        if stereotype name of property equals stereotype name do
            add propname to set
        end if
    end for
    return first object in set
else do
    create empty set
    for each property of BlockContext do
        if property name equals name to get
            add property value to set
        end if
    end for
    return first object in set
end if
```

Code Fragment 2.13: Pseudocode for handling final keywords

```
regex for default value = \$\{\(([a-zA-Z0-9_.-]*),"?([a-zA-Z0-9_.-]*)"?\)}
load the template from the file
for each line in the template do
    while there are regex matches in the line do
        parameter name = first group of match
        default value = second group of match
        get the value from the context with the parameter name
        if the context value is null do
            put the default value in the context
        end if
        replace default value in template with variable name
    end while
end for each
return the template string after processing
```

Code Fragment 2.14: Pseudocode for handling templates and default values an id, a cell type and source strings. The fields “variables” and “connectedElementNames” are for an eventual implementation of knowledge back-propagation as described in section 2.7 and will not be transferred to the IPYNB file. With this structure, the Java object can be directly written to a file with Jackson[4]. When given the “.ipynb” file ending, the file will be recognized as an IPYNB.

2.7. Knowledge back-propagation

“Knowledge back-propagation” means to propagate information back from the output of the executed IPYNB to the SysML model. Figure 2.11 shows Figure 2.1 updated to reflect this. The added “ChangeListener” listens for changes in the output and propagates these changes back to the model.
For this to happen, several criteria need to be met.

- The output needs to allow the observation of changes
- Said changes must be uniquely matched to elements of the intermediate model (the BlockContext) or the original model (the SysML model)
- If the match happens on the intermediate model, the changes must be then uniquely matched to elements of the original model from there

The used output for this thesis, IPYNB files, are run on servers, e.g. localhost. Variables of cells can be observed. The order of cells matches the order of BlockContexts as provided by the model transformation. As such, the variables can, in theory, be matched to their respective BlockContext counterparts. The BlockContext is both connected to the original SysML block and uses the qualified names of properties and stereotype attributes. If the SysML model has been modeled correctly, the qualified name is always unique. Therefore, the matching from the BlockContext to the SysML block is also possible. This leads to a possible workflow as seen in Code Fragment 2.15. The workflow is specific to the thesis implementation, but as long as the output is observable and changes can be matched to their original SysML elements, a similar workflow should be applicable regardless of output format.

[4]https://github.com/FasterXML/jackson

```
observe variables in cells in notebook
if variable changes do
    find BlockContext based on cell
    find variable in BlockContext
    get qualified name from BlockContext property
    get element from SysML model based on qualified name
    change value of element
end if
```

Code Fragment 2.15: Pseudocode for knowledge back-propagation concept

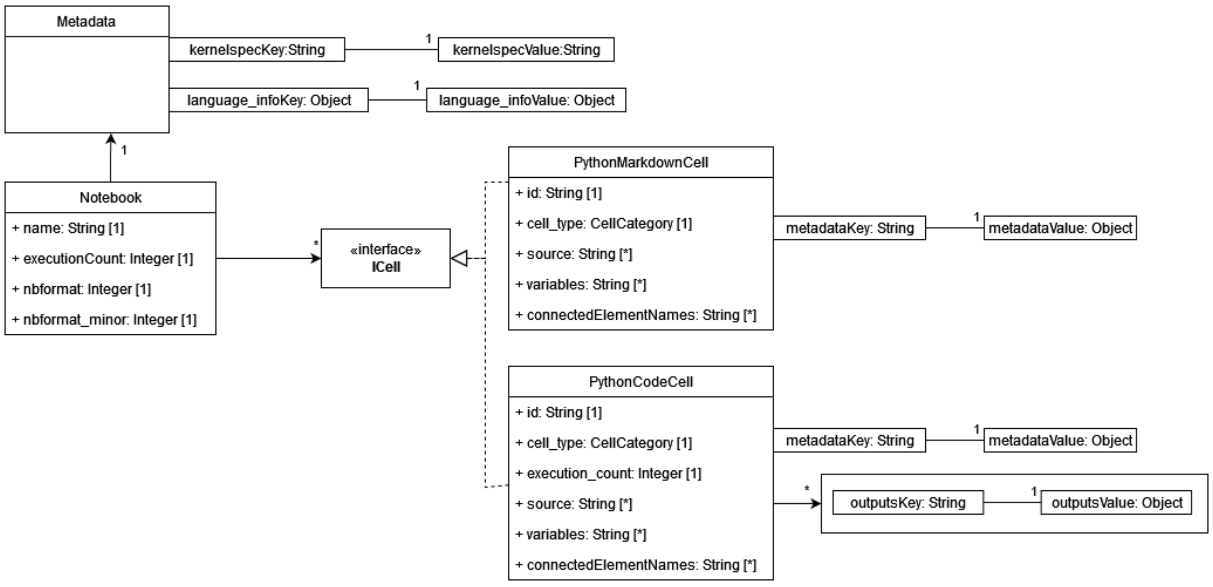
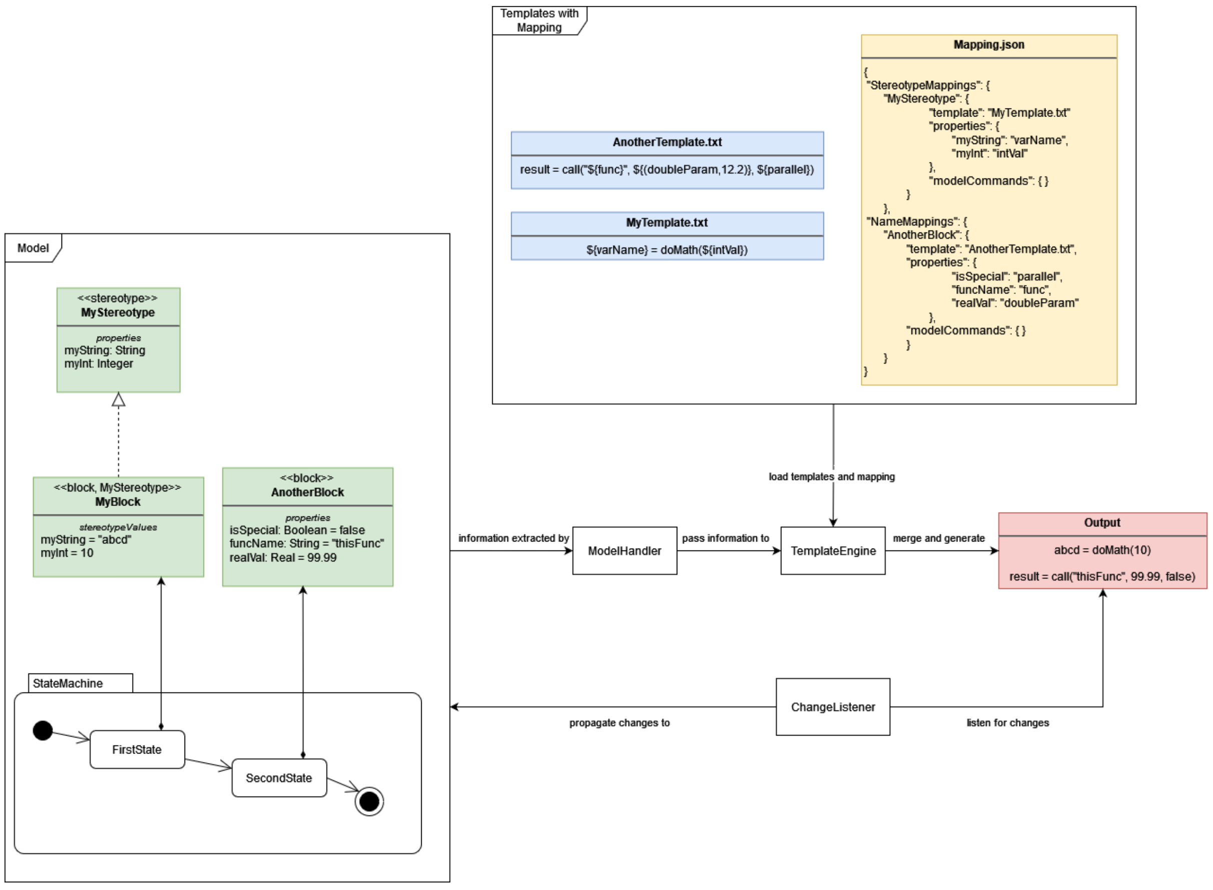


Figure 2.10.: Output notebook structure


Templates with Mapping
AnotherTemplate.txt
result = call("${func}", ${(doubleParam,12.2)}, ${parallel})
MyTemplate.txt
${varName} = doMath(${intVal})
Mapping.json

