Bauhaus-Universität Weimar
Faculty of Media
Degree Progamme Computer Science and Media

DEVELOPING A DOMAIN SPECIFIC LANGUAGE FOR THE EVOLVING FIELD OF MACHINE LEARNING

MASTER'S THESIS

Aaron Solbach Matriculation Number 112201

b. 10.10.1990 in Itzehoe

First Referee: Prof. Dr.-Ing. Norbert Siegmund

Second Referee: Prof. Dr. Benno Stein

Submission date: January, 2018

ABSTRACT

Machine Learning (ML) is an area of research from which many other domains can benefit. However, access to ML is difficult, as extensive programming skills are required. In this paper, we use Jetbrains' Meta Programming System (MPS) to develop a domain-specific language (DSL) that can easily be adapted to the notations of other disciplines. The main features of this language are (1) variability mechanisms that facilitate extensibility and allow the scope of the language to be adapted, as well as (2) direct feedback that provides information about the system state. Within a user study, the DSL is considered to be highly user-friendly compared to well-known ML libraries.

ZUSAMMENFASSUNG

Maschinelles Lernen (ML) ist ein Forschungsgebiet von dem viele andere Domänen profitieren können. Der Zugang zu ML ist jedoch schwer, da weitreichende Programmierkenntnisse vorausgesetzt werden. In der vorliegenden Arbeit entwickeln wir daher mittels Jetbrains' Meta Programming System (MPS) eine Domänen-Spezifische Sprache (DSL), die sich leicht an die Notationen anderer Fachrichtung anpassen lässt. Hauptmerkmale dieser Sprache sind Variabilitätsmechanismen, die die Erweiterbarkeit erleichtern und den Umfang der Sprache anpassen lassen, sowie ein direktes Feedback, das Aufschluss über den Systemzustand gibt. Innerhalb einer Nutzerstudie wird der DSL im Vergleich zu bekannten ML Bibliotheken eine hohe Benutzerfreundlichkeit zugesprochen.

CONTENTS

1 INTRODUCTION 1

I BACKGROUND

2 DOMAIN-SPECIFIC LANGUAGES 5
- 2.1 Distinction from General-Purpose Languages 5
- 2.2 Language Oriented Programming Paradigm 6
- 2.3 DSL Specifications of MLE 7
- 2.4 Domain-Specific Languages for Machine-Learning . . . 7
- 2.5 Requirements for MLE 8

3 PRODUCT LINE ENGINEERING 11
- 3.1 Approaches to Implement Software Variability 11
- 3.2 Visualizing Product Lines 13
- 3.3 Product Lines in MPS 13

4 LANGUAGE ENGINEERING WITH JETBRAINS MPS 15
- 4.1 Projectional Editing 15
- 4.2 Elements of Programming Languages 16
- 4.3 The MPS Project Structure 17
- 4.4 Defining Concepts in MPS 18
  - 4.4.1 Structure Aspect 19
  - 4.4.2 Editor Aspect 19
  - 4.4.3 Constraints and Type-System Aspects 21
  - 4.4.4 Actions and Intentions Aspects 23
  - 4.4.5 Behavior and Custom Aspects 24
  - 4.4.6 Generator and TextGen Aspects 25
- 4.5 MPS Frameworks 26

II CONTRIBUTION

5 VARIABLE LANGUAGE COMPOSITION 31
- 5.1 Usage of the Composition Language 32
  - 5.1.1 Interactions with the Language Component Tree 32
  - 5.1.2 Registration of Language Components 33
- 5.2 Implementation of the Component-Description 34
  - 5.2.1 Structure 34
  - 5.2.2 Constraints 36
  - 5.2.3 Editor 38
  - 5.2.4 Listeners and Behavior 39
  - 5.2.5 Components-Class 41
- 5.3 Implementation of the Components-Tree Interface . . . 43
  - 5.3.1 Structure of the Component Tree and Nodes . . 43
  - 5.3.2 Building thee Tree-Interface 44
  - 5.3.3 Editor of the Component Nodes 45
  - 5.3.4 Loading a Component Composition 46

5.4 Efficient Expansion Interface in MLE 49
6 USABILITY DRIVEN LANGUAGE DESIGN 51
6.1 Visibility of Actions, Options, and Objects 52
6.1.1 Language Optimization for Code-Suggestions . 52
6.1.2 Addressing of Items as Variables 56
6.2 Feedback 58
6.2.1 Support Programming with Prompt Feedback . 58
6.2.2 Piecewise Feedback from Sections 61
6.3 Prompt Computation of Object-States 64
6.3.1 Approaches to Track the Object-State 64
6.3.2 Implementation Details 68

III EVALUATION

7 MLE USABILITY STUDY 77
7.1 Experimental Design 77
7.1.1 Goal 78
7.1.2 Participants 78
7.1.3 Experimental Material 79
7.1.4 Experiment Variables 82
7.1.5 Design 84
7.1.6 Procedure 84
7.1.7 Deviations during Experiment Execution 85
7.2 Analysis 86
7.2.1 Pre Questionnaire 86
7.2.2 Experiment 86
7.2.3 Post Questionnaire 87
7.3 Discussion 88
7.3.1 Efficiency 88
7.3.2 Usability 90
7.3.3 Study Results 91
8 CONCLUSION 93

IV APPENDIX

A SOURCE-CODE OF CLASS: TREEBUILDER 97
B NIELSEN'S USABILITY HEURISTICS 101
C EXPERIMENT MATERIAL 103
C.1 Procedure 103
C.1.1 Welcome 103
C.1.2 Tutorials Tasks 104
C.2 Tasks 106
C.2.1 Clickbait (Original) 106
C.2.2 Clickbait (English Translation) 108
C.2.3 Brain Data (Original) 110
C.2.4 Brain Data (English Translation) 112

BIBLIOGRAPHY 115

ACRONYMS

AST	Abstract Syntax Tree
DSL	Domain-Specific Language
GPL	General-Purpose Language
IDE	Integrated Development Environment
ML	Machine-Learning
MLE	Machine Learning Evolves
MPS	Meta Programming System
PLE	Product Line Engineering
UI	User Interface

1

INTRODUCTION

Machine-Learning (ML) is the buzzword related to computer-science and technology that is currently grabbing most attention. The term's popularity is not restricted to science but is also used to promote consumer products such as smartphones or cameras. People expect those devices utilizing ML to be smart and learning devices. Aside such exaggerated expectations and misunderstandings, ML provides algorithms that independently improve their performance on learning tasks. The applications are diverse, as are the domains that can benefit from it: Systems biology attempts to understand organisms in their entirety by analyzing large datasets often obtained from computer simulations. Macro sociology focuses on the analysis of social systems and populations at large scale. Quantitative linguistics analyzes structures in language in order to develop an exact scientific theory of languages. These examples are possible applications for ML. They are chosen from subfields of completely unrelated domains, and there are many more applications in both, other subfields and other domains.

However, the usage of ML is complex and difficult to access. Algorithms depend on clean and consistent data. The data must be preprocessed to prevent errors during the further computations. Moreover, numerical features must be extracted so that algorithms can calculate on them. There are numerous ML algorithms that perform variably well depending on their parameterization and the shape of the data. Users require to know what algorithm is suitable for the respective task and how the parameterization affects the outcome. Finally, they require broad programming skills to write programs that integrate all the aforementioned considerations. Experts of domains distant from computer science need to acquire comprehensive expert knowledge, which in terms of content is unrelated to their field of interest. Thus, the field of ML remains closed to those other domains.

Domain-Specific Languages (DSLs) could provide a remedy in this situation. In contrast to General-Purpose Languages (GPLs) such as C or Java, they are constrained to the applications of a single domain. A prominent example is SQL which is used for database administration: statements in SQL are easier to read and understand for non-programmers and constrained to the database domain. As a result, the number of domain-unrelated concepts to be learned is highly reduced in comparison to GPLs.

In this thesis, we present Machine Learning Evolves (MLE), a DSL that aims at opening ML to other domains. Therefore, we outline the background of this work in part 1: We introduce DSLs in chapter 2 and specify the specific requirements for MLE. We will see that variability mechanisms are required in order to customize the scope of the DSL and to provide efficient expansion interfaces. Such mechanisms are the subject of Product Line Engineering (PLE) which we briefly introduce in chapter 3. ML is developed and used in Jetbrains Meta Programming System (MPS), an open source language workbench. We provide an overview of the workflow with MPS in chapter 4.

In the second part, we present our contribution to a modularized DSL for ML. We explain the adaption and implementation of variability mechanisms in MLE in chapter 5 and demonstrate how extensions can be developed efficiently. The main goal regarding the usability of the DSL is to keep the user informed of the program's state and available actions during programming. In chapter 6 we explain how we implement this behavior into MLE by tracking the state of objects.

The third part of this thesis covers the evaluation out our contribution. We perform a user study which reveals that the usability in ML is perceived better as in the control condition with respect to visibility, error prevention, and minimized memory load. The experiment is reported in chapter 7. Finally, we conclude our work in chapter 8

With MLE we focus on the following two key aspects:

1. **Variable Language Composition** Machine Learning is a broad and evolving field of research in which users may still be overwhelmed despite the streamlining that a DSL could achieve. Thus, fitting the product to the concrete needs of users is very important. As stated earlier, MLE is meant to be a modularized base-language on which can be expended, and from which specialized languages can be composed. Software Product Line Engineering offers solutions for variable language compositions that we attempt to integrate into MLE.

2. **Usability driven Language Design** The design of MLE strives on usability that exceeds limiting the command set to ML. MLE developers should always be able to maintain an overview of how a program works and what effect specific lines of code have. We attempt to achieve this by two means: On the one hand, helpful *feedback* should be given not only after execution but already during programming. On the other hand, the developer should always be informed on his options through the *visibility* of actions and objects.

Part I

BACKGROUND

2

DOMAIN-SPECIFIC LANGUAGES

A DSL is a language dedicated to the accomplishment of tasks within a specific domain. As the main contribution of this thesis is the development of such a language, we introduce the theoretical background on DSL engineering in this chapter. The development of DSLs in the language-workbench of our choice is explained in chapter 4. This theoretical background comprises the distinction between domain-specific and GPLs (see section 2.1), a summary of the language-oriented programming paradigm (see section 2.2) which lays out the benefits of extensible languages, and the specification of MLE based on the different characteristics of DSLs (see section 2.3).

2.1 DISTINCTION FROM GENERAL-PURPOSE LANGUAGES

A DSL is – as the name indicates – a more specific programming language than a GPL with respect to a certain class of tasks. Since there is no binary classification but rather a degree of how domain specific something is, the distinction between both DSLs and GPLs is not sharp. Moreover, Völter et al. (2013) state that even GPLs, such as C or Ruby differ in their relatedness to specific domains, which is one main for the great number of different GPLs. They describe that, for instance, C allows developers to influence the memory layout which is important when low-level memory mapped devices are involved while Ruby's string manipulation features as well as the closures are very useful when building asynchronous web applications. DSLs are designed to make programming for a specific domain more feasible. This is achieved by two means: constraint and abstraction.

CONSTRAINT In contrast to GPLs, DSLs are often much more constrained. GPLs must be Turing complete, which means that anything that is computable with a Turing machine needs to be implementable here. A DSL can be Turing incomplete which limits the possibilities of what can be achieved by them. This limitation is an actual gain of DSLs: there are less language concepts to think about or choose from.

ABSTRACTION DSLs are much more abstract than GPLs. In GPLs, users must be very concrete in describing how to achieve a certain goal while a DSL can provide general, abstract commands. For instance, SQL (which is a very common example for an effective DSL) allows the user to simply select a column of a table. It is not required to define a loop that repeats an action on each row of the table. DSLs allow users to describe on a more abstract level what they want to be done and not how this could be realized.

2.2 LANGUAGE ORIENTED PROGRAMMING PARADIGM

The idea of Language-Oriented Programing is nothing new. In fact, it has been described by Ward (1994) almost 25 years ago. The aim is to construct a very high-level language enabling an efficient development for a class of domain-specific tasks (i.e. a DSL). Ward proposes to first design the formally specified DSL. Then, domain specific problems are solved in the newly designed language and an execution engine is developed in order to make the programs executable. Ward states that the use of high-level languages enables developers to write much shorter code which also is "easier to read, analyze, understand, and modify".

Note: The order here is irrelevant. In fact, it often helps to implement programs first to become aware of potential flaws in the language design

Dmitriev (2004), the co-founder and president of the software development company Jetbrains, expand on the Language Oriented Programming Paradigm with focus on freedom in development. He states that developers are in theory enabled to create anything that is possible with modern computers. In practice, however, they are dependent on languages and software development tools that can hardly be adjusted to their actual needs in reasonable time. Thus, he pleads for tools and languages that can easily evolved by anyone which also is the motivation for Jetbrains' language workbench called MPS.

Furthermore, Völter (2014) lays out his vision of a programming paradigm, which is called "Generic Tools, Specific Languages", in his doctor thesis. This paradigm combines the ideas of freedom in development with language-oriented programming. He states that current tools are adapted to domains only perfunctorily. It allows only for changes of buttons and functions but neglects the need of modifiable underlying data structures. Generic tools that are extensible and adaptable in all aspects – their function, their user interface, and their underlying data structures – could be fitted closely to all kinds of domain-specific tasks. Moreover, specific languages are not seen as an intermediate step when building a domain-specific application but as the application itself. The premise is that a high-level language that uses domain-specific notations and data structures will be at least as efficient in solving domain-specific tasks as traditional tools with

a graphical user-interface. "Generic Tools, Specific Languages" builds on the definition, reuse, and composition of languages which together define a high-level DSL and adapt a generic tool. A language workbench such as MPS is that generic tool and it is used for implementing both, languages as well as solutions in respective languages.

2.3 DSL SPECIFICATIONS OF MLE

There are two types of DSLs: internal and external ones. Internal DSLs are based on GPLs, which means that they are bound to the syntax of their base language. Although internal DSLs can do a good job to provide a whole different programming experience the restrictions of their roots are always noticeable. The great advantage for internal DSLs is that the surrounding infrastructure is at hand. External DSLs, in contrast, are completely independent. It is not only required to implement the language syntax but also an execution engine which allows users to run programs written with the DSL. On the other hand, the syntax can be built from ground up with the domain in mind.

External DSLs can be subdivided by their type of code representation. Code can be represented textual or projectional. Textual source code can be written in any text editor and simply be shared. However, it needs to be parsed which limits the possibilities when designing the syntax since the parser needs exact indications for the end of a statement (such as a semicolon at the end of a line). Also, it is error-prone and requires the user to learn the concrete syntax of the language. Projectional code representation puts projections in place of text. A projection is a reference to a statement that can be presented in any visual matter like graphs, mathematical formulas, images, or simple text. This way it can better emphasize the program's semantics and hence promote comprehension. External DSLs are built in specialized tools called language workbenches. There are few language workbenches that match the design choices described above (as there are few projectional language workbenches at all).

2.4 DOMAIN-SPECIFIC LANGUAGES FOR MACHINE-LEARNING

The number of ML languages and libraries is large, and it is difficult to remain an overview. There are a several papers that limit themselves to providing an overview of ML languages for specific application areas to ease the entry.

For instance, Portugal, Alencar, and Cowan (2016) introduce several DSLs for ML in Big Data. Overall, they list five DSLs such as OptiML (K. Sujeeth et al., 2011) or ScalOps (Weimer, Condie, and Ramakrish-

nan, 2011) which focus on parallel and cloud calculation respectively. Also, popular ML-frameworks developed in python, such as TensorFlow (Abadi et al., 2016) and GraphLab[1] (Low et al., 2014) are listed.

However, most of the languages and frameworks we have reviewed facilitate tasks ranging from reading data from multiple sources to calculating on distributed systems, without reducing the possibilities in programming. Users still must learn programming in general in order to succeed with ML. Internal DSLs, such as OptiML which is based on Scala, cover the complexity of their base language but still require knowledge on general control structures. To our knowledge there are no external ML-specific languages that focus on opening ML to domains unrelated to Computer Science.

2.5 REQUIREMENTS FOR MLE

A DSL that aims at opening ML to other domains has special requirements. By definition, the DSL must not attempt to provide one unified language for all domains with interest in ML, or it would not be domain-specific anymore. Thus, the DSL should be thought of as a base-language for a family of languages which facilitates ML. There are three requirements for the base-language: First, the statements and concepts of the language should be easily adaptable to any domain's vocabulary and notations so that experts can express themselves in a way they are already used to. Second, the DSL's complexity should be adaptable to the needs of the specific domain so that only the required subset of the range of feature is accessible. Third, the DSL needs to be efficiently extensible to address the rapid evolution in the field of ML which might be further driven be the involvement of other domains.

With MLE, we do not try to provide a language that covers all aspects of ML, but to lay ground for a language that can flexibly adjusts to evolutionary changes in its field. Also, we do not attempt to develop another DSL but provide an intermediate layer between existing languages and libraries on the one hand, and users remote from computer science on the other.[2] Such users should not be required to learn programming in order to solve their domain tasks but use familiar notations. Thus, we decided to create an external language with no constraints to the syntax of a base language. Furthermore, projectional editing is preferred over textual editing, since it reduces

1 GraphLab is now continued as TuriCreate, https://github.com/apple/turicreate

2 In fact, the ML calculations run on a local Python server which utilizes Scikit-Learn (Pedregosa et al., 2011) – an ML-framework that features algorithms for classification, regression, and clustering. In future, the server should also be able to access other ML libraries.

the possibilities to write incorrect code. Also, it allows the use of more complex visual representations such as tables or graphs.

Jetbrains' MPS is known as a state-of-the-art language workbench that is actively supported not only by Jetbrains itself but also by other software development companies that rely on the open source platform. Also, MPS was already used to build a prototype in advance to this thesis. Accordingly, it is the language workbench of choice here. Of the few alternatives such as xText (Efftinge and Völter, 2006) or Kermeta (Drey et al., 2009), none offers projectional editing.

3

PRODUCT LINE ENGINEERING

Product Line Engineering (PLE) is a paradigm in software engineering that targets the development of products from core-components rather building them from scratch (Kang, Lee, and Donohoe, 2002). Since ML is a rapidly growing field of research and applications are widely diversified, MLE is designed to provide a range of components from which a domain-related and task-specific language can be created. The result is a family of ML-language dialects that share the same base-language, or a product line of ML-languages. PLE is used to achieve the modularization and composition of MLE.