```
{
 "StereotypeMappings": {
    "MyStereotype": {
            "template": "MyTemplate.txt"
            "properties": {
                "myString": "varName",
                "myInt": "intVal"
            },
            "modelCommands": { }
        }
    },
 "NameMappings": {
    "AnotherBlock": {
        "template": "AnotherTemplate.txt",
        "properties": {
                "isSpecial": "parallel",
                "funcName": "func",
                "realVal": "doubleParam"
            },
            "modelCommands": { }
        }
    }
}
```

Figure 2.11.: General architecture with knowledge back-propagation

3. Results

This chapter will list the results of the thesis by applying the framework to a use case. section 3.1 will describe the use case, subsection 3.1.1 the SysML model created for the use case, subsection 3.1.2 will describe the mapping configuration and subsection 3.1.3 the output IPYNB.

3.1. Use case weather

For this use case, the SysML model represents a weather station. The station consists of sensors measuring various data. ML is modeled for this system using a custom metamodel. The goal of the ML is to predict the weather condition.
The data used for this use case is taken from Kaggle[1], licensed under CC BY-NC-SA 4.0[2]. Said dataset is split into two. The first dataset has the fields "date, precipitation, temp_max, temp_min ", with the format of "date" being changed from "YYYY-MM-DD" to "DD-MM-YYYY". For the second dataset, the fields are "date, wind, weather", with the "date" field being renamed to "date_date". The delimiter is changed from "," to ";".

3.1.1. Weather SysML model

The first information modeled is the information regarding the data storage. Figure 3.1 shows the diagram. Data is stored in two CSV files, with the fields modeled as properties of the respective block with the fitting stereotype for their type applied.
Stereotype attributes for the block "CSV_1" are set as shown in Figure 3.2. The delimiter and file encoding are set to correspond to the CSV file, no lines are to be skipped, no timestamp generation happens, and the the name of the variable is to be "df_one". "Path" needs to point to the location of the CSV file. This information allows one to load the information from the CSV file into a Pandas[3] dataframe.
For the Block "CSV_2", the same stereotype attributes need to be set, as seen in Figure 3.3. In comparison to "CSV_1", the delimiter and path have changed, and the variable name is set to "df_two".
How to preprocess the loaded data is modeled next. This can be seen in Figure 3.4. Both of the CSV blocks from Figure 3.1 are reused in this diagram. For the "CSV_1" block, the date is formatted so it has the same format as the date in the "CSV_2" block.
To this end, the stereotype attribute for "Format_Date" is set as shown in Figure 3.5. The only attribute of this stereotype is the desired output format of the date conversion, "%Y-%m-%d" in this case.

[1]https://www.kaggle.com/datasets/ananthr1/weather-prediction
[2]https://creativecommons.org/licenses/by-nc-sa/4.0/
[3]https://pandas.pydata.org/

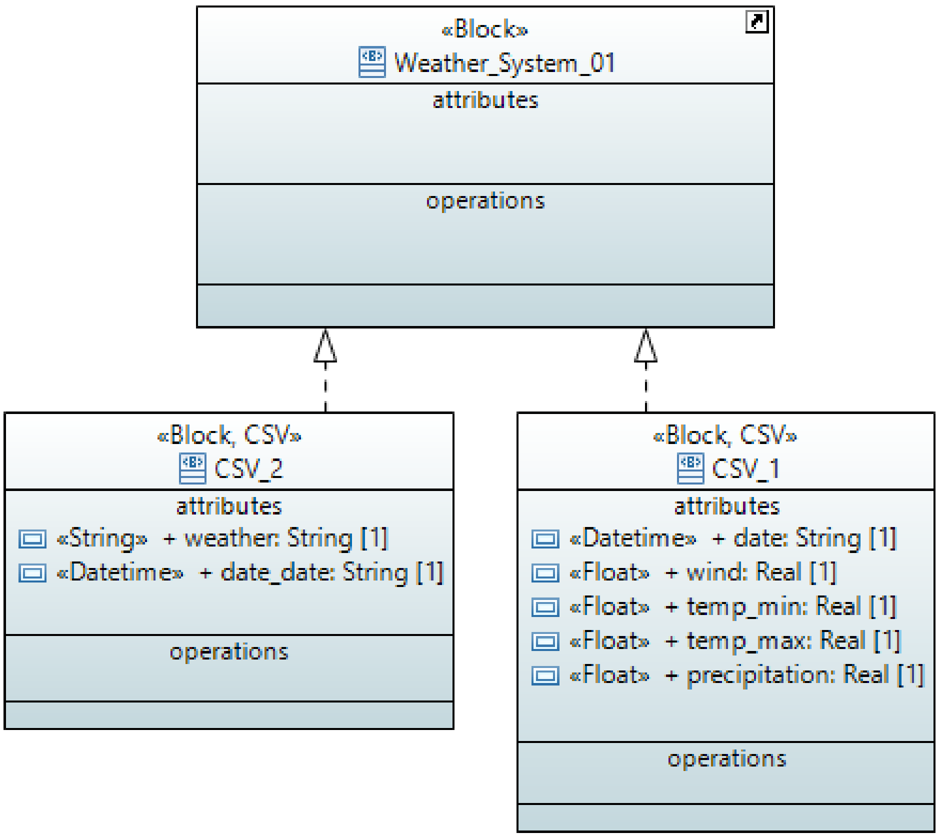
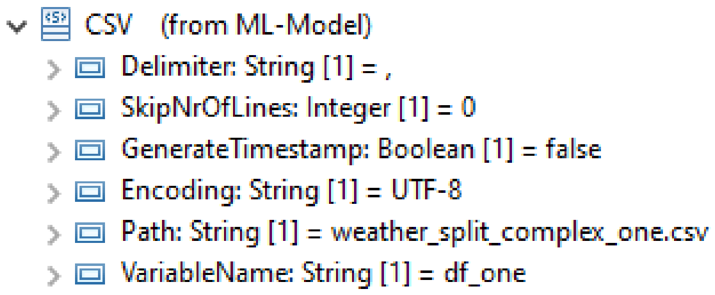
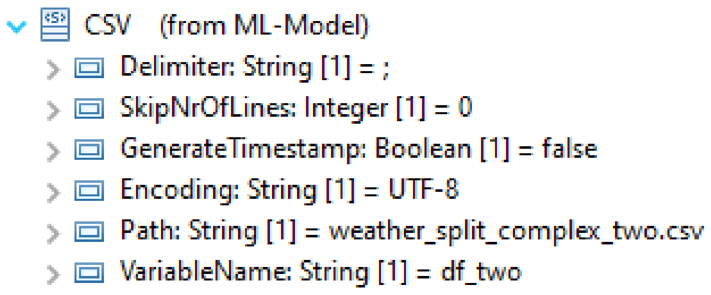

«Block»
Weather_System_01
attributes
operations
«Block, CSV»
CSV_2
attributes
«String» + weather: String [1]
«Datetime» + date_date: String [1]
operations
«Block, CSV»
CSV_1
attributes
«Datetime» + date: String [1]
«Float» + wind: Real [1]
«Float» + temp_min: Real [1]
«Float» + temp_max: Real [1]
«Float» + precipitation: Real [1]
operations

Figure 3.1.: Weather data storage model



CSV (from ML-Model)
Delimiter: String [1] = ,
SkipNrOfLines: Integer [1] = 0
GenerateTimestamp: Boolean [1] = false
Encoding: String [1] = UTF-8
Path: String [1] = weather_split_complex_one.csv
VariableName: String [1] = df_one

Figure 3.2.: Weather data storage "CSV_1" stereotype attributes



CSV (from ML-Model)
Delimiter: String [1] = ;
SkipNrOfLines: Integer [1] = 0
GenerateTimestamp: Boolean [1] = false
Encoding: String [1] = UTF-8
Path: String [1] = weather_split_complex_two.csv
VariableName: String [1] = df_two

Figure 3.3.: Weather data storage "CSV_2" stereotype attributes

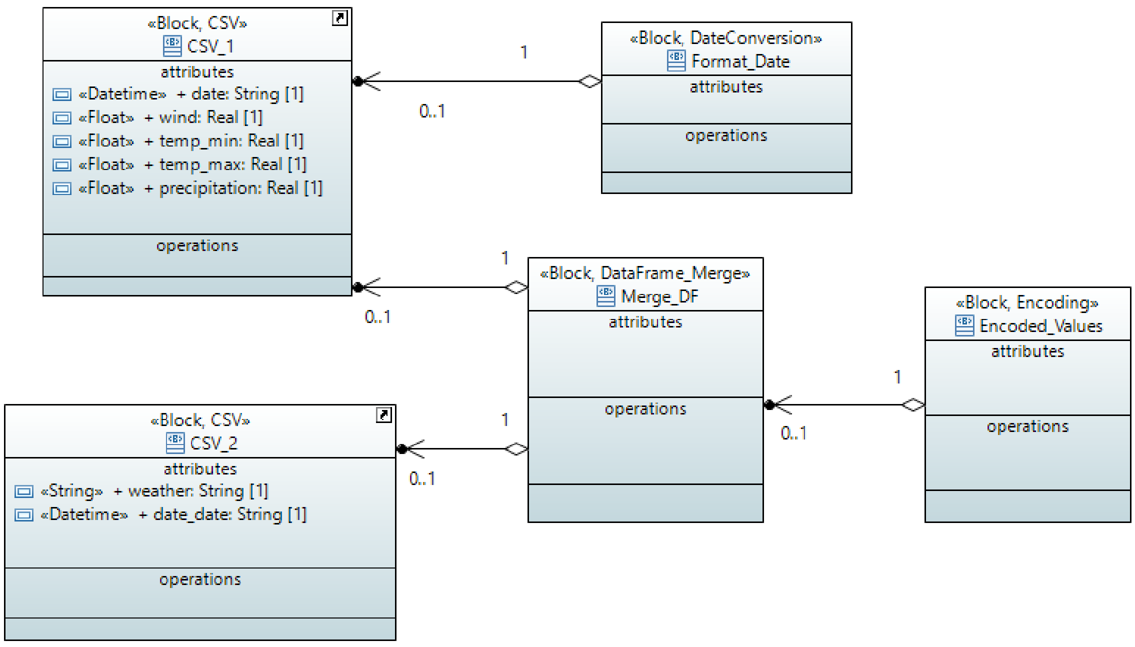


Figure 3.4.: Weather preprocessing model

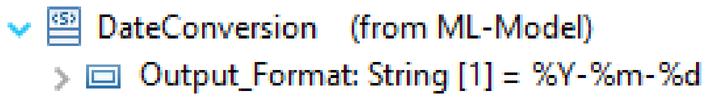

DateConversion (from ML-Model)
Output_Format: String [1] = %Y-%m-%d

Figure 3.5.: Weather preprocessing “Format_Date” stereotype attributes

Note that the information for the original format of the date is stored in the “Datetime” stereotype on the “date” property itself, as seen in Figure 3.6. The original format is “%d-%m-%Y”. The MappedToName attribute is an empty string and will not be used.
Next, “CSV_1” and “CSV_2” are merged into one dataframe. This is accomplished by connecting them to the “Merge_DF” block, which has the stereotype for merging dataframes applied with the attributes as shown in Figure 3.7. The “MergeOn” attribute specifies that the merge is to take place on the properties “date” and “date_date”, which are references to the actual properties. “How” determines the type of merge, “inner” in this case.
The final step of the preprocessing is the encoding applied to the merged dataframe. For “Encoded_Values”, the stereotype attributes are set as seen in Figure 3.8. The property to encode is “weather”, which is again a reference to the actual property that is to be encoded. The encoding is necessary to turn the string values of “weather” into numeric values that ML can work with.
Following the preprocessing, the algorithms used are defined. This is shown in Figure 3.9. The merged dataframe, reused from Figure 3.4, is split into training and test data. Algorithms which

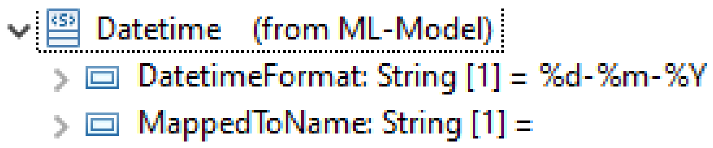

Datetime (from ML-Model)
DatetimeFormat: String [1] = %d-%m-%Y
MappedToName: String [1] =

Figure 3.6.: Weather preprocessing “date” stereotype attributes

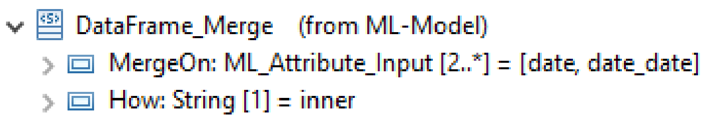

DataFrame_Merge (from ML-Model)
MergeOn: ML_Attribute_Input [2..*] = [date, date_date]