The concept of modularizing languages into components[1] is well known from the field of domain-analysis. They represent clearly distinguishable attributes that directly affect the end-user of the product (C. Kang et al., 1990). With respect to DSLs, each component corresponds to a set of functions that can be added to the base-language. By selecting only required components, the language can be fitted closely to the specific task.

In this chapter, we outline common approaches to the implementation of software variability in section 3.1. Then, we present the component model in section 3.2 which is a well-known visualization of product lines and utilized for the selection of language components in MLE. Finally, we present work related to PLE in Meta Programming System (MPS) (section 3.3) and discuss why these approaches are not suitable for this thesis.

3.1 APPROACHES TO IMPLEMENT SOFTWARE VARIABILITY

In their book, Apel et al. (2016) provide an overview of different variability mechanisms ranging from simple parameter-based to complex collaboration-based approaches. Base on this book, we present a selection of such approaches in this section. Subsequently, we outline the decision for the variability engineering strategy followed in MLE based on the prerequisites brought by the usage of MPS.

1 Regarding product-lines, the term "feature" is used in literature. Due to the confusion with features in ML we use the term "component" in this thesis.

Parameter-Based Variability

The program code contains all possible variants and conditionals such as if-statements or switches. They are used to alter the control flow at runtime. The parameters can be passed in various ways: as global variables or as parameters which are passed to methods. There are also compiler-directives that work accordingly and thereby remove code of undesired variants at compile time. So-called ifdefs (derived from the directive #ifdef), common in C and C-based languages, are as popular that the name is often used as synonym for the parameter-based mechanism. The advantage of parameter-based variability is that it is easy to understand and implement. On the other hand, the resulting code is difficult to comprehend, and the approach encourages improper ad-hoc implementation.

Software Design-Patterns

There are many well-known software design patterns which are used to describe and implement collaborating objects. Such patterns can be used to separate components so that the components correspond to the differences between variants. For instances, the observer pattern is used to integrate extensions (i.e. classes that extend the base class) into the behavior of a base class. The extensions implement predefined behavior that, once they are added to the base class, is called by the base class on specific events. In this case, variants are composed by adding extensions to base classes. The advantage here is, that code can be cleanly separated: Components do not create residues in classes that they extend.

Frameworks

There are white and blackbox frameworks. Both have in common that they provide abstract solution for a set of related problems. Extensions of the framework implement concrete implementations of such abstract solutions and variability is achieved by the choice of extensions. The difference between whitebox and blackbox frameworks is that for whitebox frameworks, the developer must know the implementation details of the framework's abstract classes. With blackbox frameworks, only the concrete extension points need to be known; hence, the extensions can be compiled independently of the framework.

In this thesis, we rely on the capabilities offered by the language workbench (see chapter 4) in which MLE is programmed. The workflow in MPS is tailored to the creation of concepts, which is achieved

by implementing their different aspects. Thereby, it inhibits the development of design patterns. Due to the intended extensibility of the DSL and the difficulty to comprehend parameter-based variability mechanisms, we follow the approach of frameworks. MLE is designed to be a blackbox framework which, however, provides insights into the implementation details in order to enable a deep integration into the system (which might only be required in individual cases).

3.2 VISUALIZING PRODUCT LINES

A product lines consists of a set of components which are in relation to each other. The relations can be visualized by a components model[2] (i.e. a tree diagram) which has been introduced first by C. Kang et al. (1990).

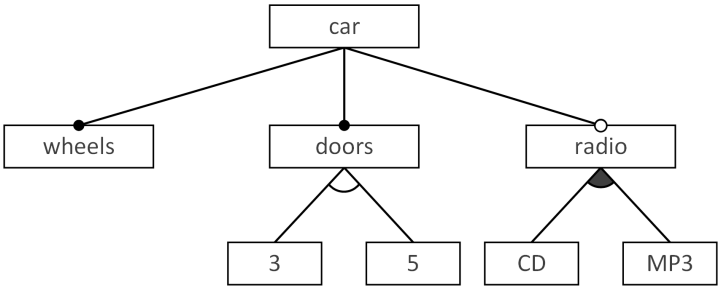

Figure 3.1: Component model – example for a car

An example of a components model for a car is shown in figure 3.1. The root node describes the type of product that can be described by the model. Child nodes correspond to components which can be mandatory (filled circle) or optional (unfilled circle): the car requires wheels and doors, but a radio is optional. Some children are further specified: The number of doors is to be selected from an alternative group (unfilled arc) that enables the selection of exactly one child. A car has either 3 or 5 doors but not both. The radio, in contrast, can have multiple playback options. The options can be selected from an or group (filled arc). The "or" is meant as logical or meaning that at least one option is required.

3.3 PRODUCT LINES IN MPS

There are two MLE frameworks to our knowledge that enable PLE inside MPS:

2 Again, the term "feature" is avoided here. In literature the component model is referred to as feature model.

First, Voelter and Visser (2011) evaluate the application of DSLs in product line engineering. In their paper, they show that DSLs are suited well as middle course between component models and the programming. The separation of components inside the repository is often problematic due to the interactions that are required between them. Here, DSLs for PLE can provide another layer of abstraction that improves efficiency in the process. Even though no framework is delivered by Voelter and Visser, the findings were obviously integrated into the mbeddr framework (see section 4.5).

Second, Behringer, Palz, and Berger (2017) present Projectional Editing of Product Lines (PEoPL), a DSL that enables multiple projections for variability mechanisms. Annotative representations such as ifdefs[3] are supported as well as modular representations (similar to component models). Developers can switch between those representations in favor for efficient realization (annotative) or easier comprehension (modular).

Regarding this thesis both frameworks, mbeddr and PEoPL, share the same disadvantage: They are designed to compile GPL solutions from the DSL code. However, this thesis aims at providing an adaptable range of function inside the DSL. An appropriate solution for this use case is presented in chapter 5.

3 i.e. labels that mark areas which are considered only if the respective component is selected

4

LANGUAGE ENGINEERING WITH JETBRAINS MPS

Jetbrains' MPS is a language workbench that supports the implementation of external DSLs. Also, language modules can be used to adapt the MPS development environment to the needs of the DSL. In this chapter we explain projectional editing (see section 4.1), since it is one of MPS' main features, and we heavily rely on these concepts in this work. Then, we introduce the components of which a programming language consists in general (see section 4.2), before we illustrate the implementation of language concepts on a concrete example (see section 4.4). We conclude this section with a brief overview on the mbeddr platform and related frameworks (see section 4.5) that facilitate the implementation of language modules in MPS.

4.1 PROJECTIONAL EDITING

Languages written with MPS are not represented textually but projectionally. This means that a user of such language is not writing plain text to accomplish a task but is choosing from a list of concepts that are valid in their specific contexts. Therefore, the source code is not textual but consists of graphical elements which can be simple text but also more complex such as graphs, tables, or formulas. Although the code might look like text, it really is a list of references to concepts.

Figure 4.1: Typing in the edit-statement: unrecognized text is red (top). When completed writing the concept-alias, the concept is auto-completed (bottom).

This type of programming might appear limiting at first: code cannot simply be copied from elsewhere into the editor. Writing own code can sometimes be cumbersome to the restrictions, and the DSL highly depends on MPS as development environment (which means that users must cope with MPS and cannot use their usual IDE). But those restrictions are accompanied by great advantages:

First, projectional editing makes it impossible to write syntactically incorrect code. Although code is typed in, it is only transformed into

the respective concepts if the concept-aliases are recognized (compare figure 4.1). The concrete syntax is auto-completed and only specific cells of the concept's representation can be edited. There will be no more annoying errors caused by skipping a parenthesis or semicolon.

Second, users can see only options that are valid in the specific context. Since the context is always known, code-suggestions can be used to explore the possibilities that developers have at the current point in the program. Therefore, users are encouraged not only to use statements and expressions they are familiar with, but to explore the greater scope of functions provided by the respective language.

4.2 ELEMENTS OF PROGRAMMING LANGUAGES

Instead of sentences, statements are used to write source code. They instantiate concepts such as variable declarations, assignments, conditionals, or loops. Unlike texts, the structure of programming-code is not linear but can be considered as a tree (which is due to conditionals, for instance).

A programming language, such as any other language, is defined by syntax and semantics. The statements themselves as well as the ordering of statements need to follow the rules of the language's syntax. Each statement has a semantic and the arrangement of statement defines the semantic (i.e. the purpose) of a program.

With respect to language engineering, there is a distinction between *abstract syntax* and *concrete syntax*: The abstract syntax describes the structure of the statement (e.g. an if-statement consists of a condition, a list of statements that are called when the condition is met, and another list of statements that are called otherwise). The concrete syntax, on the other hand, describes its concrete shape (e.g. the statement is introduced by the keyword "if", the condition is wrapped in parentheses, etc.). The if statement has different concrete syntaxes in different languages such as Java or Python (compare figure 4.2); however, the abstract syntax is the same. There is the same concept behind if-statements over different languages.

There are also two kinds of semantics: static and execution semantics. The first one defines constraints or type-system rules for statements in a specific context. A syntactically correct if-statement can be meaningless if the *static semantics* are not met. For instance, the condition inside the if statement could compare two variables of different type. Also, there could be an else-block even though the condition is always true (e.g. by comparing a variable with itself). There is no reason that a developer intended such kind of condition; thus, it should be tagged as erroneous by the static semantics. The latter one – the *execution semantics* – is independent from the concrete syntax. Only the

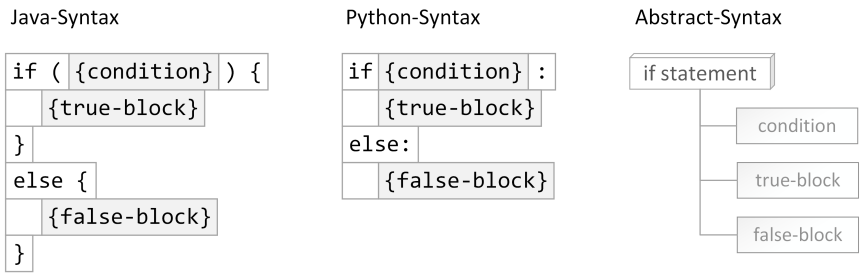

Figure 4.2: The different concrete syntaxes of an if-statement for Java (left) and Python (middle) both reflect the abstract syntax shown on the right.

abstract syntax matters here, which can be considered a tree as well: the Abstract Syntax Tree (AST). Each instance of a concept is a node in this tree. An execution engine is traversing the AST to run the program. While doing so on an external DSL, there are two approaches: First, GPL code is generated from the DSL and forwarded to the existing execution engines. Second, DSL code is directly interpreted by an execution engine that is developed alongside the DSL.

4.3 THE MPS PROJECT STRUCTURE

An MPS project consists of several nested entities that provide access to different aspects of the DSL. The entities can be considered in two different ways: structural and semantical. Both views are orthogonal to each other.

Looking at the semantics, the upper most entity is the project itself. It consists of languages, solutions, and devKits:

- *Languages* contain concepts of which programs are composed.
- *Solutions* contain files in which actual programs are written. Here the language concepts are applied.
- *DevKits* are groups of languages that can be used to reference multiple entities at once.

Looking at the structure, projects are called repositories. A repository then consists of modules (which semantically can be languages, solutions, or devKits). Then there are two entities which are both called 'model' in MPS. One of them is a model container (inside languages this is referred to as aspects) which holds a configuration (e.g. used languages and dependencies) for all child-models. The child-models are the actual files that can be edited inside MPS.

Figure 4.3 shows the semantical project view on the left side as well as the corresponding structural terms to the right. MPS indicates languages by yellow square-icon with the letter "L" inside (see language "core") and solutions by purple ones with the letter "S". There are nine aspects in the core language (listeners, structure, editor, etc.). To define a language concept models can be created for each aspect (see section 4.4 below). Besides the structural entities already described above "virtual folders" are used to visually group models. They do not affect the modules or models in any way and are only used for the sake of clarity.

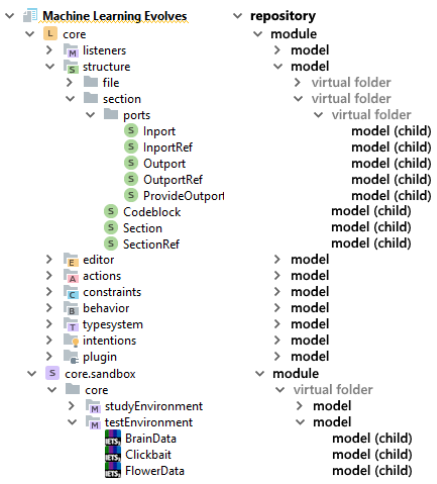

Figure 4.3: Semantical and structural view on an MPS project

DSLs created in MPS are usually composed of many language-modules. Some languages then depend on other ones (i.e. concepts of the other language are used) and optionally extend them (e.g. by implementing concept-interfaces). Thus, the DSL is modularized which enables easier maintenance and a better overview. The main work in MPS is in the creation of language concepts which is described in the next section.

4.4 DEFINING CONCEPTS IN MPS

The implementation of concepts in MPS is subdivided into aspects. Mikhail Barash wrote an illustrative introduction to the implementation of DSLs with Jetbrains MPS that addresses readers that are not familiar with language engineering in general. This section is based on this introduction but differs in three issues: First, it is more precise with respects to terminology in language engineering. Second, it is more in-depth so that the implementation details in other chapters of this thesis can better be followed. Third, it emphasizes on aspects that matter most for this work. This implementation of concepts is illustrated briefly by the example of the concepts **Variable Declaration** and **Variable Reference**. As the names suggest the concepts are used to declare (and optionally initialize) a variable and refer to it at another point in the program. The existence of some other concepts is presumed and not further explained since their utility is self-explanatory in the context.

4.4.1 *Structure Aspect: The Abstract Syntax*

The Structure Aspect models the abstract syntax of the concept. Concepts consist of elements which can be properties (displayed as gray fields in the diagrams), children (yellow fields), or references (brown fields). Properties are of primitive types, children are instances of other concepts, and references are used to refer to nodes (i.e. instances of concepts) that are already placed somewhere inside the AST. Those nodes should not be copied or dislocated to prevent inconsistency in the model.

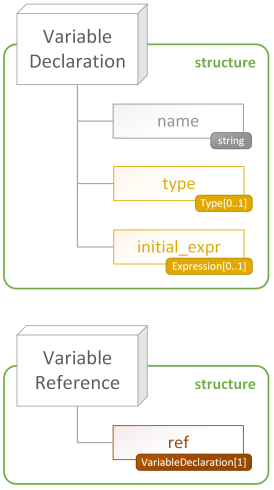

Figure 4.4: The structure-aspect. Properties are gray, children yellow, and references red fields.

Note: Primitive types in MPS are strings, integers, and enumeration values)

The structure of the concept **Variable Declaration** is shown in figure 4.4. A variable requires a *name* which is simply a string and thus a property. It also requires a *type* which is a child of concept **Type** and optional (it might be assigned only once – indicated by the numbers behind the concept name). The type is optional because it can be inferred from the other optional child *init_expr* which is of concept **Expression**. To reference a variable at any point in the AST, the other concept **Variable Reference** is defined which consists only of a reference *ref* pointing to a node of concept **Variable Declaration**.

4.4.2 *Editor Aspect: The Concrete Syntax*

The Editor Aspect is meant to define the concrete syntax of a concept. It provides the representation of a concept that the user will finally use to write code; thus, designing the editor crucially affects the usability of a language. In particular, this applies with respect to DSLs written in MPS, since its projectional editing already feels unfamiliar for users who are used to textual editing.

In MPS, language designers can build various concrete representations for the same concept. For instance, figure 4.5 presents two different editors for the concept **Variable Declaration** – one with and one without the initial expression. The type is present in both editors, because it should be specified within the declaration. However, in the editor with *init_expr*, it is inferred from the expression and therefore a read-only field.

Editors are built of cells which themselves can be arranged in cell collections. The cells of the editors in figure 4.5 are wrapped in horizontal collections (`[>` and `<]`). Cells with white background and black font-color are text constants (in this example `var` , `:` , and `=`). They are not editable so that no syntax errors can be produced. The gray `name` cell is a placeholder for property *name* and the yellow cells `type` and `init_expr` for the children of **Variable Declaration** (compare section 4.4.1). On the presentation (which is generated from the editor aspect), the user needs to fill those yellow cells with a string respective child-concepts.

Note: The figures give no clue whether cells are editable or read-only. Same is true for editor files in MPS: An Inspector Window needs to be opened to see extra parameters

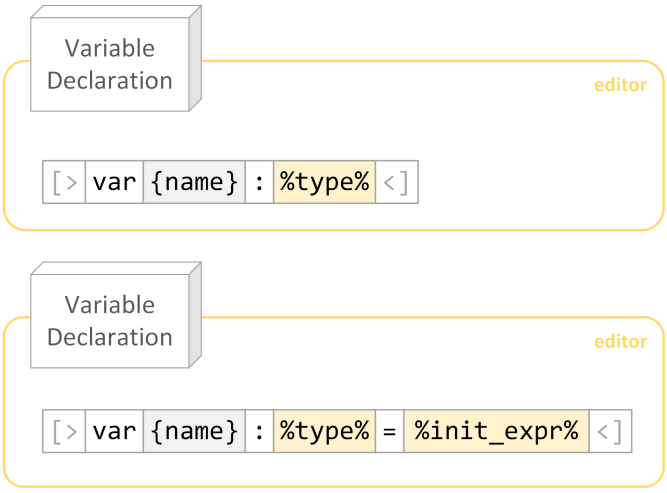

Figure 4.5: The editor-aspect. Cells are arranged in collections. Background-colors refer to the elements of the structure-aspect. Text-constants are white.

A concept is instantiated when a user simply types the concept name (or an alias) inside the editor. For the **Variable Declaration** the alias could be the term "var". When the user types in `var` the editor shows the concept:

```
var <name> : <type>
```

By default, the cursor is set to the first editable field. Thus, when the editor aspect is carefully implemented, the transformation from text to the editor projection feels like auto completion in textual editors.