How: String [1] = inner

Figure 3.7.: Weather preprocessing “Merge_DF” stereotype attributes

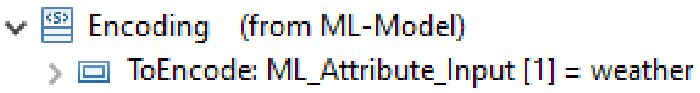

Encoding (from ML-Model)
ToEncode: ML_Attribute_Input [1] = weather

Figure 3.8.: Weather preprocessing “Encoded_Values” stereotype attributes

use the split data are defined. To model the train-test-split using the merged dataframe, the block “Merge_DF” is connected to the block “TrainSplit”.
The stereotype attributes for “TrainSplit” are set as shown in Figure 3.10. “TrainTestSplitSize” determines the split between training and test data, with 0.7 representing a split into 70 percent training data and 30 percent test data. The X and Y features are references to the properties.
“ML_1”, representing an algorithm to use, has stereotype attributes as seen in Figure 3.11. The chosen algorithm is a Random Forest Regressor, to be applied on the training and test data as split on the connected “TrainSplit” Block.
For “ML_2”, the algorithm is set as seen in Figure 3.12. It operates on the same training and test data as “ML_1”, but uses a Decision Tree Regressor instead.
After determining the algorithm to use for the ML model, it can be used to predict and measure the accuracy of said prediction. Figure 3.13 shows how this is modeled. Once again, already defined blocks are reused, “ML_1” and “ML_2”. They are then connected to “Predict_ML1” and “Predict_ML2” respectively. These blocks have the “Predict” stereotype, which has no attributes. Necessary information is gained from the connections. Each prediction is connected to a block for measuring the mean absolute error, “MAE1” and “MAE2”.
Stereotype attributes for “MAE1” can be seen in Figure 3.14. “Text”, the only attribute, is set to “First mean absolute error”. This will be the text that accompanies the mean absolute error.

For “MAE2”, the stereotype attributes are shown in Figure 3.15. The value of “Text” is “Second mean absolute error”.
To define the order in which these steps are to be executed in the output IPYNB, a state machine diagram is created, as seen in Figure 2.6 on page 20. It models the order described on page 40.

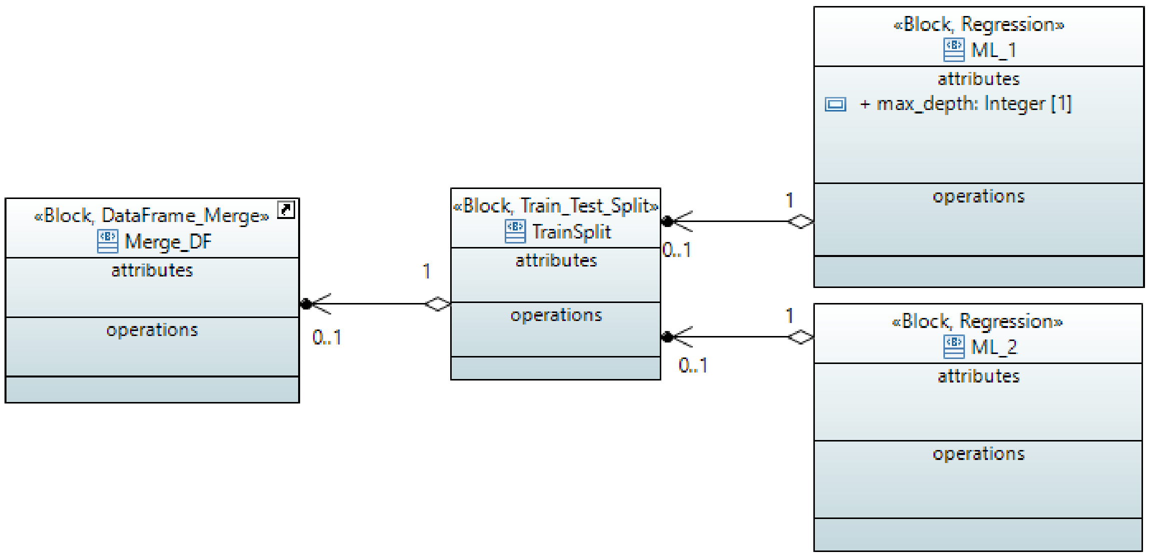
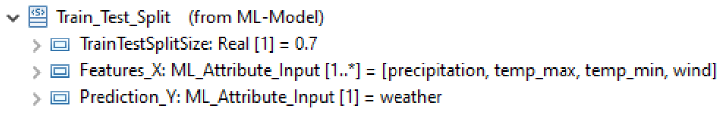
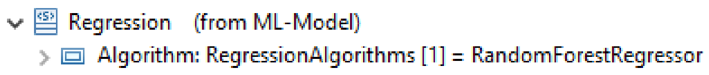


Figure 3.9.: Weather algorithms model



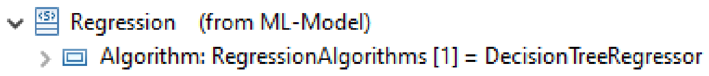
Train_Test_Split (from ML-Model)
TrainTestSplitSize: Real [1] = 0.7
Features_X: ML_Attribute_Input [1..*] = [precipitation, temp_max, temp_min, wind]
Prediction_Y: ML_Attribute_Input [1] = weather

Figure 3.10.: Weather algorithms “TrainSplit” stereotype attributes



Regression (from ML-Model)
Algorithm: RegressionAlgorithms [1] = RandomForestRegressor

Figure 3.11.: Weather algorithms “ML_1” stereotype attributes



Regression (from ML-Model)
Algorithm: RegressionAlgorithms [1] = DecisionTreeRegressor

Figure 3.12.: Weather algorithms “ML_2” stereotype attributes

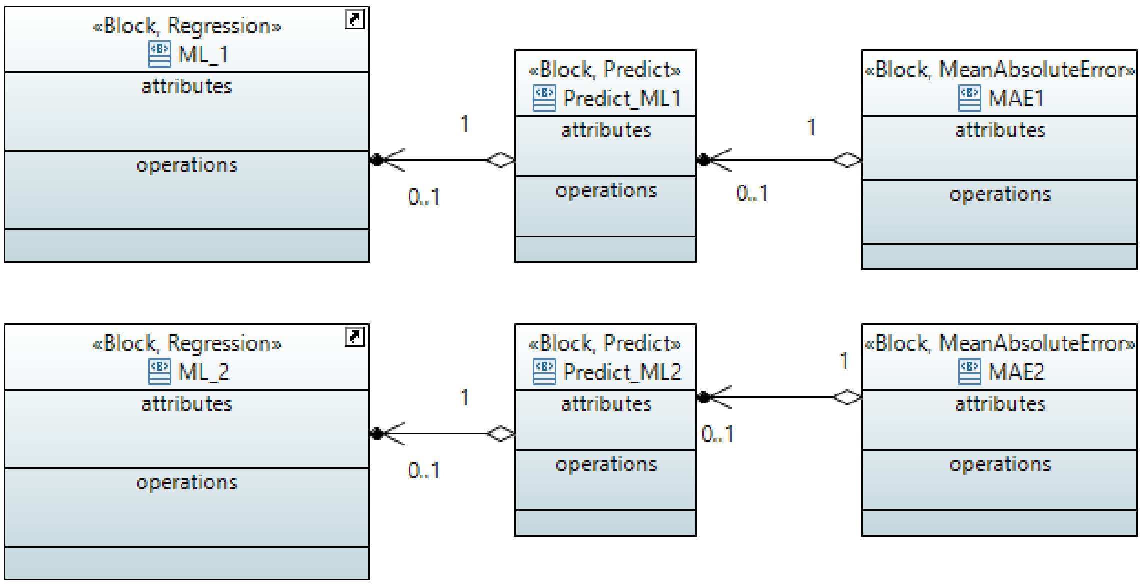
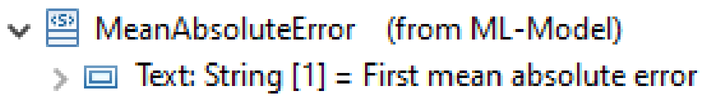


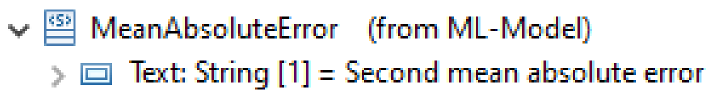
Figure 3.13.: Weather prediction and metrics model



MeanAbsoluteError (from ML-Model)

Text: String [1] = First mean absolute error

Figure 3.14.: Weather metrics “MAE1” stereotype attributes



MeanAbsoluteError (from ML-Model)

Text: String [1] = Second mean absolute error

Figure 3.15.: Weather metrics “MAE2” stereotype attributes

- Load the first CSV file
- Load the second CSV file
- Convert the date of the first CSV file
- Merge the dataframes CSV files are loaded into
- Encode the “weather” attribute on the merged dataframe
- Split the merged dataframe into training and test data
- Use a Decision Tree Regressor algorithm on the split data
- Use a Random Forest Regressor algorithm on the split data
- Predict with the Random Forest Regressor
- Predict with the Decision Tree Regressor
- Calculate the mean absolute error for the prediction of the Random Forest Regressor
- Calculate the mean absolute error for the prediction of the Decision Tree Regressor

3.1.2. Mapping configuration file and templates

The mapping between model elements and template variables is specified in a JSON file. The complete mapping file can be seen in Appendix A. The mapping file follows the specification from section 2.4, applied to the SysML model described in subsection 3.1.1. The templates said mapping refers to can be be found in Appendix B. Said templates are for Python ML code, with the dynamic part being replaced with information extracted from the model based on the mapping configuration. Stereotype mappings are used for the shown stereotypes. For the Block “CSV_1”, a name mapping is used.

3.1.3. Output

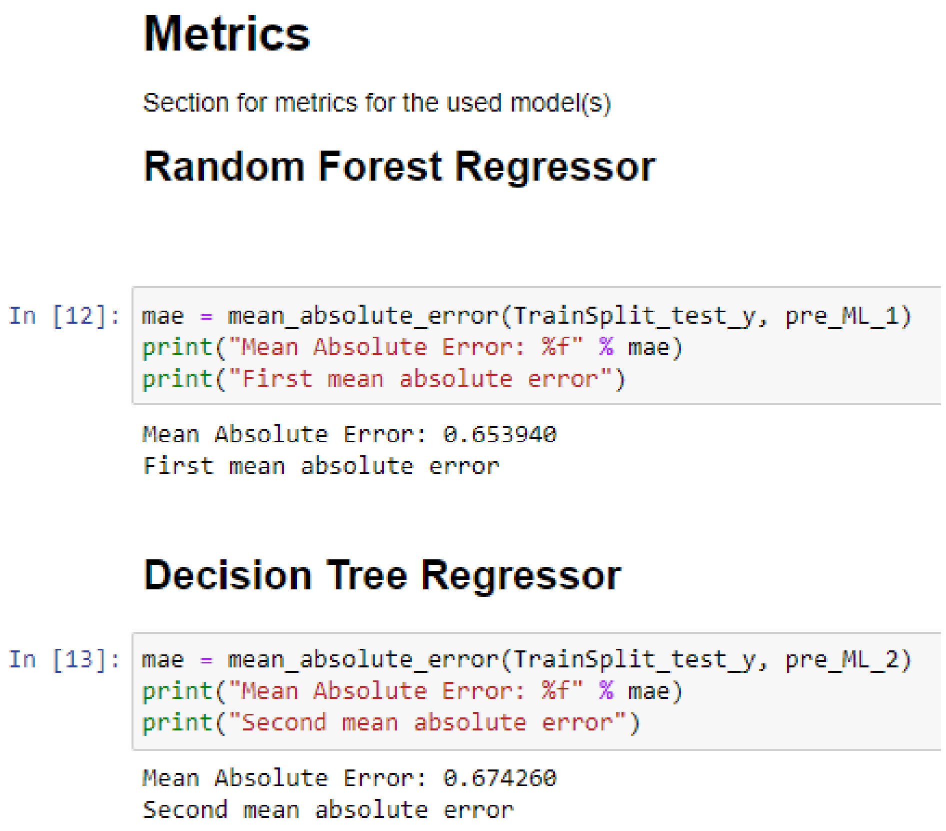
The output is an IPYNB based on the SysML model, the mapping configuration and the templates. With the given input, the notebook will:

- Import the needed libraries
- Load the CSV files into dataframes
- Convert the date of one dataframe to fit the format of the other
- Merge both dataframes into one
- Encode the “weather” field of the merge dataframe
- Split the merged dataframe into training and test data
- Train a Decision Tree Regressor on the split data

- Train a Random Forest Regressor on the split data
- Predict and show the mean absolute error with both algorithms

All comments from the SysML blocks are placed as markdown cells before the code cell that is generated from the respective BlockContext. The order of the cells follows the order defined in the state machine diagram seen in Figure 2.6 on page 20.
The output for the mean absolute error can be seen in Figure 3.16. Both algorithms have been trained and reach a similar mean absolute error. An example for a full output IPYNB file can be found in Appendix C. Keep in mind that the paths for the CSV files might need to be changed if the notebook is to be executed. Figures showing how the notebook looks like when displayed using Anaconda[4] can be found in Appendix D.

[4]`https://www.anaconda.com/`



Metrics

Section for metrics for the used model(s)

Random Forest Regressor