Assuming the user wants to initialize the variable in another line, the editor for the **Variable Reference** is required. It consists of only one cell, the reference `ref` . This cell would be replaced by the standard editor of the referenced node. However, it is pointed to the property *name* (depicted by `->` `{name}`) so that only the property cell will be shown in the editor. The property cannot be edited here but, instead, the name that

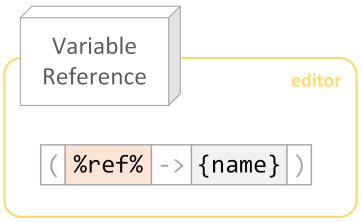

Figure 4.6: The editor-aspect of the variable reference.

is typed inside this concept is matched with the name of any **Variable Declaration**. Only when a match is found, the reference will be instantiated. Simply put, whenever **Variable References** are applicable, the user might enter the name of any **Variable Declaration** and a reference will be created automatically.

4.4.3 *Constraints and Type-System Aspects: The Static Semantics*

The constraints and type-system aspects model the static semantics of a DSL. In the constraints, aspect rules are defined that determine whether elements semantically can be attributed to a certain node depending on the node's context in the AST. A specific class of such rules is based on type expectations and separately defined in the type-system aspect.

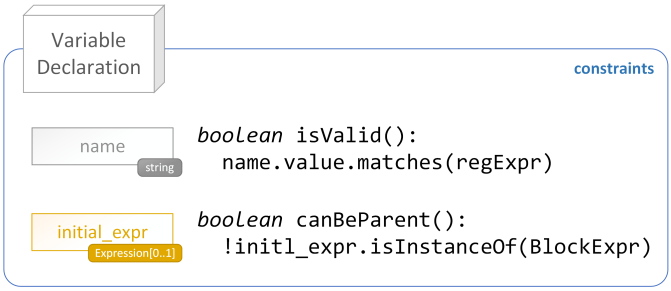

Figure 4.7: The constraints-aspect for the variable declaration. The name needs to match the regular expression and initial_expr must not be a block-expression.

Figure 4.7 shows the constraints for the *name* and the *init_expr* of the **Variable Declaration**. A string is excepted only as name when it matches the regular expression "regExpr" which could demand that only alpha-numerical characters are used with the first being alphabetical. The *init_expr* is defined as an **Expression** which also can be a node of any sub-concept such as the **BlockExpr**. The declaration of a variable should be easily understandable; therefore, block expressions – although being expressions – are forbidden by constraint.

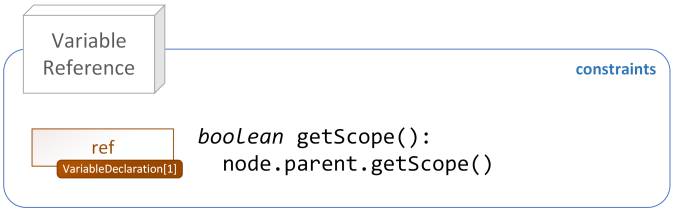

Figure 4.8: The constraints-aspect for the variable reference. The reference can be chosen from all variable-declarations inside the computed scope.

Furthermore, the constrained aspect is used to define scopes for references. By default, it can be referred to any concept that corresponds to the reference concept, but rules can constraint the scope to

a subset of the AST. The **Variable Reference** itself might be a child of the AST's root or of a block expression. It then must refer only to **Variable Declaration**s that are within the scope of its parent (see the code in figure 4.8). The method `.getScope()` might include all **Variable Declaration**s but those that are descendants and not direct children of the respective node.

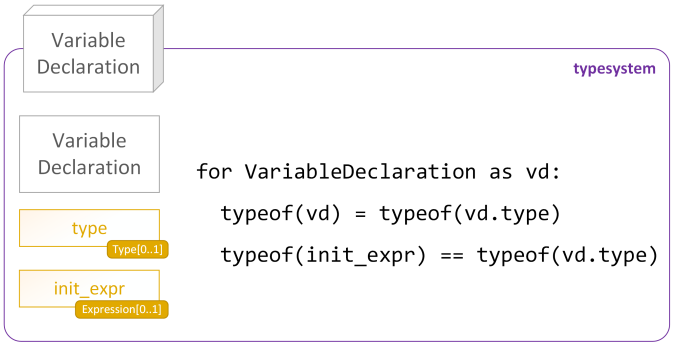

Figure 4.9: The typesystem-aspect for the variable declaration. The type of the declaration is defined as the specified child "type". Only initital expressions of the same type are applicable.

Note: Somehow the type of an object can only be inferred if the object is inside the AST which is not the case if it exists only during code interpretation

The type-system aspect is used to define the type of a concept's node as well as to demand a specific type for child concepts. The type of a concept can be referenced by the statement `typeof(concept)`. This statement looks like a function but behaves like a variable: the desired type (which in the end will be a type-concept) is assigned to the `typeof()` statement. The type of the **Variable Declaration** is specified by its child *type*. Of course, also the type of the *init_expr* should match that type. Those two clauses are reflected by the rules in figure 4.9: the first is depicted as assignment (single =) and the latter as comparison (double ==). The type of the **Variable Reference** should be the same as the type of the declaration which corresponds to the type-assignment in figure 4.10.

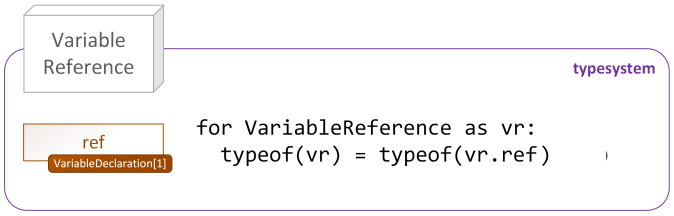

Figure 4.10: The typesystem-aspect for the variable reference. The type of the reference is defined as the type of the referenced node.

The constraints and type-system rules are considered when the user writes code with the DSL so that only applicable concepts are suggested for autocompletion which again reduces the complexity and improves the efficiency of the language.

4.4.4 *Actions and Intentions Aspects: Usability Improvements*

The actions and intention aspects are used to improve the usability of a DSL. Actions are triggered by the user while writing code inside the editor. Intentions are methods intentionally called. An example for an action is the transformation from one editor representation to another one. The **Variable Declaration** has two different editors – with and without *init_expr*. By default, the latter is shown in the editor when the concept name or alias is typed. Similarly to concepts, actions can be called by typing their "matching text". Figure 4.11 depicts an action "AddInitExpr" where the matching text is `=` . Writing this text next to the default editor will call the action that transforms it into the other editor.

Note: the same effect can be achieved with less effort by using mbeddr's so-called grammacells (see section 4.5)

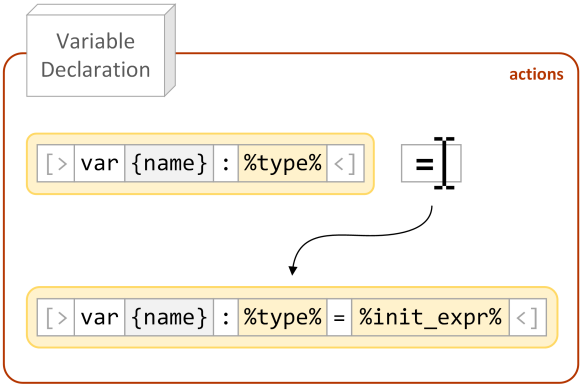

Figure 4.11: The actions-aspect for variable declarations. Typing "=" at the right side of the statement transforms the projection (the initial expression is included).

Intentions can provide time-saving shortcuts inside the MPS editor. To apply an intention-action, the corresponding node in the AST needs to be in focus. The user can then press "Alt" + "Enter" and a list of available intention actions is inserted from which one can be activated. Typical intentions are quick fixes such as the adaption of a **Variable Declaration**'s type according to the *init_expr* or just shortcuts such as the removal of the *init_expr* which might be complex and therefore time consuming to delete (compare figure 4.12).

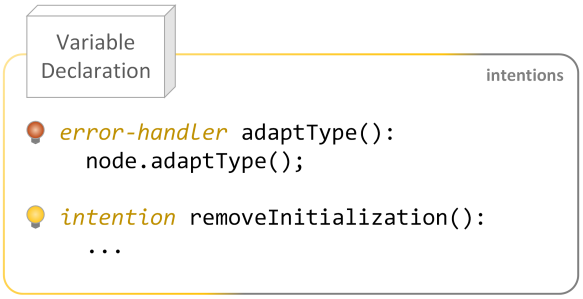

Figure 4.12: The intentions-aspect of the variable declaration. The type could be adapted if it does not fit the initialization or an initialization removed, if present.

Intentions are functions that can be called depending on the context. Error-handlers are called when an error is detected. Their availability is signaled by a red bulb in the editor. For other intentions, context conditions can be defined (such as the presence of the *init_expr*). Often, there are actions and intentions that provide the same effect in different ways. The user than can choose the preferred approach which is considered a usability improvement as well.

4.4.5 *Behavior and Custom Aspects: Utility*

Note: Although static methods can be defined there are no static members of concepts

The behavior aspect is used to encapsulate functionality that is related to the concept in a manner of object-oriented programming. Methods that are defined here can be called on an instantiated node of the concept. This allows developers to implement them once and call them in any other aspect of the respective concept or when traversing the AST.

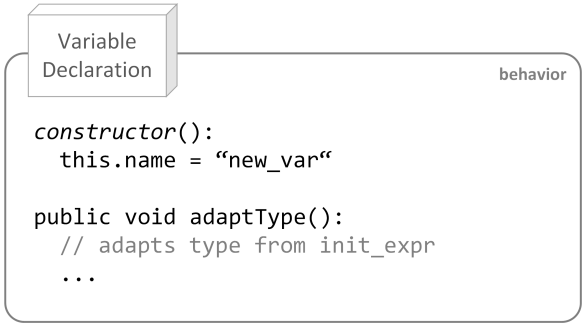

Figure 4.13: The behavior-aspect of the variable declaration. Methods are defined that can be called upon the concept-node (e.g. from inside the intentions-aspect).

Note: It is apparent that the constructor example is artificial but similar presets often come in handy

For instance, the method "adaptType()" (see figure 4.13) is defined for the **Variable Declaration** inside the behavior aspect and called inside the intentions aspect (compare figure 4.12). Also, a constructor is defined which presets the name of the node so that the user can focus on defining the *type* or writing the *init_expr*.

MPS enables not only the development of DSLs but also the enhancement of the language-workbench itself. For instance, language-modules can be implemented that provide further custom aspects. While the behavior encapsulates functionality that might be used in different contexts, custom aspects are used to provide and encapsulate specific behavior that is not reflected in other predefined aspects. A good example is the listeners aspect that enables the designer of a DSL to define events on nodes (see section 4.5).

4.4.6 *Generator and TextGen Aspects: The Execution Engines*

The generator and the text-generation aspects provide the default execution engine in MPS. They are used to transform the language-model into the model of another DSL or into text (which then usually is any GPL code). In this work, interpretation is used as execution engine; therefore, the generator and textgen aspect hardly matter for this DSL. A framework for language interpretation is provided by the mbeddr platform (see section 4.5). Unfortunately (for the consistency of this introduction), the interpreter is not designed as a custom aspect but as a language-module that is integrated into a plugin-aspect which was used in older versions of MPS to define custom aspects.

Interactions between Aspects

The aspects in MPS can be organized into three pillars of a DSL: the AST, the User Interface (UI), and the Execution Engine.

Note: An interpreter aspect is by default not provided with MPS but with the mbeddr platform. This is indicated by the dashed line in the diagram

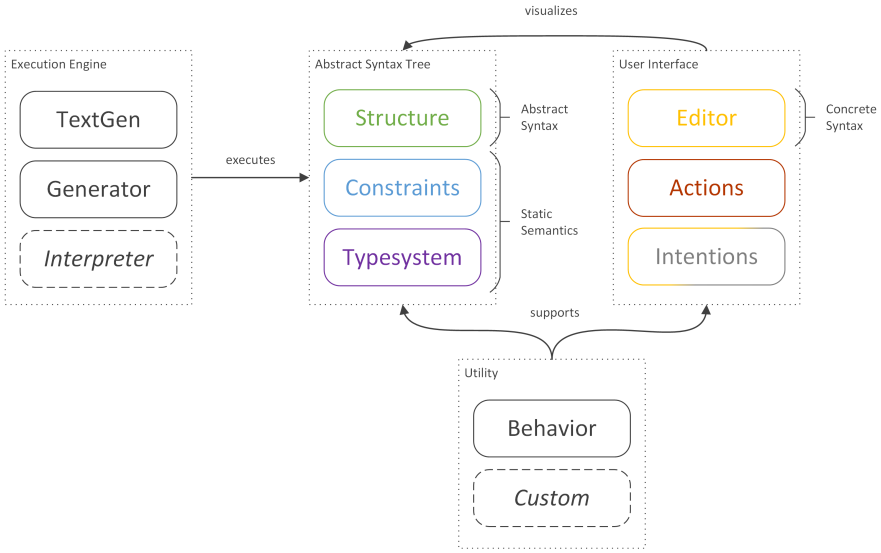

Figure 4.14: The interaction of the individual aspects in concepts

The AST contains the semantics of the program. Its shape (i.e. the possibilities how the AST can be constructed by the user) is defined by the structure aspect which provides the abstract syntax of the language and is constrained by the static semantics (modeled with the constraints and type-system aspects). The UI is responsible for the usability of the language. It consists of the editor aspect which provides the concrete syntax as well as actions and intentions aspects. The Execution Engine is meant to run the program modeled in the AST. This can be done by translation (code generation), compilation (which could be done with DSL code translated into GPL code), or by interpretation.

How these pillars relate to each other is displayed in figure 4.14: The AST is visualized by the UI which also allows modification of the AST. The Execution Engine on the other side executes the semantic that is modeled via the AST. Moreover, there is a Utility block which consists of the behavior aspect as well as potential custom aspects. It supports the development of the AST and the UI.

4.5 MPS FRAMEWORKS

MPS enables developers not only to implement and reuse programming languages, but also to configure and enhance the editor itself. Solutions that are developed in MPS can be published as plugins that provide new features such as tooling-windows, custom aspect-types for language-modules, or new editor-cells. The thesis is based on two such frameworks, which are closely connected: mbeddr and kernelF.

mbeddr

mbeddr has been the idea and topic of Völter's doctor thesis "Generic Tools, Specific Languages" (2014). It provides a projectional implementation of C as well as language-extensions such as state machines or variability mechanisms. Relevant for this thesis are tools and helper-languages that facilitate the development of new DSLs. Such tools are grouped under the label "mbeddr platform". The most important mbeddr-features for this work are listed below:

INTERPRETER-FRAMEWORK The interpreter-framework enables developers to create DSLs that are not compiled but interpreted (i.e. the code can be executed without first being translated into another language). For each language-module an interpreter can be implemented that defines how language-concepts are interpreted. The framework computes the state of concepts during interpretation inside a separate environment in order to prevent effects on the written DSL code.

Note: the actions-aspect could be used for similar effects but mbeddr's gramma-cells are more convenient and consistent

EDITOR-CELLS mbeddr provides a collection of editor-cells that can be used to create more responsive or more expressive projections. For more responsive projections Grammar-Cells such as *optional* or *wrapper*-cells are introduced. Optional-cells are invisible until a specified keyword (which triggers the option) is entered, whereas, wrapper-cells allow other properties than the name of a concept to instantiate the projection. Examples for more expressive editor-cells are tree-cells or frame-cells which are used in chapter 5 to visualize language-component trees.

LISTENERS-ASPECT The listeners-aspect can be used to listen to changes on concept-nodes. The addition or removal of a properties, child-nodes, or references can trigger a concept-listener to execute any previously defined code. Such automated, event-based actions can easily improve the usability of the DSL – for example by setting properties of a node depending on a selected child.

kernelF

KernelF (Voelter, 2018) is a functional language designed to be highly extensible and hence to be used as core-component of DSLs (Voelter, 2018). Among others, it provides basic types, literals, and operators that can be used as basis for extensions. In this work, we use kernelF as basis for feature-expressions (compare section 6.3.1). Voelter is also the main architect behind mbeddr. As a result, kernelF depends on the mbeddr-framework briefly discussed above (e.g. by using mbeddr's interpreter-framework)[1].

1 At least, this is true for the version of kernelF used in this thesis. However, kernelF aimed to be an independent core-language and might have achieved that goal by now.

Part II

CONTRIBUTION

5

VARIABLE LANGUAGE COMPOSITION

The modularized structure of MLE allows the composition of variable language-dialects. The reduced scope of dialects can help the user to deal only with language concepts that are needed in the specific context; therefore, it improves the efficiency in learning and using the DSL. In particular, this applies to projectional editing where code is not written but selected from available concepts. The number of selectable concepts is reduced by excluding language components. Also, code-completion is faster when there are less conflicting concept-names.

Note: concept names can be conflicting when they are equal (which is possible in MPS) or begin with the same letters.

As stated earlier in chapter 3, there are already frameworks developed in MPS that provide variability mechanisms. The problem with such solutions is that the resulting products are generated. In this work, however, the DSL for which variability mechanisms are required, produces various dialects of itself. Due to the complexity of external DSLs in MPS (e.g. concepts are separated into aspects which internally are recomposed to Java classes), we estimated the effort of creating a new, interpreter-based mechanism for language composition lower than using existing tools.

For language composition, we present an UI that allows for the selection of valid set of components only. It needs to list and arrange all relevant components. This is not the same as listing all modules in the repository, as there could be many utility-modules included like the composition-module itself. Therefore, components that belong to the language need to be designated, and their relation to each other to be described. Then, the interface should as well visualize the dependencies between components and ultimately provide interaction on the visual elements.

In this section, we explain the implementation of MLE's variability mechanism in detail. First, we give an overview of the composition interface in section 5.1. We then describe how the designation of a language components is implemented in section 5.2, which includes the definition of dependencies between different components. Finally, we outline how the language-component tree is built and language-composition is implemented in section 5.3.

5.1 USAGE OF THE COMPOSITION LANGUAGE

The language composition in MLE is realized with an interactive language-component tree (see figure 5.1). As part of the configuration of an MLE file the component tree is placed right below the file-header. Each node corresponds to a language-component which in terms of MPS is a language-module. The programs written in MLE are models inside solution-modules. The rationale behind the tree is that the respective language-modules can be added as dependencies of the model by interacting with the nodes.

Note: The structure of an MPS project is introduced in section 4.3.

5.1.1 *Interactions with the Language Component Tree*

This section gives an overview of the use of the language-component tree and the related interactions.