```
In [12]: mae = mean_absolute_error(TrainSplit_test_y, pre_ML_1)
         print("Mean Absolute Error: %f" % mae)
         print("First mean absolute error")
```

```
Mean Absolute Error: 0.653940
First mean absolute error
```

Decision Tree Regressor

```
In [13]: mae = mean_absolute_error(TrainSplit_test_y, pre_ML_2)
         print("Mean Absolute Error: %f" % mae)
         print("Second mean absolute error")
```

```
Mean Absolute Error: 0.674260
Second mean absolute error
```

Figure 3.16.: Weather notebook output mean absolute error

4. Discussion

As seen in subsection 3.1.1, many blocks can be used for multiple steps, "Merge_DF", for example. This would allow quick changes, like using another algorithm, while preserving the knowledge in the SysML model. Different state machine diagrams can be used to represent different workflows and will output different notebooks. Furthermore, the graphical representation makes the connections between steps clear. SysML is therefore a fitting way of modeling systems and can be extended via custom profiles to add functionality like ML modeling.

Due to the fact that the model transformation from SysML to the BlockContext metamodel is decoupled from the further steps, this part of the implementation can be switched out if desired. The only prerequisite is that the model transformation outputs a Map of UML class elements mapped to BlockContexts. If the order from the state machine diagram from the SysML model is to be preserved, this map needs to ordered. This transformation output can then be put into the template handling step. Possible options would be to use transformation frameworks like Eclipse Epsilon[1] or ATL[2].

The mapping configuration presented in concept in section 2.4 and in an example in Appendix A has many advantages. It:

- Allows for easy configuration of how information extracted from the SysML model is to be transferred to templates, including for information that is only linked indirectly (like names of connected blocks)
- Allows the combination of models with different templates and vice versa, without having to change the templates, models or the implementation at all
- Has a format that is human readable in JSON
- Will make custom solutions for handling the mapping configuration (e.g. automatically generating it from another source) easy, as modern programming languages usually support reading and writing of JSON files

A possible step to make the mapping configuration easier for the user would be to implement tools to support the creation of the configuration file. An example would be automatic completion and suggestions for the keywords or marking errors, such as wrong keyword order.

The keywords made available in the mapping configuration are already useful, but could be improved further. Currently, only a limited variety of three-word-combinations is possible. This

[1] `https://www.eclipse.org/epsilon/`
[2] `https://www.eclipse.org/atl/`

could be extended to allow any number of keywords chained together. Keywords are also separated between the “properties” and “modelCommands” sections of the mappings. It might be possible to eliminate that restriction, so that keywords are universal. However, whether that makes sense needs to be evaluated first. Further keywords can also be added to provide more functionality.

The Velocity templates with the added default value functionality allow for a wide variety of templates. Relatively simple templates can be used to reach good results, as demonstrated in the templates used for the weather use case, which can be seen in Appendix B. A possible step for the future is to replace Apache Velocity with a custom solution. This would allow more flexibility and more seamless integration of the TBCG part of the solution. If a different premade TBCG solution is desired, this, too, could be implemented.
The way the templates themselves are created could also be optimized. Currently, templates are created manually by an expert. Possatto and Lucrédio (2015) proposed an approach were they semi-automatically create and update templates from reference implementations, leading to less effort in maintenance. This could also be suitable for the approach proposed in this thesis.
The limit of one template per mapping could also be improved. Ways to provide multiple templates with conditions when which template is to be used could be added.

In general, the implementation has been kept as generic as possible. All stereotypes and properties from the SysML model are considered, not just the ones used in the custom ML metamodel. The BlockContext metamodel is tailored for use with SysML, but does not make any assumptions about custom profiles or the like. The only part that strictly adheres to the used ML use case is the output in the form of an IPYNB. Currently, the template handling and generation of the output notebook are intertwined. This can be changed to decouple the template handling and the generation of output completely. Following that, changing the output part of the implementation to become more generic and allowing users to provide their own implementations via Application Programming Interface (API) is certainly possible.

The implementation of the “knowledge back-propagation” that was conceptualized in section 2.7 would be a step forward. It would allow a bidirectional flow of information. The SysML model influences the output, the output influences the SysML model. Knowledge management would be eased further, since ML experts could directly write back any worthwhile changes in variable values. Going a step further, it would be interesting to make the implementation completely bidirectional. This would mean that the starting point could also be an IPYNB file, which would then lead to the generation of a SysML model. Or to the addition of elements generated from the IPYNB to an existing SysML model.
Since both source (SysML) and target (IPYNB) models can be modified, bidirectional synchronization for the models and/or their transformations, as described in Czarnecki et al. (2008) would be the next step. When a change happens, synchronization triggers. This could be further improved with the possibility to configure synchronization, i.e. only triggering synchronization when multiple changes have happened or blocking synchronization for certain elements. In the-

ory, this would allow automatic knowledge management, if the involved metamodels, model transformations and TBCG are properly implemented and the synchronization is correctly configured.
This could then be even further enhanced with a versioning system that automatically keeps older versions of the SysML model elements and the outputs. The idea would be to not save multiple versions of the model, but to have the versioned elements in the same model, in specific version packages, for example. This would allow for a graphical representation of the evolution of a model. By analyzing said evolution, insights into both the SysML modeling, as well as the ML process could be gained, since they would be synchronized. ML could be used in this analysis, meaning ML methods could be used to enhance ML. To support ML experts in making the right choices for the data, automated ML, as described in Hutter, Kotthoff, and Vanschoren (2019), could be used.

5. Threats to Validity

This chapter details possible threats to the validity of this thesis.
Since the thesis falls under the umbrella "Applied research", the validity types from most to least important are internal, external, construct and conclusion validity (Wohlin et al., 2012).

5.1. Internal validity

Threats to internal validity threaten the conclusions about the causal relationship between treatment and outcome (Wohlin et al., 2012). This thesis proposes a PoC implementation based on established technologies and concepts from literature. The causal relationship is based on the implementation logic. However, since the PoC was only used by the author, bias can not be ruled out.

5.2. External validity

External validity threats limit how well the conclusions can be generalized outside the context of the study (Wohlin et al., 2012). SysML is a standardized and widely used modeling language. The ML example used to test the implementation followed usual ML procedures and used a real-life dataset. The mapping configuration mechanism handles stereotypes and model elements in a general manner and is not context-dependent. However, only this one use case was tested, and only by the author. As such, generalizations regarding MDSE might not hold.

5.3. Construct validity

Construct validity threats concern how well the study represents the concept that is to be studied (Wohlin et al., 2012). The PoC was based on concepts from literature and used established technologies from the fields of MDSE and TBCG. The use case study evaluates whether the proposed PoC bridges the gap between SysML and ML. The fact that only one use case was tested might pose a threat to construct validity.

5.4. Conclusion validity

Threats to conclusion validity affect the ability to draw correct or reasonable conclusions from the relations between treatment and outcome (Wohlin et al., 2012). Since only one use case was evaluated by one person, conclusion validity can not be given.

6. Conclusion

In this thesis, a research gap concerning the use of standardized modeling languages like SysML in MDSE, specifically in the context of ML, was identified. Furthermore, the generation of output is rigid, not allowing the user to configure it without changing the source code. Users also cannot supply their own templates.
The proposed implementation uses SysML and its profile mechanism as the modeling language of choice. By defining the BlockContext metamodel and a model transformation between it and SysML, information is condensed. Said information is combined with templates and TBCG to generate an output IPYNB. To achieve the desired configuration, this thesis proposed a mechanism that takes a JSON mapping configuration with a defined structure and maps the information extracted from the model to templates as specified in said mapping. This also allows users to supply their own templates by adjusting the mapping configuration accordingly. By preserving unique identifiers during the model transformation, knowledge back-propagation from the output notebook to the SysML model can be conceptualized.

Regarding the research question "Is it possible to use MDSE to go from a SysML model to a ML IPYNB?", this implementation does that. A custom SysML profile is used to model ML elements and a state machine diagram specifies the desired ML workflow. Using a model transformation and TBCG, an IPYNB is generated. Said IPYNB follows the elements and workflow specified in the SysML model.