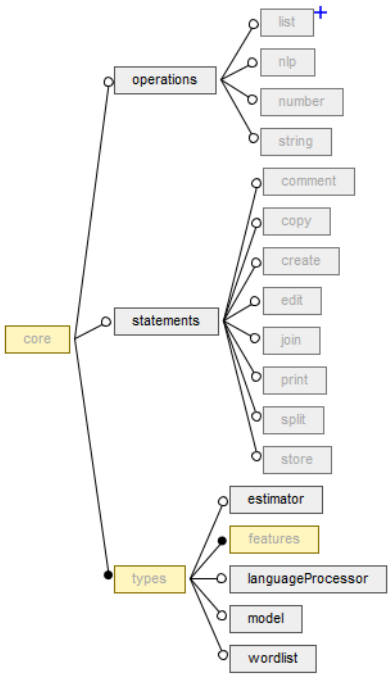

Figure 5.1: The language-components tree for MLE. The state of a node is represented by the background-color (selected) and by the font-color (editable).

The state of the nodes is visualized by colors. The background displays whether nodes are selected (yellow) or not (gray). The font-color of the label's signals if a node is editable: nodes that cannot be toggled are grayed out. This is the case if either the requirements for unselected nodes are not fulfilled[1] or selected nodes are required[2]. The restrictions regarding the selection ensure that only valid configurations can be created.

A Node can be toggled by a double-click or by pressing enter while the cursor is on it. There are also various intentions that can improve the efficiency during language composition. For instance, components that can't be toggled can be force-selected or deselected (depending on the current state). All child-nodes can be selected at once unless the number of selectable children is restricted. Also, the process of loading the components is initiated by an intention. The language-modules are then added as dependencies to the respective model inside the solution-module.

1 i.e. the parent or a required node are not selected, or an excluded node is selected
2 i.e. because it is prerequisite or other selected nodes depend on the selected one

5.1.2 *Registration of Language Components*

The registration of a language component is realized by creating a **ComponentDescription** inside the plugin-aspect of the language-module. The plugin-aspect must use the composition-language so that the description can be added as a root-concept of the model. The form in the newly created **ComponentDescription** (see figure 5.2) then needs to be filled.

The developer can choose a name for the component from which the path and the filename (i.e. the name of the model) are generated. Regarding the tree, the components do not specify their children, but attach themselves to a parent by selecting the respective reference. This allows developers to add components at any point of the tree (except the root) without modifying existing language-modules.

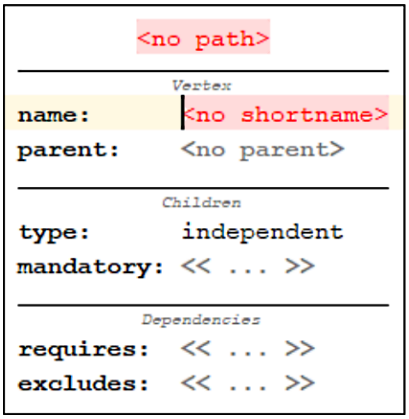

Figure 5.2: The presentation of a ComponentDescription which is used to register a component.

On the other hand, components cannot define whether they are mandatory or not. This is defined by the field "mandatory" in the parent-node to ensure that only those components are mandatory that are really needed. It is assumed that these are provided by the same developer who created the parent-node. Also, the relation between siblings is specified inside the parent node since all children of one node should stand in the same relation to each other.

In addition to mandatory children the **ComponentDescription** also specifies which other components the described one depends on and which are incompatible with it (see the "requires" and "excludes"-fields in figure 5.2). The selected components must neither be descendants nor ancestors of the component in questions since for those the inheritance already defines the dependence.

After the creation of the **ComponentDescription** the component is automatically added to the list of language components by a helper-class named "Components" (see section 5.2.5). The component can now be referenced in other description files and will be added in the language-component trees of new MLE files.

5.2 IMPLEMENTATION OF THE COMPONENT-DESCRIPTION

The **ComponentDescription** is used to register a module as a language component in MLE as stated earlier in section 5.1.2. The current section covers details on the implementation starting with the structure-aspect (see section 5.2.1) which corresponds to the abstract syntax of the concept. The constraints are discussed (see section 5.2.2) which lay out which child-elements can be added to the description and which objects can be referenced. Then the editor-aspect (see section 5.2.3) which defines the interface that a developer finally uses is explained before the listeners and the behavior-aspect (see section 5.2.4) which are used to react on input on the description-form. Finally, a helper-class (see section 5.2.5) that provides a collection of utility-functions for addressing components is outlined.

5.2.1 *Structure*

The **ComponentDescription** has two types of names which both are string-properties: the *name* is inherited from the implemented named-concept interface (see *«INamedConcept»* in figure 5.3) and is the name of the model which is used to display the file in the MPS project-view. Since there can be all types of models inside the plugin-aspect the name is put together from a prefix "ComponentDescription." and the name of the component which is stored in *shortname*. Moreover, *shortname* is used as label for the nodes in the component-tree.

Note: The choice of the name shortname seems a bit ill-fated to us and might not be final. We prefer the term alias which conflicts with an MPS property.

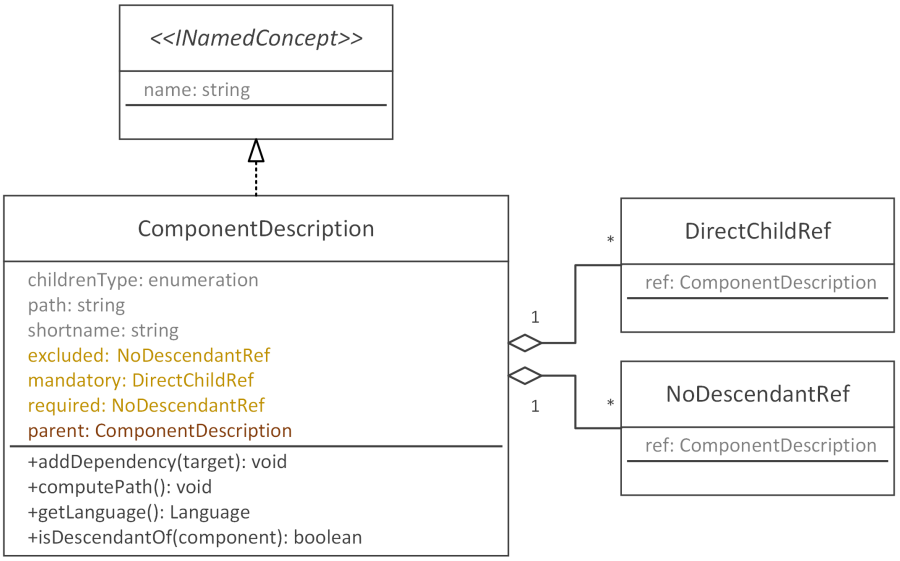

Figure 5.3: Class diagram of the ComponentDescription. The members are color-coded regarding the structure aspect of an MPS concept: gray for properties, orange for children, and red for references.

Another string-property is the *path* which reflects the position of the component in the component tree. It is only used to display it at the top of the **ComponentDescription**'s editor so that the developer of a language component can ensure that the parent-reference is set correctly. The *path* cannot be edited manually but is computed automatically every time when another component is selected as *parent* (see section 5.2.4 below).

The last property in **ComponentDescription** is *childrenType* which is an enumeration. Enumerations of the types boolean, integer, and string can be defined in models of concept **EnumDataType** inside the structure-aspect of a language-module. Here a string-enumeration with three entries is used:

```
1  enumeration datatype children_type
2
3    member type       : string
4    no default        : false
5    default           : <1-st member>
6    member identifier : derive from internal value
7
8    value ind   presentation independent     (default)
9    value or    presentation n-of-m
10   value xor   presentation 1-of-m
```

Listing 5.1: `composition.structure.children_type`
An enumeration datatype with member-type string. Entries are the strings "ind", "or", and "xor" listed in lines 8-10.

Enumeration-entries have a value and a presentation. The value can be read as in any other variable but not be edited. The presentation is what is shown in the editor (i.e. to the user of a language). For instance, a developer provides an abstract interface with the component being described. The child-components that implement that interface should be alternatives to each other; therefore, "1-of-m" is entered as children-type into the **ComponentDescription** form. The internal value that will be used to define the node-type of the component's tree-node is "xor".

The child-elements of **ComponentDescription** are reference-nodes like the **VariableReference** in section 4.4. Unlike the reference *parent* which can refer to any component inside the repository the scope of applicable components is restricted for the reference *ref* in **DirectChildRef** or **NoDescendantRef** (see section 5.2.2 below). The limited scope of the reference in those concepts also is the reason that child-concepts were used instead of references in **ComponentDescription**. By outsourcing the scope-computation, it is reused and hence not defined redundantly.

5.2.2 *Constraints*

There are two constraints defined for **ComponentDescription**: First, the concept must not be parent of any **DirectChildRef** in role "mandatory" if the *childrenType* defines an alternative-group[3]:

```
1 can be parent (node, childNode, childConcept, link,
2                operationContext)->boolean
3 {
4   boolean isOneOfMany = node.childrenType :eq: "xor";
5   boolean addsMandatory =
6     link :eq: link/ComponentDescription : mandatory/;
7   return !(isOneOfMany && addsMandatory);
8 }
```

Listing 5.2: `composition.constraints.ComponentDescription`
ComponentDescription cannot have children in the role of "mandatory" if it forms an alternative-group. The expression `link/.../` describes the role of a node in its parent.

Second, the scope for the reference *parent* is defined (see listing 5.3). The constraint here is negative since the scope is widened in this case: The default scope is defined by the module in which the **ComponentDescription** is created; however, the elements that will be returned by `Components.getAll()` are collected from all modules inside the repository. The only limitation for the parent reference is that it must not be the component currently edited.

```
1 link {parent}
2   referent set handler <none>
3   scope (referenceNode, contextNode, containmentLink,
4          position, linkTarget, operationContext)->Scope
5   {
6     node<ComponentDescription> self =
7       contextNode.ancestor<concept=ComponentDescription,+>
8     ListScope.forNamedElements(
9       Components.getAll().where({~it => it != self; }));
10  }
11  presentation (...)->string {
12    parameterNode.path;
13  }
```

Listing 5.3: `composition.constraints.ComponentDescription`
All ComponentDescriptions except the component in question can be selected as parent. The component path is used for presentation.

As already mentioned, the concepts **DirectChildRef** and **NoDescendantRef** also could have been realized as simple references in **ComponentDescription** but are outsourced to enable reuse of their scopes' computation. Listing 5.4 shows how the scope of **NoDescendantRef** is defined.

3 i.e. a group of components from which only one can be selected

```
1  link {ref}
2  referent set handler <none>
3  scope (referenceNode, contextNode, containmentLink,
4         position, linkTarget, operationContext)->Scope
5  {
6    node<ComponentDescription> component =
7      contextNode.ancestor<concept = ComponentDescription,+>
8    return ListScope.forNamedElements(
9      Components.getAll().where({ ~target =>
10         !target.isDescendantOf(component)
11         && !component.isDescendantOf(target)
12         && target != component
13         && target.parent :ne: component.parent;
14     }));
15 }
```

Listing 5.4: `composition.constraints.NoDescendantRef`
Computation of the scope for reference "ref" in concept NoDescendantRef. Only those targets that meet the condition can be selected.

The variable "component" in line 6 holds the **ComponentDescription** to which the reference is added either as required or as excluded component. The static function `.getAllComponents()` (called in line 9) is used to collect all components that are accessible in the repository. The result is reduced to those components that meet the condition in lines 10-13. The rationale behind each sub-condition is explained in the following:

- Line 10: The target must not be a descendant of the component because descendants cannot be excluded nor required (children can be mandatory though).
- Line 11: The component must not be a descendant of the target because it then already depends already on the target.
- Line 12: The target and the component must not be the same (which should be self-explanatory).
- Line 13: The target and the component must not be siblings because excludes or requirements should be defined differently then.

The last condition might appear improper as there is no offspring among siblings. The rational is that a component that depends on a sibling rather should be its child-component. Furthermore, siblings that exclude each other should be arranged inside an alternative-group [3].

Since **NoDescendantRef** consists of the reference alone, the reference-scope correspond to the code-suggestions for the concept. This means that only applicable concepts are suggested for the "requires" and "excludes" fields of the description (compare figure 5.2).

5.2.3 *Editor*

Note: Both types of cells, frame-cell and horizontal-line, are provided by the mbeddr framework

The editor for the **ComponentDescription** uses a frame-cell to visually enclose the description-form. Inside the frame horizontal-line-cells are used to separate different areas:

- Headline: the path from the tree-root to the component which is described here is displayed as text
- Vertex: the name of the component (which also is the label of the tree-node) and the component's parent are defined here
- Children: the relation between the component's children can be defined as well as mandatory children
- Dependencies: requires and excluded components can be selected (resulting in non-visible cross-tree edges)

The cells inside the frame-cell are arranged in a vertical collection which is indicated by the surrounding cells `[/` and `/]`. The three areas with the editable fields (highlighted blue in figure 5.4) in turn are arranged in horizontal-collections (`[-` ... `-]`) so that the fields are displayed next to their labels.

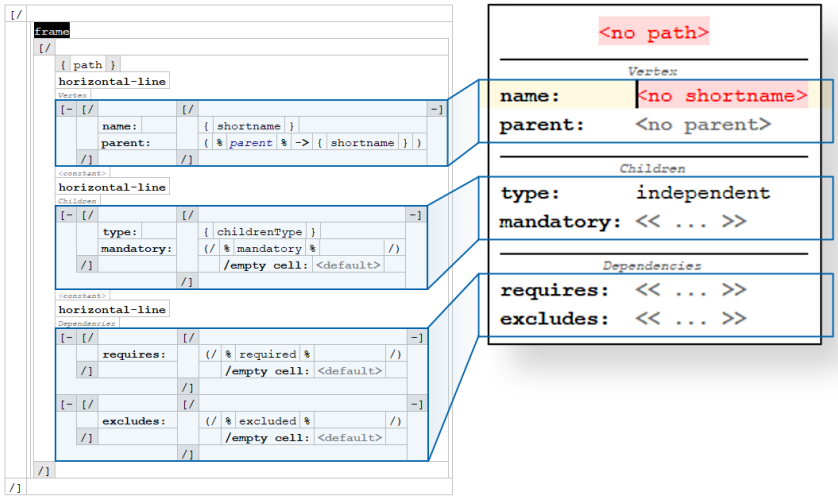

Figure 5.4: The editor-aspect (left) of the ComponentDescription next to its presentation (right). Corresponding areas are highlighted by blue frames.

The fields "mandatory", "requires", and "excludes" can each have more than one entry. Those entries are listed vertically so that for the dependencies-area two horizontal-collections are required. They make sure that the label "excludes" moves together with its input field when more than one entry is selected in the required-field.

The space between labels and input-fields is set equidistant by with-spaces in text-constants. This is no ideal solution but necessary since there are no grid-cells that consider vertical *and* horizontal spacing.

5.2.4 *Listeners and Behavior*

Listeners are used to directly react on inputs in the **ComponentDescription**. They trigger the concept's behavior (i.e. they call methods on the concept) as soon as a respective event is recognized. Listing 5.5 presents the reaction on the change of *shortname*:

```
1 model listeners for ComponentDescription {
2   property changed of property shortname (
3       instance, propertyName, oldValue, newValue)->void
4   {
5     instance.name = "ComponentDescription."
6       + instance.shortname;
7     instance.computePath();
8   }
9 ...
```

Listing 5.5: `composition.listeners.ComponentDescription`
A listener on the event "property changed" that listens to the change of ComponentDescription's property "shortname"

The listener manipulates the instance which triggered the event by changing the property *name* (line 5) and calling the behavior-method `.computePath()` (line 7). The *name* is simply used to display the respective **ConceptDescription** in the project view of MPS. Since there might be all kinds of models inside the plugins-aspect where the description is placed the prefix "ComponentDescription." is used to clarify the concept-type. The method `.computePath()` is presented in listing 5.6.

```
1 public void computePath() {
2   // compute path for this instance
3   string path = this.shortname;
4   node<ComponentDescription> curNode = this;
5   while (curNode.parent != null) {
6     curNode = curNode.parent;
7     path = curNode.shortname + "." + path;
8   }
9   this.path = path;
10
11  // compute path for all children
12  foreach node in Components.getChildren(this) {
13    node.computePath();
14  }
15 }
```

Listing 5.6: `composition.behavior.ComponentDescription`
The computePath-method computes the new path for the concept and all of its descendants.

The *path* of the current component in the components-tree is built in lines 2 to 8: First, the *shortname* is set as path-string and the component as `curNode`. Then the *shortname* of `curNode`'s parent is put in

front of the *path* and `curNode` is updated to be its parent while the parent is not null. Furthermore, also the path of each of the component's children is updated. To do so, the children are collected by the helper-class "Components" (see section 5.2.5) and the method `.computePath()` called recursively.

As can be seen in the UML-diagram in figure 5.3 there are three more behavior-methods in **ConceptDescriptions**:

- `.addDependency()`: Adds language-modules of referenced components to the dependencies of the description (see listing 5.7). Otherwise MPS would report missing-dependencies errors.
- `.getLanguage()`: Returns the language-module of the component so that it can be referenced in solution-models which have the component selected in the language-component tree.
- `.isDescendantOf()`: Returns whether the component is a descendant of another one. This is used to define reference-scopes as done for the *ref* in **NoDescendantRef** (compare listing 5.4)

The programmatic manipulation of dependencies is realized via the OpenAPI[4] which provides access to internal structures of MPS like project-properties. The method `.addDependency()` (see listing 5.7) requests the model (i.e. the **ComponentDescription**-file) via a concept-property call (line 4). It then requests the module (line 5) to which a reference of the target-module is added (line 7).

```
1 public void
2   addDependency(node<ComponentDescription> target)
3 {
4   SModel model = this.model;
5   AbstractModule smodule =
6     (AbstractModule) model.getModule();
7   smodule.addDependency(
8     target.getLanguage().getModuleReference(), false);
9 }
```

Listing 5.7: `composition.behavior.ComponentDescription`
The method addDependency uses MPS' OpenAPI to manipulate dependencies inside language-modules.

It should be noted that the method `.addDependency()` on the `AbstractModule` (line 7 in listing 5.7) is different than the one on component. Also, the method `.getLanguage()` does not belong to the OpenAPI but to implemented on the component.