The second research question, "Can MDSE be combined with TBCG in such a way that the generation of output can be adapted without changing the input model or the underlying code?", led to the proposal of the mapping configuration. Said mapping configuration is a structured JSON file and allows the mapping of SysML stereotype attributes to template variables. Keywords can be used to map elements of the SysML model to template variables as well. TBCG is extended to allow the usage of default values in templates. This allows configuration of the output generation without having to change the underlying code.
"Can a user supply templates for TBCG with the approach mentioned above?" can be answered with "yes, but...". Using the mapping configuration, stereotype attributes and elements of the model can be mapped to any template variable, including those in user templates. However, with the current PoC, templates that do not represent Python ML code can be supplied, but the output IPYNB will not be functional.

The next step would be to have the implementation be used by engineers in real-life environments and refining the implementation based on received feedback. Several iterations of this process leveraging on industrial use cases could improve the applicability as well as the benefits of the approach. In parallel, the knowledge back-propagation, as proposed in section 2.7, should be implemented. With this improved implementation as a basis, further steps into the future of MDSE, as outlined in chapter 4, can then be taken.

References

Arnould, V., Balmelli, L., Bailey, I., Baker, J., Bialowas, C., Bock, C., ... Willard, B. (2019, November). *OMG System Modeling Language, v 1.6* (Standard No. formal/19-11-01). Object Management Group (OMG). Retrieved 2022-06-13, from https://www.omg.org/spec/SysML/1.6/

Awad, M., & Khanna, R. (2015). *Efficient Learning Machines: Theories, Concepts, and Applications for Engineers and System Designers.* Berkeley, CA: Apress. Retrieved 2022-07-04, from http://link.springer.com/10.1007/978-1-4302-5990-9 doi: 10.1007/978-1-4302-5990-9

Beihoff, B., Oster, C., Friedenthal, S., Paredis, C., Kemp, D., Stoewer, H., ... Wade, J. (2014). *A World in Motion – Systems Engineering Vision 2025.* Retrieved 2022-06-23, from https://www.researchgate.net/publication/277019221_A_World_in_Motion_-_Systems_Engineering_Vision_2025

Beyerer, J., Maier, A., & Niggemann, O. (Eds.). (2021). *Machine Learning for Cyber Physical Systems: Selected papers from the International Conference ML4CPS 2020* (Vol. 13). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved 2022-06-26, from http://link.springer.com/10.1007/978-3-662-62746-4 doi: 10.1007/978-3-662-62746-4

Bhattacharjee, A., Barve, Y., Khare, S., Bao, S., Kang, Z., Gokhale, A., & Damiano, T. (2019, December). STRATUM: A BigData-as-a-Service for Lifecycle Management of IoT Analytics Applications. In *2019 IEEE International Conference on Big Data (Big Data)* (pp. 1607–1612). Los Angeles, CA, USA: IEEE. Retrieved 2021-09-29, from https://ieeexplore.ieee.org/document/9006518/ doi: 10.1109/BigData47090.2019.9006518

Bhattacharya, S. (2021). *A Primer on Machine Learning in Subsurface Geosciences.* Cham: Springer International Publishing. Retrieved 2022-07-04, from https://link.springer.com/10.1007/978-3-030-71768-1 doi: 10.1007/978-3-030-71768-1

Bock, C., Barbau, R., Matei, I., & Dadfarnia, M. (2017, September). An Extension of the Systems Modeling Language for Physical Interaction and Signal Flow Simulation. *Systems Engineering*, *20*(5), 395–431. Retrieved 2022-06-23, from https://onlinelibrary.wiley.com/doi/10.1002/sys.21380 doi: 10.1002/sys.21380

Brambilla, M., Cabot, J., & Wimmer, M. (2017). *Model-driven software engineering in practice* (Second edition ed.) (No. 4). San Rafael, Calif.: Morgan & Claypool Publishers.

Cook, S., Bock, C., Rivett, P., Rutt, T., Seidewitz, E., Selic, B., & Tolbert, D. (2017, December). *Unified Modeling Language, v2.5.1* (Standard). Object Management Group (OMG). Retrieved 2022-06-13, from https://www.omg.org/spec/UML/

Czarnecki, K., et al. (Eds.). (2008). *Model Driven Engineering Languages and Systems: 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings* (Vol. 5301). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved 2022-06-23, from http://link.springer.com/10.1007/978-3-540-87875-9 doi: 10.1007/978-3-540-87875-9

Czuchra, W. (2010). *UML in logistischen Prozessen.* Wiesbaden: Vieweg+Teubner. Retrieved 2022-06-24, from http://link.springer.com/10.1007/978-3-8348-9698-8 doi: 10.1007/978-3-8348-9698-8

France, R., & Rumpe, B. (2014). Model-Driven Development of Complex Software: A Research Roadmap. , 18. Retrieved 2022-06-23, from https://arxiv.org/abs/1409.6620 (Publisher: arXiv Version Number: 1) doi: 10.48550/ARXIV.1409.6620

Friedl, J. E. F. (2006). *Mastering regular expressions* (3rd ed ed.). Sebastapol, CA: O'Reilly. (OCLC: ocm76945355)

Giese, H., Karsai, G., Lee, E., Rumpe, B., & Schätz, B. (Eds.). (2010). *Model-Based Engineering of Embedded Real-Time Systems: International Dagstuhl Workshop, Dagstuhl Castle, Germany, November 4-9, 2007. Revised Selected Papers* (Vol. 6100). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved 2022-06-23, from http://link.springer.com/10.1007/978-3-642-16277-0 doi: 10.1007/978-3-642-16277-0

Géron, A. (2019). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: concepts, tools, and techniques to build intelligent systems* (Second edition ed.). Beijing Boston Farnham Sebastopol Tokyo: O'Reilly.

Harrand, N., Fleurey, F., Morin, B., & Husa, K. E. (2016, October). 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* (pp. 125–135). Saint-malo France: ACM. Retrieved 2021-09-29, from https://dl.acm.org/doi/10.1145/2976767.2976812 doi: 10.1145/2976767.2976812

Hartmann, T., Moawad, A., Fouquet, F., & Le Traon, Y. (2017, September). The Next Evolution of MDE: A Seamless Integration of Machine Learning into Domain Modeling. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)* (pp. 180–180). Austin, TX, USA: IEEE. Retrieved 2021-09-29, from http://ieeexplore.ieee.org/document/8101263/ doi: 10.1109/MODELS.2017.32

Hutter, F., Kotthoff, L., & Vanschoren, J. (Eds.). (2019). *Automated Machine Learning: Methods, Systems, Challenges.* Cham: Springer International Publishing. Retrieved 2022-07-04, from http://link.springer.com/10.1007/978-3-030-05318-5 doi: 10.1007/978-3-030-05318-5

Jeusfeld, M. A. (2009). Metamodel. In L. LIU & M. T. ÖZSU (Eds.), *Encyclopedia of Database Systems* (pp. 1727–1730). Boston, MA: Springer US. Retrieved from https://doi.org/10.1007/978-0-387-39940-9_898 doi: 10.1007/978-0-387-39940-9_898

Jörges, S. (2013). *Construction and Evolution of Code Generators* (Vol. 7747; D. Hutchison et al., Eds.). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved 2022-06-23, from http://link.springer.com/10.1007/978-3-642-36127-2 doi: 10.1007/978-3-642-36127-2

Moin, A., Badii, A., & Günnemann, S. (2021, July). A Model-Driven Engineering Approach to Machine Learning and Software Modeling. *arXiv:2107.02689 [cs]*. Retrieved 2021-09-29, from http://arxiv.org/abs/2107.02689 (arXiv: 2107.02689)

Morin, B., Fleurey, F., Husa, K.-E., & Barais, O. (2016, April). A Generative Middleware for

Heterogeneous and Distributed Services. In *2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)* (pp. 107–116). Venice, Italy: IEEE. Retrieved 2021-09-29, from http://ieeexplore.ieee.org/document/7497437/ doi: 10.1109/CBSE.2016.12

Nyamsi, E. A. (2020). *IT-Lösungen auf Basis von SysML und UML: Anwendungsentwicklung mit Eclipse UML Designer und Eclipse Papyrus*. Wiesbaden: Springer Fachmedien Wiesbaden. Retrieved 2022-06-23, from http://link.springer.com/10.1007/978-3-658-29057-3 doi: 10.1007/978-3-658-29057-3

OMG. (2016, October). *Meta Object Facility, v2.5.1* (Standard). Object Management Group (OMG). Retrieved 2022-06-26, from https://www.omg.org/spec/MOF/

Possatto, M. A., & Lucrédio, D. (2015, November). Automatically propagating changes from reference implementations to code generation templates. *Information and Software Technology*, *67*, 65–78. Retrieved 2022-07-09, from https://linkinghub.elsevier.com/retrieve/pii/S0950584915001226 doi: 10.1016/j.infsof.2015.06.009

Raschka, S., & Mirjalili, V. (2021). *Machine learning mit Python und Keras, TensorFlow 2 und Scikit-learn: das umfassende Praxis-Handbuch für data science, deep learning und predictive analytics* (3., aktualisierte und erweiterte Auflage ed.; K. Lorenzen, Trans.). Frechen: mitp.

Rodrigues da Silva, A. (2015, October). Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, *43*, 139–155. Retrieved 2022-06-23, from https://linkinghub.elsevier.com/retrieve/pii/S1477842415000408 doi: 10.1016/j.cl.2015.06.001

Sendall, S., & Kozaczynski, W. (2003, September). Model transformation: the heart and soul of model-driven software development. *IEEE Software*, *20*(5), 42–45. Retrieved 2022-06-16, from http://ieeexplore.ieee.org/document/1231150/ doi: 10.1109/MS.2003.1231150

Sillitto, H., Martin, J., McKinney, D., Griego, R., Dori, D., Krob, D., … Jackson, S. (2019, January). *Systems Engineering and System Definitions, Version 1.0* (Tech. Rep.). INCOSE. Retrieved 2022-06-13, from https://www.incose.org/docs/default-source/default-document-library/incose-se-definitions-tp-2020-002-06.pdf?sfvrsn=b1049bc6_0

Syriani, E., Luhunu, L., & Sahraoui, H. (2018, June). Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures*, *52*, 43–62. Retrieved 2022-06-16, from https://linkinghub.elsevier.com/retrieve/pii/S1477842417301239 doi: 10.1016/j.cl.2017.11.003

Vogel-Heuser, B., Schütz, D., Frank, T., & Legat, C. (2014, October). Model-driven engineering of Manufacturing Automation Software Projects – A SysML-based approach. *Mechatronics*, *24*(7), 883–897. Retrieved 2022-06-23, from https://linkinghub.elsevier.com/retrieve/pii/S0957415814000853 doi: 10.1016/j.mechatronics.2014.05.003

Winzer, P. (2016). *Generic Systems Engineering: Ein methodischer Ansatz zur Komplexitätsbewältigung*. Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved 2022-06-23, from https://link.springer.com/10.1007/978-3-662-52893-8 doi: 10.1007/

978-3-662-52893-8

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved 2022-06-19, from http://link.springer.com/10.1007/978-3-642-29044-2 doi: 10.1007/978-3-642-29044-2

Wolny, S., Mazak, A., Carpella, C., Geist, V., & Wimmer, M. (2020, January). Thirteen years of SysML: a systematic mapping study. *Software and Systems Modeling*, *19*(1), 111–169. Retrieved 2022-06-23, from http://link.springer.com/10.1007/s10270-019-00735-y doi: 10.1007/s10270-019-00735-y

A. Appendix A

Default mapping configuration

```
{
  "trimEmptyLines": true,
  "blockedStereotypes": [
    "BlackBox_Storage",
    "Text_File"
  ],
  "blockedNames": [
    "TestStorage"
  ],
  "constants": {
    "TRAIN": "train",
    "TEST": "test",
    "PREDICT": "pre",
    "MODEL": "mod",
    "X": "X",
    "Y": "y"
  },
  "stereotypeMappings": {
    "ML_Attribute_Input": {
      "template": "attribute_input.vm",
      "properties": {
        "MappedToName": "mapName"
      },
      "modelCommands": {}
    },
    "Datetime": {
      "template": "datetime.vm",
      "properties": {
        "DatetimeFormat": "old_format"
      },
      "modelCommands": {}
    },
    "BlackBox_Storage": {
      "template": "blackbox_storage.vm",
      "properties": {
        "VariableName": "varname"
      },
      "modelCommands": {}
    },
    "Text_File": {
      "template": "text_file.vm",
      "properties": {
        "Path": "path",
        "Encoding": "enc"
      },
      "modelCommands": {}
    },
    "CSV": {
      "template": "csv_load.vm",
      "properties": {
        "Delimiter": "delim",
        "SkipNrOfLines": "skip",
        "GenerateTimestamp": "GenerateTimestamp"
      },
      "modelCommands": {}
    },
    "DateConversion": {
      "template": "date_conversion.vm",
      "properties": {
        "Output_Format": "new_format"
      },
      "modelCommands": {
        "CONNECTED.Datetime.PROPNAME": "d"
      }
    },
    "Encoding": {
      "template": "label_encode.vm",
      "properties": {
        "ToEncode": "feature"
      },
      "modelCommands": {
        "CONNECTED.DataFrame_Merge.NAME": "df"
      }
    },
    "DataFrame_Merge": {
      "template": "dataframe_merge.vm",
      "properties": {
        "MergeOn[0].OWNER.VariableName": "one",
        "MergeOn[1].OWNER.VariableName": "two",
        "MergeOn[0]": "left",
        "MergeOn[1]": "right",
        "How": "how"
      },
      "modelCommands": {
        "THIS.BLOCK.NAME": "new_name"
      }
    },
    "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"
      }
    },
    "Regression": {
      "template": "regression.vm",
      "properties": {
        "Algorithm": "algo"
      },
      "modelCommands": {
        "THIS.BLOCK.NAME": "model_name",
        "CONNECTED.Train_Test_Split.NAME": "split"
      }
    },
    "Predict": {
      "template": "predict.vm",
      "properties": {},
      "modelCommands": {
        "CONNECTED.Regression.NAME": "pred_name",
        "CONNECTED.Train_Test_Split.NAME": "train_test"
      }
    },
    "Metrics": {
      "template": "metrics.vm",
      "properties": {
        "Text": "txt"
      },
      "modelCommands": {}
    },
    "MeanAbsoluteError": {
      "template": "mae.vm",
      "properties": {},
      "modelCommands": {
        "CONNECTED.Predict.NAME": "pred_name"
      }
    }
  },
  "nameMappings": {
    "CSV_1": {
      "template": "name_csv.vm",
      "properties": {
        "VariableName": "varname",
        "Path": "p",
        "Encoding": "e",
        "Delimiter": "d",
        "SkipNrOfLines": "s",
        "GenerateTimestamp": "GenerateTimestamp"
      },
      "modelCommands": {}
    }
  }
}
```

Code Fragment A.1: Default configuration mapping JSON file

B. Appendix B

Velocity templates

```
THIS SHOULD NOT BE CALLED
```

Code Fragment B.1: Velocity template “attribute_input.vm”

```
old_format = ${old_format}
```

Code Fragment B.2: Velocity template “datetime.vm”

```
${(varname,"ABCD")}
```

Code Fragment B.3: Velocity template “blackbox_storage.vm”

```
${(varname,"myTest")} = ("${(path,"PLACEHOLDER")}", "${(enc,"UTF-8")}")
```

Code Fragment B.4: Velocity template “text_file.vm”

```
import pandas as pd

${varname} = pd.read_csv("${path}", sep="${delim}", encoding="${(enc,"UTF-8")}
    ↪ ", skiprows=${skip})
```

Code Fragment B.5: Velocity template “csv_load.vm”

```
import pandas as pd
path = "${p}"
separator = "${d}"
enc = "${e}"
${varname} = pd.read_csv(path, sep=separator, encoding=enc, skiprows=${s})
```

Code Fragment B.6: Velocity template “name_csv.vm”

```
import datetime

${varname}["${d}"] = ${varname}["${d}"].apply(lambda old_date, old_format,
    ↪ new_format: datetime.datetime.strptime(old_date, old_format).strftime(
    ↪ new_format), args=("${old_format}", "${new_format}"))
```

Code Fragment B.7: Velocity template “date_conversion.vm”

```
import from sklearn.preprocessing import LabelEncoder

${df}["${feature}"]=LabelEncoder().fit_transform(${df}["${feature}"])
```

Code Fragment B.8: Velocity template “label_encode.vm”

```
import pandas as pd

${new_name} = pd.merge(left=${one}, right=${two}, left_on="${left}", right_on="
    ↪ ${right}", how="${how}")
```

Code Fragment B.9: Velocity template “dataframe_merge.vm”

```
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})
```

Code Fragment B.10: Velocity template “train_test_split.vm”

```
from #if (${algo} == "RandomForestRegressor")
from sklearn.ensemble import RandomForestRegressor
#else
from sklearn.tree import DecisionTreeRegressor
#end

${MODEL}_${model_name}=${algo}(random_state=1)
${MODEL}_${model_name}.fit(${split}_${TRAIN}_${X}, ${split}_${TRAIN}_${Y})
```

Code Fragment B.11: Velocity template “regression.vm”

```
${PREDICT}_${pred_name} = ${MODEL}_${pred_name}.predict(${train_test}_${TEST}_$
    ↪ {X})
```

Code Fragment B.12: Velocity template “predict.vm”

```
print("${txt}")
```

Code Fragment B.13: Velocity template “metrics.vm”

```
from sklearn.metrics import mean_absolute_error

mae = mean_absolute_error(${split_name}_${TEST}_${Y}, ${PREDICT}_${pred_name})
print("Mean Absolute Error: %f" % mae)
print("${txt}")
```

Code Fragment B.14: Velocity template “mae.vm”

C. Appendix C

Output IPYNB file