4 Unfortunately there is no clear documentation for the OpenAPI. Nevertheless, the corresponding page in the online Wiki can be a starting point for further information: "Open API - accessing models from code" (2016)

5.2.5 *Components-Class*

The class `Components` provides static helper-functions that are used to collect **ComponentDescriptions**. It is required for two reasons: first, since components are added by creating a respective description inside the MLE project, there is no central store for language components that can be addressed directly. Second, components attach themselves to parent-nodes which means that parent-nodes do not know their child-nodes or even more distant descendants. The `Components`-class is used to collect all available components in the project or descendants of specific components.

Collect all components – getAll()

```
1  public static nlist<ComponentDescription> getAll() {
2    // look for repository
3    SModule srcModule = language/core/.getSourceModule();
4    SRepository repo = srcModule.getRepository();
5
6    // for every module in repository: check for config
7    nlist<ComponentDescription> components =
8      new nlist<ComponentDescription>;
9    foreach curModule in repo.getModules() {
10     if (curModule instanceof Language) {
11       SModel pluginAspect = findPluginAspect(curModule);
12       node<ComponentDescription> component =
13         findComponent(pluginAspect);
14       if (component :ne: null) {
15         components.add(component);
16       }
17     }
18   }
19   return components;
20 }
```

Listing 5.8: `composition.behavior.Components`
The method getAll browses through all modules in the repository and searches for ComponentDescriptions inside the plugin-aspect model.

To collect all components inside the project, the repository is required. It is selected from the core-component-module (i.e. the language "core" referenced by the expression `language/core/`) in lines 3 and 4 of listing 5.8. Then it is looped over all modules inside the repository (line 9). Only in modules of type language (line 10) is searched for **ComponentDescription**s. Descriptions are stored inside the plugin-aspect which can be found based on the model-name in `.findPluginAspect()` (line 11). If a **ComponentDescription** is found (which is checked with the not-equals operation `:ne:` in line 14) it is added to the list of components. The type `nlist<C>` is a list of nodes of the specified concept `C` (compare line 7).

Note: there are also modules of type solution and devKit as stated earlier in section 4.3

Collect descendants of a specific component

The two functions `getChildren(node)` and `getDescendants()` utilize the collection of all components explained above. The first one simply filters the result of `.getAll()` to get the nodes that have the target-node as parent (see listing 5.9) which is done in a single line of code:

```
1 public static nlist<ComponentDescription>
2 getChildren(node<ComponentDescription> node)
3 {
4 return getAll().where({~it => it.parent :eq: node; });
5 }
```

Listing 5.9: `composition.behavior.Components`
The method getChildren returns all components that attached themselves onto the target-node "node".

In a similar way, `getDescendants()` collects all child-nodes of the target-node by using the function `getChildren()` and repeats this recursively (see line 6 in listing 5.10) for the child-nodes found until the leaves of the tree (i.e. the nodes that have no further children) are found.

```
1 public static nlist<ComponentDescription>
2   getDescendants(node<ComponentDescription> node)
3 {
4   nlist<ComponentDescription> children = getChildren(node)
5   foreach component in children {
6     children.addAll(getDescendants(component));
7   }
8   return components;
9 }
```

Listing 5.10: `composition.behavior.Components`
The method getDescendants collects all descendents of the target-node by recursively collecting child-nodes.

5.3 IMPLEMENTATION OF THE COMPONENTS-TREE INTERFACE

The components-tree is used to visualize and edit possible language-compositions. Thus, it depends on the component-descriptions outlined earlier and transforms them into corresponding nodes of a component-model (see section 3.2).

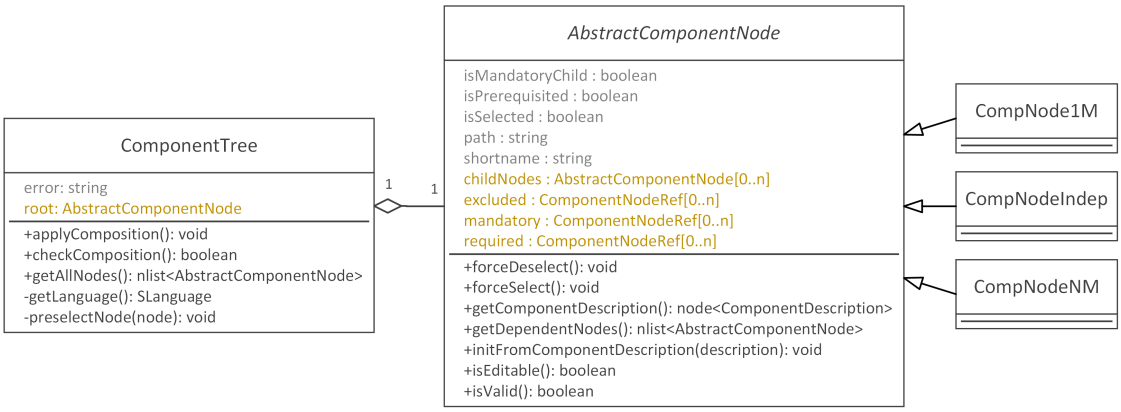

Figure 5.5: Class diagrams of the ComponentTree and the AbstractComponentNode: The tree is used to apply selected composition. Nodes contain the logic that restricts selections to valid compositions.

Two concepts are used to implement the components-tree: the concept **ComponentTree** which contains the root-node of the tree and finally applies the composition to the project, and the abstract concept *AbstractComponentNode* from which the component-nodes inherit their structure and behavior. As the class-diagram in figure 5.5 suggests, the nodes contain logic themselves which is used to prevent invalid language compositions.

In this section, we first outline the structure of both, the **ComponentTree** and the *AbstractComponentNode* in section 5.3.1. Then, we explain how the component model is built from **ComponentDescriptions** (section 5.3.1), and how the tree-editor is arranged (section 5.3.3). Finally, we discuss the loading process of a selected language composition.

5.3.1 *Structure of the Component Tree and Nodes*

The **ComponentTree** consists only of the root-node *root* and a string-property *error*. The root apparently provides access to the whole tree-structure. However, *error* is used to store an error-message during the check of the tree, if the composition is not valid. This should not happen, since the interactions on the tree are restricted by the underlying logic of the nodes.

A large part of the component-node's structure is adopted from the **ComponentDescription** (compare figure 5.3): the properties *path* and *shortname* as well as the children *excluded*, *mandatory*, and *required*. The main difference is, that **ComponentDescriptions** attach themselves on parents but component-nodes have children. Thus, the reference *parent* is replaced with a child-group *childNodes*. Furthermore, additional boolean-properties are added which flag if a node is mandatory, prerequisite, or selected. More complex flags are implemented as behavior.

5.3.2 *Building thee Tree-Interface*

The **ComponentTree** is automatically added to an MLE file on create. The concept itself, in turn, initiates the tree structure by calling the static function `.build()` on a helper-class that is called TreeBuilder. The whole class is provided as appendix C to this thesis.

The build-function performs the following steps, in order to create a components-tree from component-descriptions:

1. *Collecting all descriptions (lines 4/5).* This is accomplished by the class `Components`. Details on this class were already presented in section 5.2.5.

2. *Creating nodes from the description (lines 6/7).* The utilized function is declared in line 75. It creates concrete component-nodes depending on the property *childrenType* of the **ComponentDescription** (compare the chained if-else-statements starting in line 92). For instance, a **CompNode1M** is create for a description defines the children to be an alternative-group. The parent-node needs to reflect the relation between children in order to display the respective outgoing decoration on the edges of the tree (see section 5.3.3)

3. *Finding the root among the yet unconnected node (lines 8-10).* For this purpose, it first looks for the root-node in the component-descriptions.[5] The description-root is then passed to the function `.findEquivalentVertex()` in line 9, which compares the description to the nodes according to their path (starting at line 110) and returns the respective node.

4. *Connecting the nodes (line 11).* The function responsible for connecting the nodes is `.connectVertices()` and starts in line 17. A linked-list is utilized to traverse the descriptions in a

5 The function `.findRootInDescriptions()` starting in line 119 is called, which searches for the description which refers to the model "core". This is a very specific way to identify the root and should be generalized (e.g. find description with no parent) to be applicable in other scenarios, as well.

breadth-first manner: First, the root-node is added to the list (line 24). For each node in the list, the node is removed from the list (lines 27/28), and the children of the corresponding description are requested (line 30). The nodes corresponding to the child-descriptions are then collected and added to the child-nodes of the respective parent (line 32-34) as well as to the list (line 35). This procedure is repeated until the list is empty meaning that all descendants of the root-description were handled.

5. *Restores relations between nodes (line 12).* The parent-child relations were already restored by connecting the nodes. However, the cross-tree relations, "requires" and "exclude", still need to be refreshed, since they point to descriptions instead of nodes. The function `.restoreCrossTreeRelations()` starting in line 39 achieves this by traversing over all descriptions and manipulating according nodes. The manipulation is outsourced to the function declared in line 65 `.restoreReference()`, which replaces the old references.

Finally, the root-node is returned and placed as *root* into the **ComponentTree**. All other nodes are retrieved over the edges of the tree.

5.3.3 *Editor of the Component Nodes*

There are three concepts that inherit from *AbstractComponentNode*. Such concepts do not add to the structure or behavior of the inheriting class but provide different editors. Also, the editors look similar. However, they differ when looking into the inspector-window: Figure 5.6 shows the editor of all component-nodes on the top, and an area of the inspector view of **CompNodeNM** at the bottom.

The editor consists of a tree-cell provided by the mbeddr platform (see section 4.5) which is highlighted yellow. It has two editable sub-cells. First, the one in the top is filled with a frame-cell (which is provided by mbeddr as well) that is used to refine the node's visualization with a frame and a background-color. Second, the cell on the bottom contains the node's child-nodes, which have the same editor-structure.

When the tree node is in focus, the section "Tree" can be found in the inspector window. It allows some customization on the tree visualization and interaction. For the components-tree only the outgoing shape is relevant, since the other values are defined in other files. However, the outgoing shape is the property that distinguishes the various component-nodes.

Mbeddr provides two types of decorators: arcs and circles. The arc is used in MLE to visualize alternative or respectively or-groups of

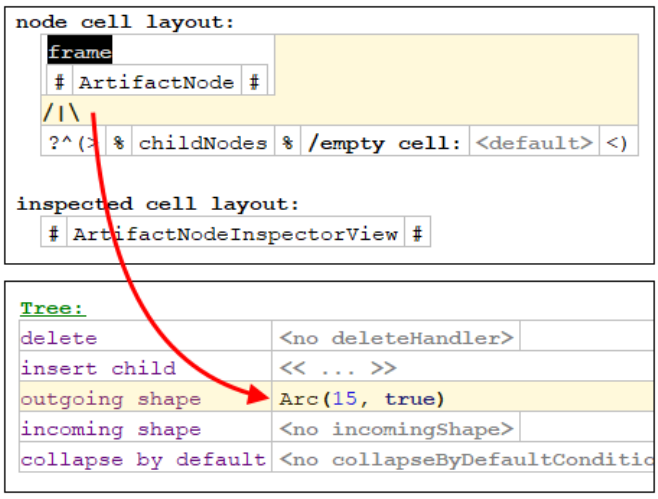

Figure 5.6: The editor view of a component-node (top) and the respective inspector view of the highlighted tree-cell (bottom).

component models (see section 3.2). The parameters are used to specify the radius of the arc, and whether the shape should be filled (or-group) or not (alternative-group).

Other visualization-options are defined in a stylesheet "treeStyle" which is stored inside the editor-aspect of the composition class. Among other, the use of incoming-shapes is specified for the tree-edges. This is achieved by using conditional styles: filled circles decorate the end of edges that at mandatory concepts and circles at optional ones.

5.3.4 *Loading a Component Composition*

To load a selected composition, the **ComponentTree** provides an intention "Load Language Components" (see listing 5.11) that is applicable on the whole tree. When the intention is executed, it calls the method `.checkComposition()` on the tree which returns a boolean value corresponding to the validity of the configuration:

```
1 execute(node, editorContext)->void {
2   if (node.checkComposition()) {
3     node.applyComposition();
4     // prompt success message ...
5   } else {
6     // prompt error message ...
7   }
8 }
```

Listing 5.11: `composition.intentions.LoadLanguageComponents` The execution of the intention "Load Language Components".

If the configuration is valid, it is applied by calling the method `.applyComposition()`. Otherwise, an error is prompted with the message that has been stored in property *error* during the execution of `.checkComposition()`.

Validation of a Selected Composition

The check of the composition's validity is outsourced to the nodes of the tree at large extend. The method `.checkComposition()` only iterates over all nodes in the tree (using `.getAllNodes()`) and calls `.isValid()` on them. This function, in turn, consists of a list of conditions which represents invalid states (compare listing 5.12). If one of the conditions is met, the method returns "false" – an exception is the first condition which ensures that only selected nodes are checked. The method `.isEditable()` similarly checks whether a node can be selected (i.e. if the preconditions are fulfilled) or respectively deselected (i.e. if no other selected node depends on this one).

Note: Unselected nodes are relevant for the validity check as well. However, they are covered within the checking of selected nodes.

```
1  public boolean isValid() {
2    if (!this.isSelected) { return true; }
3    // check parent
4    if (this.parent.isNotNull
5        && !this.parent:AbstractComponentNode.isSelected) {
6      return false;
7    }
8    // check required children
9    foreach mandatoryNode in this.mandatory {
10     if (!mandatoryNode.ref.isSelected) { return false; }
11   }
12   if (this.isInstanceOf(ComponentNode1M)
13       && this.childNodes.where({
14         ~it => it.isSelected; }).size != 1) {
15     return false;
16   }
17   if (this.isInstanceOf(ComponentNodeNM)
18       && this.children.where({
19         ~it => it.isSelected; }).size == 0) {
20     return false;
21   }
22   // check dependencies (required, excluded)
23   foreach requiredNode in this.required {
24     if (!requiredNode.ref.isSelected) { return false; }
25   }
26   foreach excludedNode in this.excluded {
27     if (excludedNode.ref.isSelected) { return false; }
28   }
29   return true;
30 }
```

Listing 5.12: `composition.behavior.AbstractComponentNode`
The validity of a selected node is defined by the selection of other nodes on which the corresponding component depends.

Applying a Valid Composition

The method `.applyComposition()` is used to set the dependencies of the model that contains the MLE file according to the selected language composition. It is used for both, the removal and the adding of dependencies. In listing 5.13 the implementation of the method is presented:

```
1  public void applyComposition() {
2    // get model to which features are added
3    SModelInternal model = (SModelInternal) this.model/;
4
5    foreach component in getAllNodes() {
6      Language lang =
7        component.getComponentDescription().getLanguage();
8      if (component.isSelected) {
9        model.addLanguage(lang);
10     } else {
11       SLanguage rmv =
12         getLanguage(model, lang.getModuleName());
13       if (rmv != null) {
14         model.deleteLanguageId(rmv);
15       }
16     }
17   }
18 }
```

Listing 5.13: `composition.behavior.ComponentTree`
A language composition is applied to the model to which the corresponding MLE file belongs by adding and removing dependencies.

In line 3 the model is selected in its role of an internal model (i.e. an instance of `SModelInternal`). While the class `SModel` is used to manipulate the structure of model (e.g. by adding or removing nodes or listeners) the `SModelInternal` addresses meta-information such as dependencies, which is required in this context. The model referenced by `this.model` in line 3 is of type `SModelType` and is restricted to reading. The operation `/` transforms it into an `SModel` which is then casted into an `SModelInternal`.

Starting in line 5, it is iterated over all components of the tree. The method `.getLanguage()` in line 7 is called on the **ComponentDescription** to get a reference to the respective language-module. Depending on the state of the component, the method attempts to add or remove the retrieved language-module: If the component is selected, the module can simply be added to the internal model (line 9). Otherwise, the private helper-function `.getLanguage()` (not to confuse with the method in line 7) is used to get the module-import of the internal model. If this import exists, it is deleted in line 14.

Note: A description is a concept within the corresponding language module; thus, it can reference the module, which is not the case for the component node.

5.4 EFFICIENT EXPANSION INTERFACE IN MLE

New language components in MLE must be efficiently integrable into the DSL without establishing permanent dependencies. For the variability of the language, it is important that components can be added or removed without effect on concepts which not directly depend on them.

The integration of new components in MLE is presented on the example of an object-type's implementation. The **Featureset Type** draws great benefit from the `IItemContainer` interface. Therefore, it is well suited to demonstrate this type of extension points in MLE.

Note: Extension points are also called hotspots in literature

Extending objects in MLE is achieved by the Template-Method Pattern, which Apel et al. (2016) refer to as foundation for whitebox frameworks. By implementing the base class (also referred to as skeleton) as completely as possible, we minimize the effort for newly developed concepts that aim using the skeleton.

Featureset Structure

The structure of the **FeaturesetType** (further referred to as featureset) is composed of the concept **AbstractType** and the interface *IContainItems* (further item-container). From the abstract type it inherits the *name* which will be used as variable name. Also, the featureset brings its own structure which consists of the property *numRows* as well as the references *index* and *truth*. As item-container it receives the child group *items* and from the synchronization the property *id*.

Figure 5.7 illustrates the dependencies of the featureset's structure from the interface. For simplicity, the inheritance of the abstract type is not shown, but the name property is added directly to the list of members. Furthermore, the members of the interface are not repeated on the featureset.

Implementing the Item-Container Interface

As the name indicates, the featureset is primarily a set of features. The item-container interface is used in order to develop features as visible items (compare section 6.1). Only `.getItemConcept()` must be implemented by the featureset which simply returns the concept **FeatureType** (further referred to as feature). Accordingly, the feature is implemented as *IItemType* with no special requirements. In total, only one line of code has been written to register the featureset as item container which is line 4 in listing 5.14.

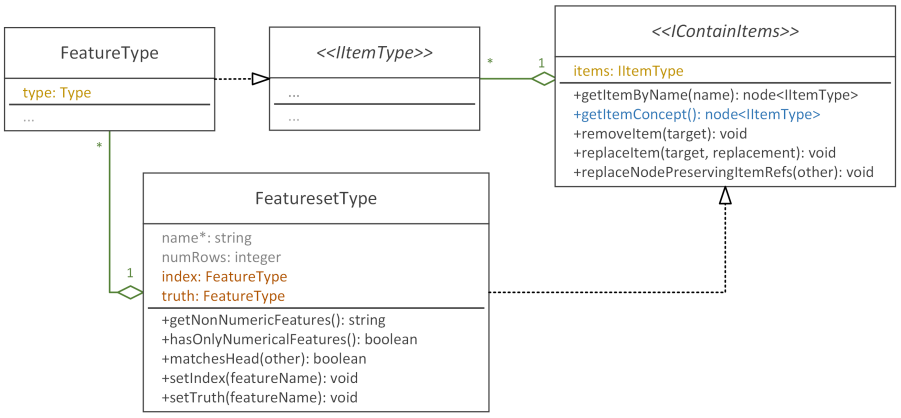

Figure 5.7: Structure of the FeaturesetType: The featureset is enhanced by the implementation of the skeleton IContainItems. Only the blue highlighted method must be implemented. The green highlighted aggregation-edges correspond to each other.

```
1 public concept<IItemType> getItemConcept()
2   overrides IContainItems.getItemConcept
3 {
4   return concept/FeatureType/;
5 }
```

Listing 5.14: `features.behavior.FeaturesetType` Implementaion of getItemConcept()

The pre-implementation of the item-container interface ensures that items can be referenced within object-related statements (compare section 6.3). This is realized by the computation of the scope of the **ItemRef** (further referred to as item reference) which requests the current object state. The object state, in turn, is stored in the object-related statement.

The item references are of the same type as the concept returned from `.getItemConcept()`. This type can as well be variable. For instance, the feature's type is inferred from its child concept *type*. Therefore, items can be used as operands in expressions and operations can be called upon them (for applications of item references see section 6.3.1).

6

USABILITY DRIVEN LANGUAGE DESIGN

MLE has two key paradigms in terms of usability: visibility and feedback. The visibility of actions, options, and objects aims at keeping the developer informed on his options at any point of the program. The provision of helpful feedback, not only after execution but already during programming, reveals the state of the program and its objects. Moreover, the two paradigms are incorporated in both, the language design and the tooling.

Language design is usually independent of the tooling provided by Integrated Development Environments (IDEs). A developer can choose the environment that suits his or her needs best. In the case of external DSLs developed in MPS, however, there is a close connection between the two. On the one hand, the language depends on the environment and cannot be used elsewhere. On the other hand, the language-designer might implement existing tools as well as completely new ones that improve the usability of the language. As a result, we can consider the language and the tools being one system.

With usability driven language design, we aim at optimizing the usability of the whole system. In section 6.1 we present the mechanisms of the visibility-paradigm which includes the optimization of code-suggestions as well as the ability to address concept-members like variables. Respectively, we present the feedback-mechanisms during programming and after program execution in section 6.2. The prompt computation of object-states is a requirement for mechanisms of both paradigms; thus, it is explained in detail in section 6.3.

6.1 VISIBILITY OF ACTIONS, OPTIONS, AND OBJECTS

The visibility-paradigm aims at keeping actions, objects, and options visible for the user at any point of the program. With actions and options, we refer to statements and expressions that are applicable in specific places of the program. It is a common behavior in projectional editors to use code-suggestions in order to find out which actions or options are valid in the current context. However, the utility of this procedure strongly depends on the language's optimization with respect to the suggestions, which we outline in section 6.1.1 for MLE. On the other hand, the visibility of objects refers to the content of objects, which is not entirely separable from the object-state. While the state is revealed to the user by feedback, some child-elements are made visible as language concepts dynamically. They can be addressed as if they were instantiated as variables. The advantages of being able to reference elements in this way are discussed in section 6.3.1.

Note: code-suggestion is nothing more than code-completion on empty input

6.1.1 *Language Optimization for Code-Suggestions*

Code-suggestions in projectional editors can reveal possible action and options. However, they can also overwhelm the user by being cluttered and inconsistent. In MLE, we made two design decisions in order to prevent those issues: First, only statements are allowed at the toplevel of a section. Second, statements constrained regarding their context. Another optimization for code-suggestions is the choice of enumerations over string-labels for the selection of options.

Toplevel-Statements in Sections

Only statements are allowed on the toplevel of sections. At sublevels, also expressions can be used which as well allow for operating on objects. Listing 6.1 illustrates the difference between statements and expressions at the toplevel:

```
1 section{
2   // pseudo-code demonstrating nested statements in MLE
3   edit obj {
4     addFeature "newFeature": obj.oldFeature.length() * 2
5   }
6
7   // similar pseudo-code using an expression
8   obj.newFeature = obj.oldFeature.length() * 2
9 }
```

Listing 6.1: Pseudocode: Nested statements in MLE in comparison to expressions.

Lines 3-5 demonstrate the design decision: At toplevel, an editing-statement is called, which accepts "obj" as parameter. Inside "edit", the statement "addFeature" takes a string (used as feature-name) and an expression as input. The expression consists of a multiply-operation on an integer-value (2) and an object-item on which the operation `.length()` is called.

On the other hand, line 8 semantically does the same using as lines 3-5 an expression on toplevel. While the single line is easily readable, it has some drawbacks regarding the code-suggestion and the overall visibility of actions: First, any variable inside the scope (i.e. any variable in the respective section) must be considered. This has two different kinds of suggestions as a result (statements and variables) as well as an increasing number of suggestions. Then, the introduction of a new member is done without any indication so that miss-spelling could lead to unintended instantiations (e.g. the intend could have been to overwrite a member "nowFeature"). Of course, the name and expression could also be passed as parameters of a method as follows:

```
obj.add("newFeature", obj.oldFeature.length() * 2)
```

However, this leads to the next problem in terms of visibility: Possible actions are hidden behind. One could argue, that it seems natural to add a member *on* the object, but this is not the case anymore if take other examples such as print, store, or split.

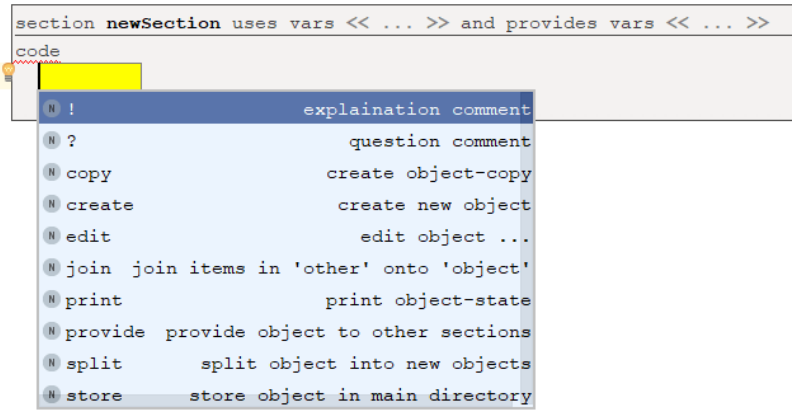

Figure 6.1: The code-suggestion in MPS presenting all toplevel statements.

In MLE, all statements that manipulate an object are nested into the edit-statement which is indicated by the three dots in figure 6.1.[1] Other object-related statements either read the object-state (print, provide, store) or instantiate new objects (copy, create, split).

1 An exception is the join-statement which currently add items of the second parameter into the first

Context-based Constraints of Statements

Hiding options that are incompatible with selections already made is another way of reducing the amount of suggestions to those that are relevant to the user. For instance, preprocessing can be applied on the features of a feature-set as follows:

```
edit dataObj: preprocess feature <feature>: <method>
```

Depending on the feature-type, different preprocessing-methods – such as label-encoding for strings or outlier-replacement for numbers – are applicable. The selection of an incompatible pair could be handled with an error message. However, it is also possible to prevent incompatible selections by filtering the methods according to their applicability as we do in MLE.

It is important to recognize that this reduction of code-suggestions is a tradeoff in this case: The fact that different preprocessing methods are displayed for different features can also irritate the user. Certain methods would then no longer be searched for at this location, as they were not displayed here the last time with the other feature types. Clearly, there are simple solutions to bypass this compromise such as graying out suggestions instead of hiding them completely. Unfortunately, we could not yet spend the time on changing code-suggestion menus.

Selection of Options from Enumerations

With the term "option" we refer to parameters of methods or functions which are used to choose an alternative behavior. For instance, a sort-method could accept a optional[2] boolean-parameter "reverse" that by default is set to "false". In most frameworks and internal DSL, many options are specified by a string-parameter which must match a specific format or an entry of a list of labels. Some examples are time-formats, language-locales, file-extensions, etc. In contrast, MPS offers an enumeration-type that can be implemented efficiently. The advantage of options realized via enumerations is that users of the DSL do not need to learn and remember the values but can recall them as they are available inside code-suggestions.

One example for an enumeration-type in MLE has been presented earlier in section 5.2.1 in which the children-type of a **ComponentDescription** is presented (see listing 5.1). Another one is the selection of the length-measure inside `.count()`, which is an operation applicable on string-features. It is presented in listing 6.2:

2 Some languages such as Java do not support optional parameters. Here, overloading is used to create different implementations – with and without the optional parameter – of the same method.

```
1  enumeration datatype length_measures
2
3    member type : integer
4    no default : false
5    default : <1-st member>
6    member identifier : derive from presentation
7
8    value 0   presentation characters     (default)
9    value 1   presentation syllables
10   value 2   presentation words
11   value 3   presentation sentences
```

Listing 6.2: `string.structure.length_measures`
An enumeration datatype with member-type integer. The default length-measure is "characters" (0).

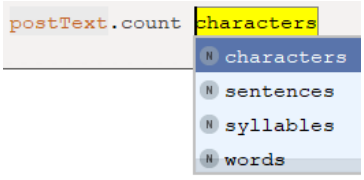

Figure 6.2: Code-suggestions for the property "lengthMeasure" of the string-operation "count".

The enumeration-type distinguishes between value and presentation. The length-measure is used as property *lengthMeasure* in **CountOperation** and the property-type corresponds to the type of the value (called "member type" in listing 6.2). The presentation-string, on the other hand, is presented to the user when code-suggestion is prompted (compare figure 6.2). The information that the presented strings are linked to integer-values internally is irrelevant to the user. The major advantage for developers is that the integer-representation allows switch-statements to be used instead of nested if-statements (see listing 6.3).

```
1  switch (this.measure) {
2    case enum/length_measures/.<characters>.value :
3        return this.charLength(targetString);
4    case enum/length_measures/.<syllables>.value :
5        return this.syllablesLength(targetString);
6    case enum/length_measures/.<words>.value :
7        return this.tokenLength(targetString);
8    case enum/length_measures/.<sentences>.value :
9        return this.sentencesLength(targetString);
10 }
```

Listing 6.3: A switch-statement that accepts an enumeration-property as input.

Regardless of the member-type, values should be compared using the enumeration-concept instead of constants (e.g. see line 2 in listing 6.3): From enumeration-concepts, the options are available as members, which themselves have the properties "name" (i.e. the presentation) and "value". Regardless the complex syntax, comparing enumeration-values this way is much more robust, since changes of the internal values have no effect on the condition and misspelling are excluded.

6.1.2 *Addressing of Items as Variables*

Note: A data-frame can be considered a matrix with features on the x-axis and instances on the y-axis

It was an early goal in the language-design to make features within a feature-set visible to the user. The Python Data Analysis Library (pandas) provides easy access to columns in its data-frame structure. A column can be addressed by simply using the column-name as a dot-operation on the data-frame (`df.column_name`). However, the user still needs to know by heart which columns currently are stored. It would certainly be a great benefit if they were suggested (e.g. by auto-completion) which would implement the principle "recognition rather than recall".

The suggestion of features present inside set is rather simple in MPS. The names and types are easily accessible inside the DSL. A new feature-set is always created from file which the python-server reads in. The server informs the DSL on the features (including names, types) which are stored as items inside the set. For statements which address those stored features, the information is already at hand.

However, the visibility of features within a feature-set implies the requirement that also the effects of feature manipulations must be visible directly. If in one statement a feature is added to the feature-set then it needs to be visible in the next statement. If the type of a feature is changed in the first statement, then it needs to be applicable to operations of the new type in the next statement. Also, it should not be able to address features in the current statement that were previously deleted. Meeting this requirement presents a major challenge: Not only must be known what effect a command has on the feature, but also how changes are recorded with respect to the feature-set.

Since there can be more data-types that contain addressable entities, a more general vocabulary is used in the following: The entities (e.g. features) are from now on called items and the containing objects (e.g. feature-sets) item-container or simply container. In this section, we present two applications that make use of the visibility of items and demonstrate the advantage: Item selections, which can be used to select multiple items at once, and item-dependent expressions, which allow for intuitive manipulation of items.

Item Selections

Item-selections statements enable the selection of multiple items. They are used to avoid the repetition of similar editing-statements for numerous items. For instance, there can be many numeric features in a feature-set that should be scaled in order to avoid confusion during the learning process. Instead of calling the scale-operation on every single feature, a selection could be utilized.

A selection consists of multiple selectors which are separated by commas. Each selector can refer to a single item or an item-range. This way, a desired subset can be selected easily from a set of items:

```
multiple items:      a, b, c, d, e
item-range:          a .. e
disrupted range:     a .. c, e
```

Another advantage of item-selections is that they are supported by notifications such as warnings that appear when ranges overlap (`a .. d, b .. e`), or errors which notify if the first item of a range is behind the last one (`c .. a`). For feature-selections it is possible to check for homogeneous feature-types inside the selection.

Item-Dependent Expressions

Item-dependent expressions make use of items as if they indeed were instantiated as variables: Items can be used as operators in binary operations (e.g. arithmetic operations), and operations can be called upon items depending on the item type. The utility of such expressions is again illustrated by the example of features, which can be edited or replaced by expressions:

```
! compute the flesch-kincaid-score
edit data
  add feature fleschKincaidScore from expression:
          [number] postText.count words / postText.count sentences
  edit instances on feature truthClass
          if (wordsPerSentence >= 20): [string] "longSent"
```

Figure 6.3: The editing-block contains two examples of item-dependent expressions: the calculation in the third last line, and the string-constant in the last line.

In figure 6.3 a new feature "wordsPerSentence" is added to the feature-set by the dividing the number of words in feature "postText" by the number of sentences in the same feature. The respective counts are determined by the count-operation. Some additional support is provided by revealing the type that is inferred from the expression.

The edit-instances statement in the next line directly used the newly added feature. Instances for which the feature "wordsPerSentence" is greater or equal 20 get a new value for the feature "truthClass". Here again, an item-dependent expression is used. A warning is indicated by the yellow, waved line below. It will notify the user, that the value of the expression is independent from the features (i.e. the items in the feature-set). As a result, the change of the value will be constant for all instances that contain the condition. In this example, however, the constant value is set on purpose.

6.2 FEEDBACK

The feedback-paradigm aims at revealing the state of single objects and the whole program. The state of single objects is defined by its properties, child-relations, and references. The state of the program, in turn, depends on interactions between objects and actions: A syntactically correct statement can be invalid, if an object passed as parameter to the statement, does not meet specific prerequisite. For instance, an item-container that is to be split should consist of more than one item. MLE supports users with feedback on both levels, regarding the state of objects and the execution semantics.

In this section, we present how feedback is provided in MLE for single statements and the program as a whole. This is done not alone after the execution of the program but also during programming. In section 6.2.1 we discuss examples of prompt feedback already given during programming. Then, in section 6.2.2 we explain the principle behind section feedback and give details to the implementation.

6.2.1 *Support Programming with Prompt Feedback*

As stated earlier, helpful feedback is already gained through the responsiveness of the application. We present a mechanism that is used to reveal the result-type of expressions similar to the way in which objects and actions are made visible. Subsequently, we present implementations of prompt warning and error-notifications.

Revealing Object-Types

In MLE, variables and expressions can reveal their type visibly to the user. To keep the editor clean, the revealing of types is optional, and the option stored as property in the MLE-file. It can be activated by the intention "Reveal Types". When activated, the reveal-type option presents the type of any expression or variable that implements the `IRevealType` interface. The type-string is enclosed in square brackets and shown in front of respective concepts as can be seen on the two expressions in figure 6.3.

Type revealing is used in various statements. Here, we highlight two specific use-cases: type-revealing in section headers and in item-dependent expressions. In section headers, the users' memory load is reduced by revealing the type of used variables. Users are not forced to remember the type of provided or used variables which is especially useful for code-comprehension. In item-depend expressions, on the other hand, the item-type can vary. Revealing the type not only

helps the user to remind the item-type, but also shows how items are transformed when used in operations or when operated on them.

The interface `IRevealType` requires the implementation of one method called `.getType()`. The return value is a type-object which can be MLE-types, kernelF's simple-types that are used in MLE (i.e. two concepts: `StringType` and `NumberType`). However, on every concept for which the type is defined via the type-system aspect has a property *type* which refers to the inferred type. Thus, the type of an expression can be returned in one line:

```
return (node<Type>) this.expression.type;
```

The only obstacle here is that the inferred type must be casted from the base-language type to the more specific concept type.

Interfaces in MPS have the same aspects as other concepts. Thus, it is possible to create the editor-aspect for the revealing of types inside *IRevealType*. The editor needs to be embedded in other editors at different positions. For that reason, it is created as an editor-component which can be used like a single editor-cell in other editors.

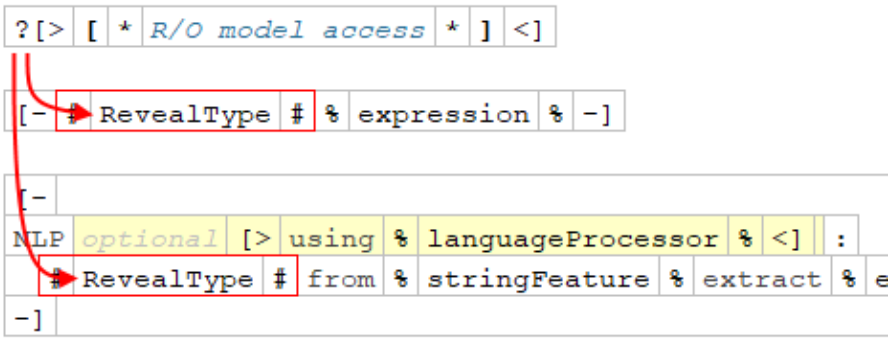

Figure 6.4: The editor for *IRevealType* (top) is stored in an editor-component which is used in the editors of ItemDependentExpression (middle) and ExtractFromLanguageProcessing (bottom).