```
{
    "cells": [
        {
            "id": "af34b8e8-f60d-4028-a800-7c4522a5337b",
            "cell_type": "markdown",
            "metadata": {},
            "source": [
                "# Import section\nImports for notebook"
            ]
        },
        {
            "id": "3497b42f-0793-4a17-9373-bd1f0dcdf2ae",
            "cell_type": "code",
            "execution_count": null,
            "metadata": {},
            "source": [
                "from sklearn.ensemble import
   ↪ RandomForestRegressor\nfrom sklearn.metrics import
   ↪ mean_absolute_error\nimport pandas as pd\nfrom sklearn.
   ↪ preprocessing import LabelEncoder\nfrom sklearn.tree import
   ↪ DecisionTreeRegressor\nfrom sklearn.model_selection import
   ↪ train_test_split\nimport datetime"
            ],
            "outputs": []
        },
        {
            "id": "49eea9eb-e0e1-4ae0-a182-71429fe17f5e",
            "cell_type": "markdown",
            "metadata": {},
            "source": [
                "<div>\n<h1>Dataset load</h1>\n\n<p>Load the
   ↪ dataset(s)</p>\n</div>\n"
            ]
        },
        {
            "id": "96aebc81-4f72-44bc-8f43-6c5426084cb7",
            "cell_type": "code",
            "execution_count": null,
            "metadata": {},
            "source": [
                "path = \"weather_split_complex_one.csv\"\
↪ nseparator = \",\"\nenc = \"UTF-8\"\ndf_one = pd.read_csv(
↪ path, sep=separator, encoding=enc, skiprows=0)",
                "\n"
            ],
            "outputs": []
        },
        {
            "id": "2fc30982-2654-4761-95fc-a1f0fb978fb6",
            "cell_type": "code",
            "execution_count": null,
            "metadata": {},
            "source": [
                "df_two = pd.read_csv(\"weather_split_complex_two.
↪ csv\", sep=\";\", encoding=\"UTF-8\", skiprows=0)"
            ],
            "outputs": []
        },
        {
            "id": "52f49204-734c-4de0-9f25-d770e05672ac",
            "cell_type": "markdown",
            "metadata": {},
            "source": [
                "<div>\n<h1>Data Preprocessing: Date Conversion&
↪ nbsp;</h1>\n</div>\n\n<p>&nbsp;</p>\n"
            ]
        },
        {
            "id": "c59bd8ee-a07b-4d3a-aff4-bf0baa029b15",
            "cell_type": "code",
            "execution_count": null,
            "metadata": {},
            "source": [
                "df_one[\"date\"] = df_one[\"date\"].apply(lambda
↪ old_date, old_format, new_format: datetime.datetime.strptime
↪ (old_date, old_format).strftime(new_format), args=(\"%d-%m-%
↪ Y\", \"%Y-%m-%d\"))"
            ],
            "outputs": []
        },
        {
            "id": "18f3286a-214f-4893-9451-84226e77a491",
            "cell_type": "markdown",
            "metadata": {},
            "source": [
                "<div>\n<h1>Dataframe Merge</h1>\n\n<p>Merge
↪ dataframes</p>\n</div>\n"
            ]
        },
        {
            "id": "4be890a1-b15f-4c83-b755-9647afd7b813",
            "cell_type": "code",
            "execution_count": null,
            "metadata": {},
            "source": [
                "Merge_DF = pd.merge(left=df_one, right=df_two,
↪ left_on=\"date\", right_on=\"date_date\", how=\"inner\")"
            ],
            "outputs": []
        },
        {
            "id": "034b1a4f-75c7-4e80-ba41-8766caeaa4ea",
            "cell_type": "markdown",
            "metadata": {},
            "source": [
                "<div>\n<h1>Data Pre-Processing: Label Encoding</
↪ h1>\n</div>\n"
            ]
        },
        {
            "id": "4756bcdc-7400-4031-a824-fd211ab053a8",
            "cell_type": "code",
            "execution_count": null,
            "metadata": {},
            "source": [
                "Merge_DF[\"weather\"]=LabelEncoder().
↪ fit_transform(Merge_DF[\"weather\"])"
            ],
            "outputs": []
        },
        {
            "id": "0a1be93a-7088-45a8-ad8b-2211db710739",
            "cell_type": "code",
            "execution_count": null,
            "metadata": {},
            "source": [
                "X=Merge_DF[[\"precipitation\", \"temp_max\", \"
↪ temp_min\", \"wind\"]]\ny=Merge_DF.weather\
↪ nTrainSplit_train_X, TrainSplit_test_X, TrainSplit_train_y,
↪ TrainSplit_test_y = train_test_split(X, y,random_state = 0,
↪ train_size=0.7)"
            ],
            "outputs": []
        },
        {
            "id": "d6206ebf-4c77-41ad-8639-19f801c0f468",
            "cell_type": "markdown",
            "metadata": {},
            "source": [
                "<div>\n<h2>Decision Tree Regressor</h2>\n\n<p>
↪ Creation of decision tree regressor</p>\n</div>\n"
            ]
        },
        {
            "id": "821449ed-c40d-4d2e-b18a-c8d6a4794a9f",
            "cell_type": "code",
            "execution_count": null,
            "metadata": {},
            "source": [
                "mod_ML_2=DecisionTreeRegressor(random_state=1)\
↪ nmod_ML_2.fit(TrainSplit_train_X, TrainSplit_train_y)"
            ],
            "outputs": []
        },
        {
            "id": "1f4fc5bd-c199-4309-908d-baa0d6a64e35",
            "cell_type": "markdown",
            "metadata": {},
            "source": [
                "<div>\n<div>\n<h1>Create Model</h1>\n\n<p>Section
```

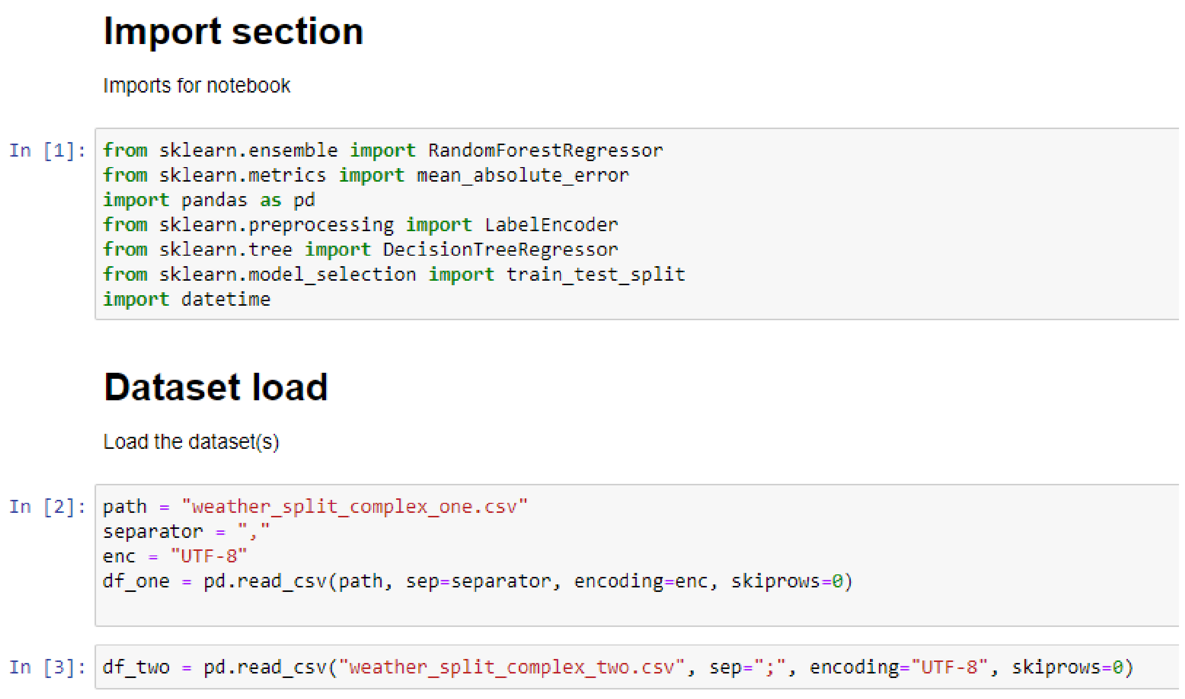
```
↪  where models are created</p>\n</div>\n\n<h2>Random Forest
↪ Regressor</h2>\n</div>\n"
        ]
    },
    {
        "id": "016f4fde-523d-4bd4-af67-df85ed0e9491",
        "cell_type": "code",
        "execution_count": null,
        "metadata": {},
        "source": [
            "mod_ML_1=RandomForestRegressor(random_state=1)\
↪ nmod_ML_1.fit(TrainSplit_train_X, TrainSplit_train_y)"
        ],
        "outputs": []
    },
    {
        "id": "523794e6-faa9-4669-9ece-f5e077cd3bf9",
        "cell_type": "markdown",
        "metadata": {},
        "source": [
            "<div>\n<h1>Prediction</h1>\n\n<p>Section where
↪ models are used to predict</p>\n</div>\n\n<div>\n<h2>Random&
↪ nbsp;Forest Regressor</h2>\n</div>\n\n<p>&nbsp;</p>\n"
        ]
    },
    {
        "id": "7195d57f-c2c0-4be4-84ca-2ac4108f13ac",
        "cell_type": "code",
        "execution_count": null,
        "metadata": {},
        "source": [
            "pre_ML_1 = mod_ML_1.predict(TrainSplit_test_X)"
        ],
        "outputs": []
    },
    {
        "id": "0bcb7847-bbe5-4f6d-921d-643c732df961",
        "cell_type": "markdown",
        "metadata": {},
        "source": [
            "<div>\n<h2>Decision Tree Regressor</h2>\n</div>\n
↪ "
```

```
            ]
        },
        {
            "id": "810a9ccf-2f3a-40cd-8ee7-5b79cddc6ad1",
            "cell_type": "code",
            "execution_count": null,
            "metadata": {},
            "source": [
                "pre_ML_2 = mod_ML_2.predict(TrainSplit_test_X)"
            ],
            "outputs": []
        },
        {
            "id": "e2ade2ed-6871-4941-b13b-e15e276a4ea8",
            "cell_type": "markdown",
            "metadata": {},
            "source": [
                "<div>\n<h1>Metrics</h1>\n\n<p>Section for metrics
↪  for the used model(s)</p>\n</div>\n\n<div>\n<h2>Random
↪ Forest Regressor</h2>\n</div>\n\n<p>&nbsp;</p>\n"
            ]
        },
        {
            "id": "f0ffb298-dfa5-48ec-9ee6-931fc9c3ee41",
            "cell_type": "code",
            "execution_count": null,
            "metadata": {},
            "source": [
                "mae = mean_absolute_error(TrainSplit_test_y,
↪ pre_ML_1)\nprint(\"Mean Absolute Error: %f\" % mae)\nprint(\
↪ "First mean absolute error\")"
            ],
            "outputs": []
        },
        {
            "id": "c7df0e22-c1dd-4037-b109-40e523ca51f4",
            "cell_type": "markdown",
            "metadata": {},
            "source": [
                "<div>\n<h2>Decision Tree Regressor</h2>\n</div>\n
↪ "
            ]
        },
        {
            "id": "ca56a028-2f1e-46d3-a0e3-0aced97ec893",
            "cell_type": "code",
            "execution_count": null,
            "metadata": {},
            "source": [
                "mae = mean_absolute_error(TrainSplit_test_y,
   ↪ pre_ML_2)\nprint(\"Mean Absolute Error: %f\" % mae)\nprint(\
   ↪ "Second mean absolute error\")"
            ],
            "outputs": []
        }
    ],
    "nbformat": 4,
    "nbformat_minor": 5,
    "metadata": {
        "kernelspec": {
            "name": "python3",
            "language": "python",
            "display_name": "Python 3"
        },
        "language_info": {
            "pygments_lexer": "ipython3",
            "nbconvert_exporter": "python",
            "codemirror_mode": {
                "name": "ipython",
                "version": 3
            },
            "name": "python",
            "mimetype": "text/x-python",
            "file_extension": ".py",
            "version": "3.7.12"
        }
    }
}
```

Code Fragment C.1: Output IPYNB file

D. Appendix D

Output IPYNB in figures



Import section

Imports for notebook

```
In [1]: from sklearn.ensemble import RandomForestRegressor
        from sklearn.metrics import mean_absolute_error
        import pandas as pd
        from sklearn.preprocessing import LabelEncoder
        from sklearn.tree import DecisionTreeRegressor
        from sklearn.model_selection import train_test_split
        import datetime
```

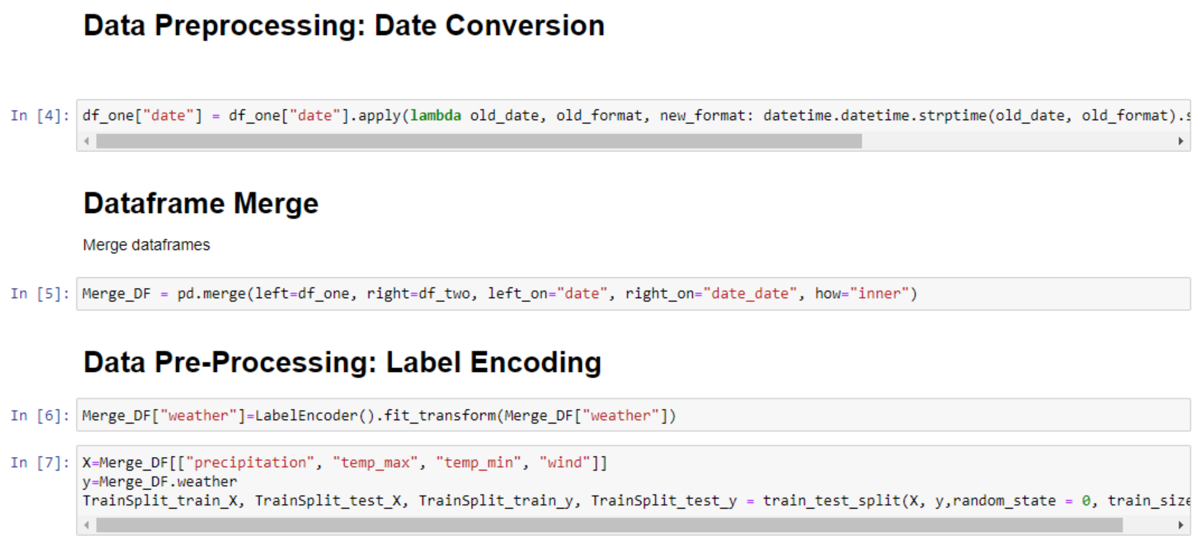
Dataset load

Load the dataset(s)