The editor defined in *IRevealType* can be embedded in other editors as a component-cell (compare figure 6.4) which is indicated by the surrounding `#` . The reuse of the component ensures that the revealed type is displayed consistently in all places.

Prompt Warnings and Error-Notifications

Whenever possible, MLE notifies the user not only on syntactical errors[3] but on semantical errors as well. This kind of feedback requires

3 In MPS users are unable to produce errors regarding the concrete syntax. However, they still can violate the static semantics (see section 4.4.3)

up-to-date object-states (see section 6.3) in order evaluate the semantics of a section (i.e. to consider the effect of previous statements). In this section, we give two examples for prompt warnings and errors that GPLs often occur only at runtime.

SPLIT The split-statement notifies the user when the related object is, or parts of the split will be empty (see figure 6.5). To do so, the statement requests the current object-state of the related object and checks the number of units to be split. It then pre-calculates the number of units, if possible[4], and warns the user in the case of an empty part.

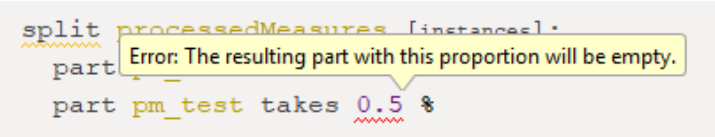

Figure 6.5: State-dependent feedback provided before runtime.

SET TRUTH The model-object in MLE (which is used to apply ML on a feature-set) is fitted on the first set that is passed. Consequently, test and training-data must consist of the same features. The feature-set which is used to fit the model must designate the feature[5] that serves as truth-class which is done by the sublevel-statement "set truth" inside an edit-block. If this step was missed, MLE notifies the user with an error-notification.

When implementing these notifications, it is important that the system does not check whether certain statements have been made beforehand, but instead reads the actual state of the respective object at this position. In this way, the notifications also work across sections and across different programs. In fact, variables that exist on the server as well are not passed from previous sections but, in favor of synchronicity, from the server itself.

In principle, most domain-specific notifications could as well be provided in frameworks and internal DSLs based on the presence of previous statement if there was access to the tooling of IDEs. Also, the language-extensions would be bound on supported IDEs which, eventually, leads to systems very close to external DSLs.

4 Some statements manipulate the number of items unpredictably. The number of units is then inverted in the current state to indicate that it is unknown.

5 Strictly speaking, the truth class is not a feature, since features are characteristics of instances from which the truth class is to be inferred. However, there are inconsistencies regarding names in MLE that have not been solved yet.

6.2.2 *Piecewise Feedback from Sections*

The feedback after execution of program sections is of great importance. It should enable the user to determine whether the program has been executed as intended. MLE enables any statement to inform the user on changes that have been made on an object-state. Furthermore, the print-statement is used to represent the state of an object in a comprehensive manner.

Statement Feedback

When executing sections, the individual statements are requested to submit feedback. Toplevel-statements that provide environments for sublevel-statements forward the request accordingly. The responsible code-line inside the respective statement's interpreter is illustrated in figure 6.4.

```
1 node.ancestor<concept = Section>.results.addAll(
2     statement.getFeedback());
```

Listing 6.4: The collection of feedback from statements inside the interpreter.

The section has a child-group *results* of type **AbstractOutput**. Accordingly, the return value of the method `.getFeedback()` must a concrete object of such concept. All outputs are presented line-by-line below the section[6]. It is recommended to use only so-called simple-message output for statements in order to highlight the output of visually more complex print-statements (figures, tables, graphs, etc.). However, there are no constraints that force developers to act according to the recommendation. On the contrary, the statement "find estimator" returns figures that illustrate learning-curves of the top three estimators for the respective feature-set (compare figure 6.6).

As stated earlier, there was not enough time to implement all aspects we had planned for section feedback. Two such aspects, which we think would greatly improve the use of feedback, are:

1. The obligation to subtle but extensible statement-feedback (i.e. all statements except print). The output is then constrained to be a simple string with an upper-bound of characters. However, more complex visualizations could be collapsed by default and expanded on demand.

6 Strictly speaking, they are presented at the bottom of the section. However, the visualization of sections draws a frame around the header and the code block in order to separate the program from the output.

2. The highlighting of corresponding statements. When a specific output is hovered or in focus, the statement that provided such output is highlighted (e.g. by darkening the background color). This way, even with complex sections, the output can be easily assigned.

```
section extractFleschKincaidScore uses vars [featureset] wordCount
code
  ! collect counts required to compute score
  edit clickbaitdata
    add feature numSyllabels from expression: [number] postText.
    add feature numWords from expression: [number] postText.count
    add feature numSentences from expression: [number] postText.

  ! compute the flesch-kincaid-score, remove previously coll
  edit clickbaitdata
    add feature fleschKincaidScore from
        expression: [number] 0.39 * (numWords / numSentences) + 1
    edit instances on feature truthClass if (wordsPerSentence >=
    preprocess feature fleschKincaidScore: fill empty values wit
    delete features numSyllabels .. numSentences

  print clickbaitdata
  store clickbaitdata
```

```
Feature 'numSyllabels' of type number was added to featureset 'in
Feature 'numWords' of type number was added to featureset 'instan
Feature 'numSentences' of type number was added to featureset 'in
Feature 'fleschKincaidScore' of type number was added to feature
Empty values in feature 'fleschKincaidScore' were filled with val
In total, 3 features were deleted from featureset 'instances': nu
instances[9 dimensions, 19538 rows]:
```

name	type	#empty
id	integer	0
postMedia	stringlist	0
postText	string	0
truthClass	string	0
easyLanguage	float	0
numMentions	integer	0
numHashtags	integer	0
numDots	integer	0
fleschKincaidScore	float	0

Figure 6.6: Output of a section. The table resulting from the print-statement directly catches attention.

Print-Statement Output

The print-statement is used to visualize all important aspects of the respective object's state. At the bottom of figure 6.6 the output of a printed feature-set is shown. In contrast to the beforehand statement-outputs, this output is visually more complex and catches the user's attention.

The output of a feature-set reveals its shape (i.e. the number of dimensions and instances) and lists the type and the number of instances without value for each contained feature. The table is created with a swing-component cell which can be used to output elements created with Java's UI toolkit.

Outputs provided by the print-statement are descendants of *AbstractOutput*. The abstract class is used only to collect all outputs of one section in one child-group. In order to be applicable to print-statements, the **FeaturesetType** implements the *ICanBePrinted* interface which consists of the method `.getOutputType()`. The method returns the type of concept that is used to output the respective feature-set when printed.

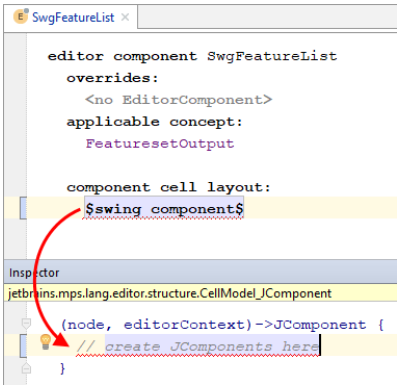

Figure 6.7: The swing component cell is defined in the inspector window.

The output-types of the current version of MLE have only been implemented rudimentary due to the focus on up-to-date object-states and prompt notifications. We plan to focus on more sophisticated output-types that provide for better feedback in future.

6.3 PROMPT COMPUTATION OF OBJECT-STATES

The prompt computation of object-states is utilized for the provision of both, visible items and detailed feedback. The special requirement of both applications is that the object-state must be calculated before the actual DSL code is executed. The prompt computation of the state can remarkably contribute to the usability of a DSL. Six out of ten usability heuristics provided by Nielsen (1994) may profit from the object-state:

- *Minimize the users' memory load*: The heuristic is also described as the principle "recognition rather than recall". Users do not need to recall everything that is inside an object but recognize it from suggestions made while the object is manipulated.
- *Feedback*: We consider the up-to-date state of objects itself as valuable feedback. The user gets confirmation that the previous code works as intended since the new state influences constraints and actions on the object.
- *Speak the user's language*: More precise hints can be given by the DSL since more information is available. Also, more specific messages (like warnings or errors) can be provided which allows for more natural language possible.
- *Good error messages*: Error and warnings can be provided, that could not be detected otherwise. For instance, the DSL could warn if all items are accidentally removed from a container due to an imprecise condition.
- *Shortcuts*: With information on the exact object-state, more and better shortcuts such as intentions or quick-fixes can be provided. For instance, preprocessing-steps could be inserted, or common subsequent steps suggested.
- *Prevent Errors*: Some conflicts can only be detected, if it is known how objects were manipulated before. For instance, only matrices with fitting shape are accepted in a matrix-operation.

In this section, we outline different approaches for the prompt computation of the object-state (section 6.3.1) and give then details on the implementation for this thesis.

6.3.1 *Approaches to Track the Object-State*

Typically, all occurrences of a variable reference the same object that has one state. The state may change during the execution of the program (and generally does or else the variable may not be required).

However, to provide an updated object-state *based on previous statements*, changes need to be tracked already while editing the program code. In this thesis, four approaches for keeping the object-state up to date during code editing were considered:

1. Manipulate Referenced Object: The referenced object by default has one state which here is manipulated by each statement.
2. Store Local Copies: Copies of the altered object-state are produced for each editing-statement at initialization.
3. Compute On Demand: The object-state is computed every time when a statement is edited.
4. Update Altered Copies: A combination of approaches two and three.

Figure 6.8 shows a simplified illustration of a section, which will be used to visualize the different approaches. It is depicted next to a screenshot of a section which (amongst others) uses the variable "instances". All lines in the illustration in which "instances" is manipulated are highlighted yellow. Note, that the edit-statement itself does not manipulate the object but initiates an environment in which editing-statements can be placed. Therefore, the third last line is not highlighted in the illustration and the following lines are indented.

Figure 6.8: A simplified illustration (left) and screenshot (right) of a section. Statements which manipulate the first input variable (instances) are highlighted in the illustration.

In the example section, we assume that the variable "instances" references an item-container (compare section) and the change of items inside the container is to be tracked. This helps to explain the approaches in less generic terms, without using a specific example that is constrained to specific use-cases.

Manipulate Referenced Object

In the simplest case, the manipulation of a referenced object is executed directly on the object itself. To keep the object state up to date, the statement's effect is computed on the original object-state

(indicated by the arrows in figure 6.12) as soon as the statement is completed. This manipulation of the object-state comes with several obvious flaws:

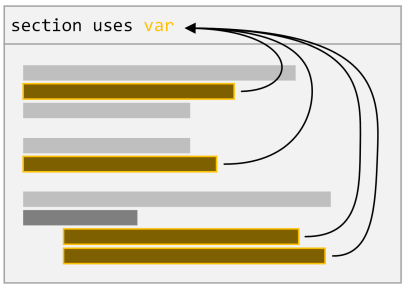

Figure 6.9: Default Setup. Editing-statements directly manipulate the object-state of the variable.

If the first statement deletes an item from the container all following statements won't be able to address it anymore (which is wanted). However, also the reference in the first statement itself will be broken as soon as the container is manipulated, because the referred item is removed from the object. The same problem occurs when an item is deleted by a statement at the end of the section: If that item was referred to earlier, the respective references will be broken. On the other hand, items that are added in one statement can be addressed in previous statements, which also is not desired.

It becomes clear that more than one object-state is required to avoid such contradictions. The following approaches do this in various ways.

Store Local Copies

In this approach, copies of the object-state are stored in each statement that refers to the object. When a statement is created that relates to the object, the object-state is computed for the position of the new statement. To do so, all statements in between need to reveal how they alter the object-state (e.g. remove an item or change the item's type). Since the new object-state is computed on the fly, it is important that the effects of editing-statements are calculated efficiently. Otherwise, delays could become apparent while programming – where no computations are expected by the user.

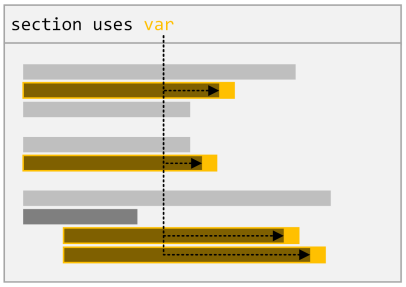

Figure 6.10: Storing Local Copies. A copy of the object-state is computed for each statement by traversing all previously created statements.

The object-state for the current context is stored directly inside the statement (see the yellow squares in figure 6.10) and considers all *previously created* statements. This works well, if code is written from top to bottom. However, problems occur if the user notices that another editing-statement was required, which was not thought of before. Assume the editing-statement in the middle of the section in figure 6.10

was entered after the ones at the end: The object-state at the end is then computed independent from the statement in the middle. This means that, for instance, the change of an item-type would not be recognized which could lead to semantical errors such as calling a string-operation on a number-feature.

The problem here is, that only previously created statements (instead of all previous statements) determine the object-state. To solve this issue, the state would always be recomputed when the code before the respective statement changes. However, tracking such changes to update object-states is a rather complex task, so that less complicated and less costly approaches are examined first.

Compute On Demand

This approach does not store the current object-state inside the statement but computes it every time the user edits the statement. This is achieved by searching for the closest, previous statement that manipulates the same object. This statement is then demanded to reveal its effect on the previous object-state, which is computed in the same way. The recursive procedure stops, when the closest previous statement is the imported variable (compare figure 6.11). Copies of items which are referenced inside a statement are as well stored that statement. This way, only an item-copy and not a copy of the whole object-state needs to be created.

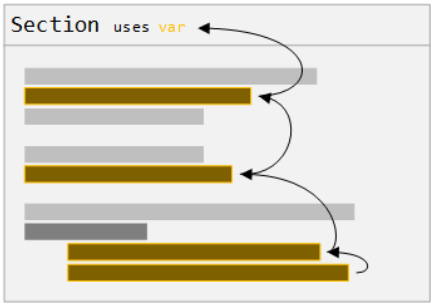

Figure 6.11: Compute On Demand. The effect of previous editing-statements is added recursively to the object-state until the imported variable is reached

During the implementation of this approach two problems became apparent: The first one is the lack of clarity. The object-state is spread over all editing-statements inside the section. It needs to be composed of all statements which is hardly comprehensible when inspecting the code in which the DSL is written. While this is no problem for a user of the DSL it becomes a major obstacle for developers who are going to extend it with new object-types.

The second problem is related to the type-system in MPS: The type-system in MPS uses the function `typeof(object)` to infer the type of a specific object (compare section 4.4.3). However, the object that holds the current state is created only temporarily which means that it is floating outside the Abstract Syntax Tree (AST). For some unknown reason (which should be related to the mechanism used to infer the

type of objects) the type-system does not work for such floating objects. As a result, MPS cannot compute whether an item is applicable in a specific context if the condition depends on the item-type.

For problems related to type-system solutions could be found. However, those solutions would add to the lack of clarity which strongly conflicts with the aim to provide efficient expansion interfaces.

Update Local Copies

This approach combines storing local copies with computation on demand. The local object-states are updated whenever the editing-statement is altered.

This approach meets all requirements to make items visible. For the computation of the object state mbeddr's interpreter-framework can be utilized (which also handles the execution of a section). This is particularly beneficial because the implementation of the object-states updates in higher-level concepts (i.e. more abstract concepts) simplifies the adaption not only for other item-containers but all kinds of object-types. This means that not only item-related state changes are included but also other members or property changes.

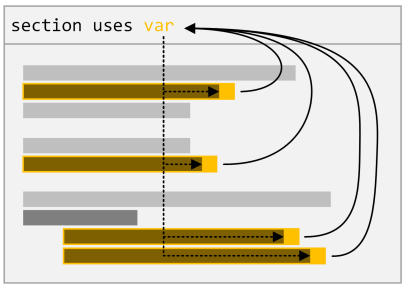

Figure 6.12: Update Local Copies. All editing-statements refer to the same object. Changes of the object-state are computed up to the respective statement and stored there.

6.3.2 *Implementation Details*

There are two events which trigger the computation of the object-state on a specific position: Either the constraints-aspect of any parameter that is to be edited requests it, or the checking-rules inside the type-system aspect when it is noticed that the conditions have been altered. In any case, the method `.getObjectAtPosition()` is called on the respective section.

In this section, we explain first how the method works. Then we outline the mechanism that keep the execution efficient. Finally, we discuss how the object-state is used.

Computing the Object-State at Position

The method `.getObjectAtPosition()` is presented in listing 6.5. It is called on the section and requires two parameters: the object whose state is to be computed and the position where the state is required (i.e. the statement requesting the new state).

To compute the object-state, all previous statements need to be evaluated. They are collected by `.getStatementsBefore()` in line 6, which simply returns a list of all toplevel-statements prior to the position. Since the position itself could be a sublevel-statement (e.g. an editing-statement inside an edit-block), also a partial statement is constructed inside `.getPartialStatement()` line 9. This is done by creating a copy of the toplevel-statement which contains the position and removing all sublevel-statements starting with the position itself. The partial statement is added to the list of prior statements.

In lines 15-16 the interpreter and an interpreter-context are initiated. In line 16, the interpreter is initialized with the group "arithmetic" which includes all interpreter-files inside MLE, as well as the kernelF-framework (compare section 4.5) whose expressions and types are used in MLE. The MLEContext is an interpreter-context with one additional flag: the boolean-property "local" is used to signal whether the interpretation takes place during code-execution or to compute the object-state. The influence of this flag on the interpretation is explained in the section on efficiency below.

In lines 20-22 the in-ports (i.e. the variables used by the section) are interpreted with the previously defined context. This is done to copy the initial object-states into the temporary interpreter-environment. Moreover, the collected statements are interpreted in lines 25-29: Statements that only add visual structure to the code are excluded in line 26 (since interpretation is undefined for them), before the remaining statements are evaluated in order to compute the new object-state.