```
In [2]: path = "weather_split_complex_one.csv"
        separator = ","
        enc = "UTF-8"
        df_one = pd.read_csv(path, sep=separator, encoding=enc, skiprows=0)
```

```
In [3]: df_two = pd.read_csv("weather_split_complex_two.csv", sep=";", encoding="UTF-8", skiprows=0)
```

Figure D.1.: Output IPYNB import and CSV load



Data Preprocessing: Date Conversion

```
In [4]: df_one["date"] = df_one["date"].apply(lambda old_date, old_format, new_format: datetime.datetime.strptime(old_date, old_format).s
```

Dataframe Merge

Merge dataframes

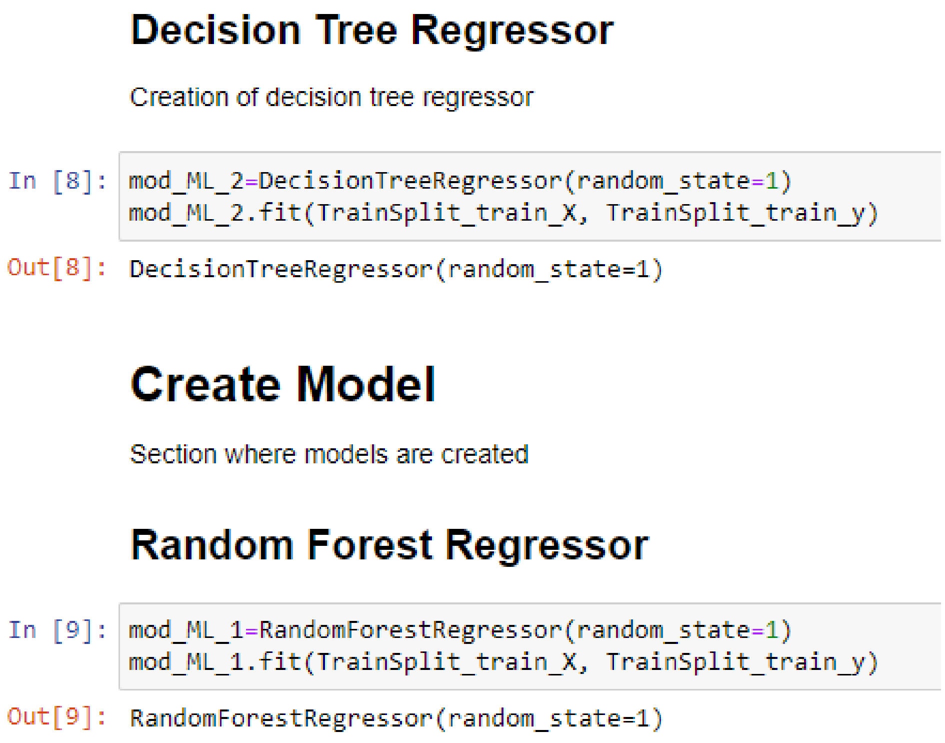
```
In [5]: Merge_DF = pd.merge(left=df_one, right=df_two, left_on="date", right_on="date_date", how="inner")
```

Data Pre-Processing: Label Encoding

```
In [6]: Merge_DF["weather"]=LabelEncoder().fit_transform(Merge_DF["weather"])
```

```
In [7]: 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
```

Figure D.2.: Output IPYNB data preprocessing and dataframe merge



Decision Tree Regressor

Creation of decision tree regressor

```
In [8]:  mod_ML_2=DecisionTreeRegressor(random_state=1)
         mod_ML_2.fit(TrainSplit_train_X, TrainSplit_train_y)

Out[8]:  DecisionTreeRegressor(random_state=1)
```

Create Model

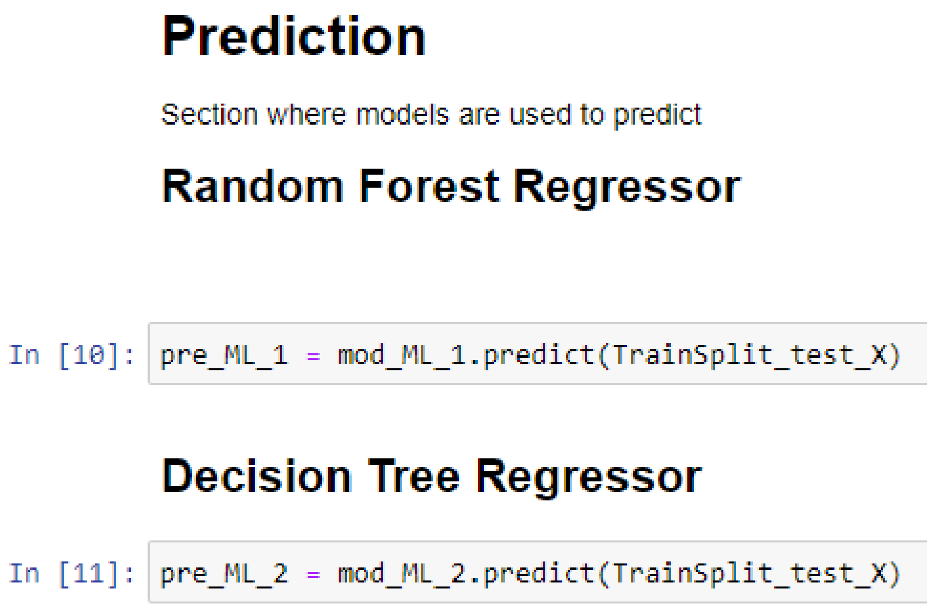
Section where models are created

Random Forest Regressor

```
In [9]:  mod_ML_1=RandomForestRegressor(random_state=1)
         mod_ML_1.fit(TrainSplit_train_X, TrainSplit_train_y)

Out[9]:  RandomForestRegressor(random_state=1)
```

Figure D.3.: Output IPYNB creation of ML model



Prediction

Section where models are used to predict

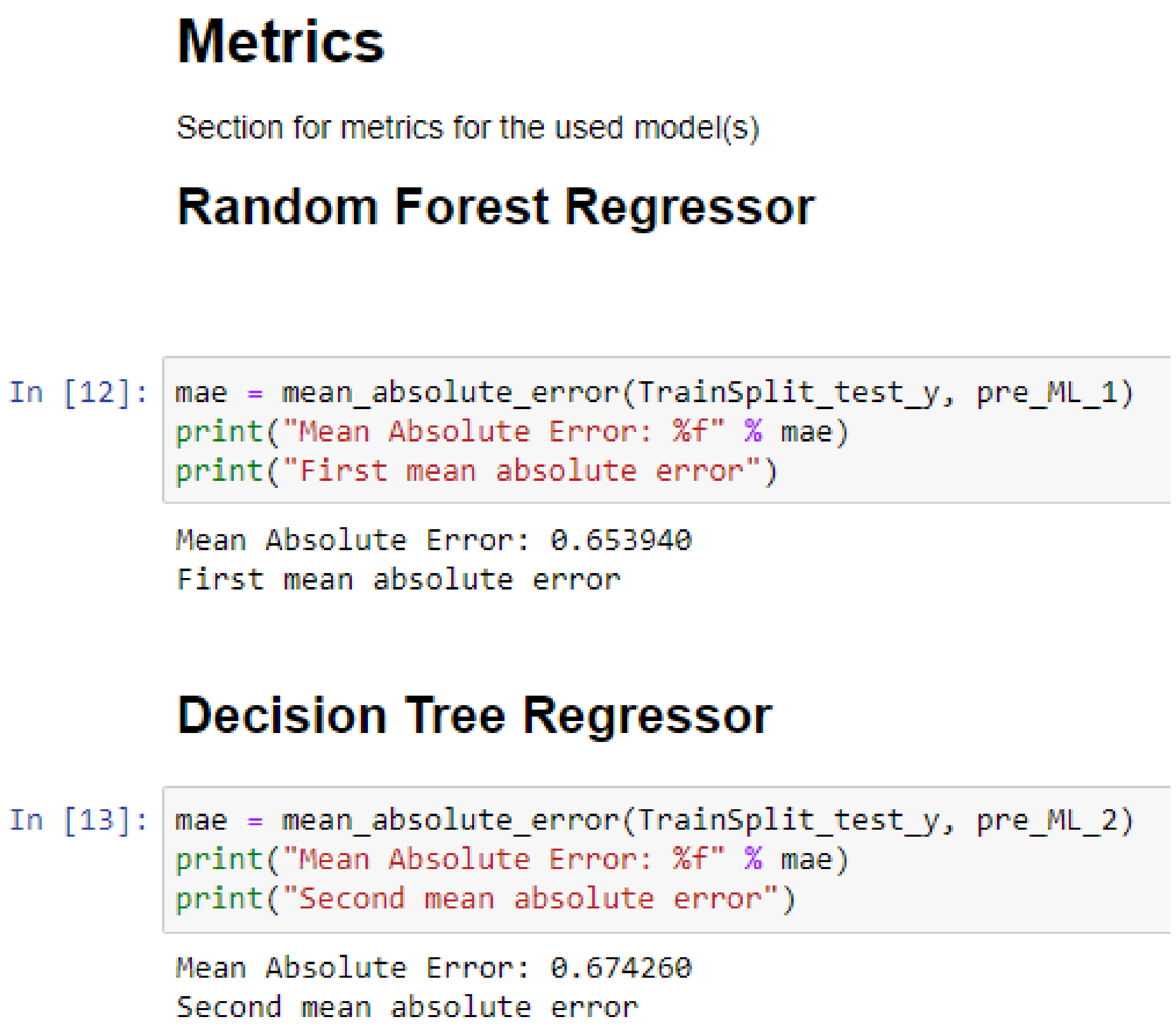
Random Forest Regressor

```
In [10]:  pre_ML_1 = mod_ML_1.predict(TrainSplit_test_X)
```

Decision Tree Regressor

```
In [11]:  pre_ML_2 = mod_ML_2.predict(TrainSplit_test_X)
```

Figure D.4.: Output IPYNB prediction



Metrics

Section for metrics for the used model(s)

Random Forest Regressor

In [12]:
```
mae = mean_absolute_error(TrainSplit_test_y, pre_ML_1)
print("Mean Absolute Error: %f" % mae)
print("First mean absolute error")
```

```
Mean Absolute Error: 0.653940
First mean absolute error
```

Decision Tree Regressor

In [13]:
```
mae = mean_absolute_error(TrainSplit_test_y, pre_ML_2)
print("Mean Absolute Error: %f" % mae)
print("Second mean absolute error")
```

```
Mean Absolute Error: 0.674260
Second mean absolute error
```

Figure D.5.: Output IPYNB metrics

Statement of Affirmation

I hereby declare that all parts of this thesis were exclusively prepared by me, without using resources other than those stated above. The thoughts taken directly or indirectly from external sources are appropriately annotated. This thesis or parts of it were not previously submitted to any other academic institution and have not yet been published.

Dornbirn, 10.07.2022 Matthias Rupp