The object-state at the final position is stored only in the temporary environment. It contains the object-state of all objects in this section. The desired object is requested in line 32 and finally returned as result of the method. Still, the object-state is not connected to the actual AST. It is up to the calling functions whether to store the object-state or to use it only temporary. The usage of the object-state is discussed in the respective section below.

```
1  public node<AbstractType>
2    getObjectAtPosition(node<Type> object, node<
         AbstractStatement> pos)
3  {
4   // add previous statements
5   nlist<IToplevelStatement> statementsBefore =
6     this.getStatementsBefore(pos.ancestor<concept =
          IToplevelStatement,+>);
7
8   // add partial statement
9   node<IToplevelStatement> partialStatement = this.
        getPartialStatement(pos);
10  if (partialStatement.isNotNull) {
11    statementsBefore.add(partialStatement);
12  }
13
14  // initiate interpreter
15  InterpreterEvaluationHelper evaluator =
16    new InterpreterEvaluationHelper("arithmetic");
17  MLEContext context = new MLEContext(true);
18
19  // load inports
20  foreach inport in this.inports {
21    evaluator.evaluateWithContext(inport, context);
22  }
23
24  // evaluate statements
25  foreach expr in statementsBefore.where({
26    ~it => !it.isInstanceOf(VisibleStructureStatement); })
27  {
28    evaluator.evaluateWithContext(expr, context);
29  }
30
31  // return object from interpreter-environment
32  return (node<AbstractType>) context.getEnvironment().get(
        object);
33 }
```

Listing 6.5: `core.behavior.Section`
The method used to compute the object-state for a specific position.

Efficiency of Object-State Computation

The object-state is requested while users of the DSL edit statements. Therefore, it is important, that the object state is computed efficiently. There must be no noticeable delay caused by the computation, since that delay would affect every concept that users try to instantiate.

Concepts that effect the object-state implement the Interface **IMayAlterObject** and thereby the method `.applyLocalChanges()`. In the interpreter-framework, the pattern shown in listing 6.6 is used several times to compute the object-state or (depending on the property *local* of the context) run the section without constraints.

Note: Other methods of IMayAlterObject only concern the communication with the server and are not considered here.

To keep the duration of object-state computations minimal, the method `.applyLocalChanges()` does not compute alterations from the previous state but infers the effect of the commands. For instance, on features the preprocessing-function "fill empty values" can be used to replace empty values with other values. Instead of computing the new values on the server (e.g. by interpolation or calculating the arithmetic mean), `.applyLocalChanges()` infers that there will be no empty values after the calculation. Similarly, label-encoding does not have to be performed to conclude, that the resulting feature will be of type integer instead of string. These conclusions are simply applied to an object that is passed into the method and which, finally, represents the new object-state.

```
1  // efficiently compute object-state changes
2  statement.applyLocalChanges(context, object);
3
4  if (!((MLEContext) context).local) {
5    // apply all changes (locally and on server)
6    ...
7  }
```

Listing 6.6: Common construct inside the interpreter-framework used to distinguish between fast object-state computations and the execution of a section.

The condition in line 4 of listing 6.6 converts the interpreter-context into an MLE context in order to access *local*. The respective code block is only executed, if *local* is set to false. The property-name "local" was introduced in an early state of MLE to separate client-server communication from other code. Now, it is misleading, since also locally computations are put inside that block to separate them from the more efficient computations. Also, the method-name "applyLocalChanges" is misleading in the same matter. A clean solution would be the use of a separate interpreter for the fast computation of object-states.

Usage of the Object-State

The usage of the object-state depends on the context. As stated earlier, the object-state is requested either via the constraints-aspect (i.e. when the scope of concepts is computed) or when checking-rules are applied (inside the type-system aspect).

When checking-rules request the object state, then it is only used temporarily and dropped afterwards. This is for two reasons: First, there is no need to store the current state, since it will not be referenced by warnings or error-messages. All required information is extracted and transformed into a string which servers as notification

to the user. Second, checking-rules are applied in read-mode which makes it impossible to manipulate the AST.[7]

The object-state needs to be stored somewhere in the AST when requested by the constraints-aspect, since its members might be referenced then. Therefore, it is stored directly in the statement, that references the respective state. However, the object-state must not be simply overwritten if one already exists, but it needs to be replaced properly using `.replace with()` instead of `state x = y` in order to preserve references to the state.

```
1  public virtual void
2    replaceNodePreservingItemRefs(node<IContainItems> other)
3  {
4    node<IContainItems> otherCopy = other.copy;
5
6    // replace references
7    node<Section> sec = this.ancestor<concept = Section>;
8    foreach item in otherCopy.items {
9      node<IItemType> found = this.items.findFirst({
10         ~it => it.name.equals(item.name); });
11     if (found.isNotNull) {
12       found.replaceReferences(sec, item:IItemType);
13     }
14   }
15
16   // replace this object with copy of other object
17   this.replace with(otherCopy);
18 }
```

Listing 6.7: `core.behavior.IContainItems`
ItemRefs refer to object-states. Thus, new object-states must update references.

If the respective object is an item-container (i.e. it implements *IContainItems*), the statement could include references not only the object-state but to its items. Those item-references must be preserved as well, which is achieved by `.replaceNodePreservingItemRefs()` called directly on the object-state. It replaces all references that refer to the current object-state by corresponding ones of the new state, before also the old state is replaced by the new one.

7 However, any code can be executed in both, read and write-mode. Therefore, it can be convenient to check programmatically if the AST is editable via the ModelAccess-class: `ModelAccess.instance().canWrite();`

In line 4, a copy of the new object-state is created, to ensure that the local copy will not be affected by any other operations of the interpreter. The loop starting in line 8 iterates over each item of the other container and searches for corresponding one in the current container. If such corresponding item is found (what may not always be the case), it is used to replace the link in all incoming references of the corresponding section to the new item (`.replaceReferences()` in line 12).

Finally, the current object-state is replaced by the new one after all references are replaced. There still may be broken references, if the new object-state does not contain all referenced items of the old one. The resulting errors, however, correspond to incorrect semantics. Such references should not exist anymore (e.g. due to deletions).

Part III

EVALUATION

7

MLE USABILITY STUDY

With Machine Learning Evolves (MLE), we attempt to open Machine-Learning (ML) to domains distant from computer science. Regarding the usability of DSLs, we focused on providing the visibility of objects and actions as well as feedback (compare chapter 6). Thus, the user should get an insight into the data on which he works and which interaction possibilities are available in the respective context.

In this chapter we report on an qualitative experiment that we conducted in order to assess the overall usability of MLE based on the 10 usability heuristics (Nielsen, 1994). According to the template of the Goal/Question/Metric method, we define the goal of the study as follows: We analyze MLE for the purpose of its evaluation with respect to usability from the viewpoint of a developer in the context of the DSL's development. Since external DSLs are deeply integrated into the respective language workbenches, we consider both together as one system which is to be evaluated on the basis of typical ML tasks. As control condition serves the combination of the python frameworks pandas and scikit-learn (which are as well used on MLE's local server on which the ML algorithms are processed) together with pyCharm, a python IDE with very similar tooling to MPS. The experiment design is a within-participant design meaning that each participant performs with both systems. Eight participants were recruited from which fife were unfamiliar with ML. The systems were rated by the participants based on the usability heuristics after both tasks had been performed.

The details on the experiment design are explained in section 7.1 of this chapter. Subsequently, the collected data is analyzed in section 7.2 and finally discussed in section 7.3

7.1 EXPERIMENTAL DESIGN

In this section we provide all information related to the experiment. This comprises a refined definition of the goals, the recruiting process of participants, the material used during the experiment, the variables involved, the experiment design, the procedure, and deviations from that procedure during the task execution.

7.1.1 *Goal*

In the introduction of this chapter we defined the objective of this study: We analyze MLE for the purpose of its evaluation with respect to usability from the viewpoint of a developer in the context of the DSL's development. In this section, we are going to refine this objective by addressing sub goal: The usability is to be evaluated with respect to the different prerequisites of participants with regard to ML. On the one hand, we are looking for feedback from participants familiar to ML in order to examine whether the structure of DSL is compatible with common practices. On the other hand, we want to check whether participants who are inexperienced with ML can find easier access via the DSL. Thus, the overall goal can be subdivided to reflect two facets:

- We analyze MLE [...] with respect to usability for *programmers unfamiliar with ML* [...]
- We analyze MLE [...] with respect to usability for *programmers familiar with ML* [...]

Since usability is a broad term, we narrow down the goal to the evaluation of the DSL Nielsen according to the 10 usability heuristics (1994) which we summarize in point (5) of section 7.1.3. A secondary quality focus which is not directly mentioned in the goal-statements is efficiency. Nielsen only marginally discusses efficiency with respect to shortcuts. We are interested whether participants can perform with the DSL as fast as with established ML frameworks. However, the early state of the DSL as well as the restricted time for the experiment inhibit the proper evaluation of two focuses. Thus, we use simply measures (see section 7.1.4) to evaluate efficiency as an additional indicator of the DSL's usability.

7.1.2 *Participants*

The study was planned as a qualitative study; thus, a total of eight participants was considered sufficient. According to the study goals, participants with and without previous experience in ML were required. For this reason, we searched the first four participants with an interest in the topic, regardless of their previous knowledge with ML or computer science, among fellow students. From those four participants none were familiar with ML. The other four participants were recruited from the Chair of Intelligent Software Systems at the Bauhaus University Weimar, because of their previous experience with ML.

The level of education between participants was heterogeneous: one participant only begun studying computer science, while others

already finished their master's degree. Also, another participant was completely without background in computer science but interested in data-science. However, all participants of the study either already had a university degree or were on their way to acquire it, which suits the target group of MLE which are experts of different domains with interest in various aspects ML.

The participation in the study was voluntary and participants were aware that they could cancel the experiment at any time. The only incentive for participation was a local craft beer, which could rather be seen as a small thank-you than an expense allowance.

7.1.3 *Experimental Material*

The material used in the study were: (0) computer and recording devices (1) a questionnaire to determine the prior knowledge of participants, (2) the tools provided to perform the ML tasks, (3) introductions to both tools based on a simple ML example, (4) two ML tasks representing common use-cases for classification, and (5) a questionnaire to evaluate the usability of the DSL after the experiment.

(0) Computer and recording devices

The study was conducted on a computer equipped with two screens. On one screen the terminal output of the python server was monitored, while the other was used to process the tasks. A browser was started in advance in which three websites with documentation on MLE, pandas and scikit-learn were opened. Links to the main pages of such websites were also provided as shortcuts below the search bar. The search history was cleaned at the end of each session. A tablet was used for audio-recording.

(1) Questionnaire for the Assessment of Prior Knowledge

The questionnaire used to assess prior knowledge of participant in programming in general as well as ML in particular is based on a paper of Feigenspan et al. (2012) which examines the measurement of programming skills among students. Feigenspan et al. found that, out of 18 questions, two questions explain 24.1% of the variance in the number of correct answers. We utilized those two questions in order to assess experience in programming and added four questions with respect to python and the ML tools used in the study. The resulting questionnaire includes the six questions presented in table 7.1.

Subject	Question	Abbreviation
programming	How do you estimate your programming experience compared to your colleagues?	prg.compare
	How do you estimate your experience with the logical programming paradigm?	prg.logical
python	How do you estimate your experience with the python programming language?	py
ml	How do you estimate your experience with machine-learning?	ml.general
	How do you estimate your experience with the python framework pandas?	ml.pandas
	How do you estimate your experience with the python framework scikit-learn?	ml.sklearn

Scales are integer values from 1 (very inexperienced) to 5 (very experienced)

Table 7.1: Prequestionnaire used to assess prior knowledge with respect to programming in general, python in general, and ML with scikit-learn and pandas.

(2) Tools for Performing ML Tasks

As tools for performing ML tasks, the DSL presented in this paper as well as python in combination with PyCharm (i.e. a python IDE) are used. Both, MLE's language-workbench MLE and PyCharm, are products developed by Jetbrains; thus, they provide similar tooling (such as code-completion and intentions) which increases comparability between both conditions.

To perform ML with python in PyCharm, the python frameworks pandas (for data-analysis) scikit-learn (for ML algorithms) were suggested. However, the participants had the freedom to also use aids other than these frameworks.

(3) Introduction to ML Tools

To introduce participants to the tools used in the study, small tutorials were prepared in advance. The tutorial task was to classify irisflowers according to their dimensions. The dataset was introduced by Fisher (1936) with respect to the linear discriminant analysis and has become a common test case for classification with ML.

In each introduction, the actions were demonstrated by the supervisor and replicated by the participant. The steps were similar for both tools. They consisted of the loading of data, the distinction of features and truth class, the standardization of measurements, the splitting of data into train and test set, as well as the application of a classifier on the resulting data.

(4) ML Tasks

We designed two tasks, each representing a common ML use-case. The first task (Clickbait Classification) was based on the clickbait-challenge 2017 (Potthast et al., 2018) and focused on the extraction of numerical features from text-samples. The second task (Brain-Scans Classification) was based on a dataset kindly supplied from ongoing research at the University of Magdeburg. Here, the challenge has been the preprocessing of the data such as scaling, condensing measurements.

The two tasks were each explained in a two-page task description. The first page briefly introduced the topic and then described the datasets on which the tasks were to be solved. On about half of the page the datasets were displayed graphically (i.e. as table). The second page divided the use-case into six subtasks. The datasets were reduced in their scope in order to avoid long computation times during the study. We justify the reduction of the datasets by the fact that our goal is a usability evaluation. For such, the actual results of the ML algorithms are not relevant. The usual tasks, such as preprocessing data and extracting relevant features, remain unaffected. In the following paragraphs, we briefly describe the shape of the datasets.

CLICKBAIT DATASET The overall goal was to classify whether tweet-posts are clickbait or not. Data was to be loaded and cleaned (subtask 1), features to be extracted from the post-text (subtasks 2-5), and ML to be applied on the extracted features (subtask 6). The extraction included: counts of words and sentences in string features, arithmetic expressions based on counts, presence of wordlist-entries in a string feature, and the extraction of word 3-grams from string features. The dataset consists of two *.jsonl-files. The first one, tweets.jsonl, contains twitter-posts and linked articles retrieved from twitter-accounts of english news-publishers. Besides the post and article texts, also meta-data of both (such as keywords or post-media) are stored so that tweets.jsonl consists of 9 features in total. The other file, truth.jsonl, consists of ratings for the twitter-posts which determine whether posts were classified as clickbait due to crowd-sourced annotations. Also, statistical measures are stored for the annotations (i.e. truthMean, truthMedian, etc.). Both files consist of about 19500 json-objects, which were reduced to a total of 2000 for the experiment.

BRAIN-SCAN DATASET The overall goal was to classify whether measurements belong to a comprehension or not. Data was to be loaded and preprocessed (subtasks 1-4 and 6), and ML to be applied on the processed measurements (subtask 5). The preprocessing included the scaling of measurements grouped by users, the replace-

ment of outliers, and the aggregation of measurements based on conditions on the respective instance. The dataset consists of a *.csv-file containing brain-activity scans. Such scans were collected during the processing of two different tasks, code-comprehension and syntax-debugging, as well as in rest-phases in between. Each scan contains more than 1400 measurements and additional meta-data such as a task ID, processing time, etc. The dataset was reduced to avoid long computations: only 50 measurements per scan were considered in the study and only about 5000 out of 15000 scans.

(5) Questionnaire for the Usability Evaluation

The questionnaire used for the usability evaluation is based on the ten usability heuristics suggested by Nielsen, 1994 (1994). Each heuristic was described shortly and participants rated the compliance with these heuristics for both tools, the DSL and the python frameworks, on a scale from 1 (very poor) to 5 (very good). Additionally, participants were asked to name the three biggest advantages and disadvantages of the DSL.

7.1.4 *Experiment Variables*

In this section we explain the variables that are involved in the experiment grouped by the treatments.

Pre-Questionnaire

The questionnaire introduced in section 7.1.3 is used to measure the prior knowledge of participants regarding three aspects of programming:

1. The first two questions, *prg.compare* and *prg.logical*, are used to assess participants' general programming skills which is assumed to explain between-participants deviations during the experiment. The formula suggested by Feigenspan et al. is used to calculate a single value from both answers:

$$prg = 0.441 \cdot prg.compare + 0.286 \cdot prg.logical$$

2. The third question, *py*, is used to assess python programming skills. The value is considered to be explanatory with respect to between-participants deviations, as well, since python is one of both tools. The variable value is accepted without further processing.

3. The last three questions, *ml.general*, *ml.pandas*, and *ml.scikit*, are used to control the experiment groups: For each task sequence, we targeted for one participant with and one without prior ML experience. We decided, not to weight the values of the *ml*-questions, since there was no basis on which to rely such weighting.[1] Instead we use the arithmetic mean to aggregate the values:

$$ml = \frac{1}{3}(ml.general + ml.pandas + ml.python)$$

Experiment

Independent variables are given by the tools, the task descriptions, the tool order, the task order, and the processing duration. Although our main objective is to evaluate the usability according to Nielsen (1994), we measured variables that can be indicators for efficiency:

1. The number of subtasks solved per task (integer value ranging from 1 to 6). Only complete and correctly processed tasks are counted. The variable is denoted as $\#Suc_{<tool>}$.
2. The number of searches per task (integer value). Only searches in the search bar of the browser or website were counted, or, with respect the DSL, the intentions to find specific information inside the documentation. The variable is denoted as $\#Sear_{<tool>}$.
3. The response time for each subtask (to the tenth second). Response times were not measured by annotations of the supervisor. The variable is denoted as $t^{<subtask>}_{<tool>}$.

Since the values of the respective variables have the same range for both tasks, it is tempting to comparable variables as well between the tasks. It is important, however, that only the values within a task (i.e. given the same tasks between the tools) are compared, since the granularity of the subtasks has not been matched.

Post Questionnaire

The ten usability heuristics suggested by Nielsen (1994) serve as criteria for the system's usability (see appendix B). The descriptions of the individual heuristics are direct quotations from the appendix in "Enhancing the Explanatory Power of Usability Heuristics". The post questionnaire contains brief descriptions of the heuristics and requests participants to rate the compliance with such heuristics on

1 We considered to weight *ml.general* higher, since participants whith experience in ML but not with python frameworks are disadvantaged. On the other hand, the positive weighting of the parameter would equally favour users of the framework.

a scale from 1 to 5. Furthermore, participants are asked to name the three largest advantages and disadvantages of MLE.

7.1.5 *Design*

With the experimental design we attempt to control confounding parameters with approaches suggested by Siegmund and Schumann (2015). We consider the following variables to be confounding parameters regarding the experiment: prior experience of participants (e.g. experienced participants could be biased towards familiar tools or practices), the tool order (e.g. the experience with the first could exhaust the participant or set the mood the second trial), and the task order (e.g. participants could be trained already after the first task).

We decided to control those variables by using them as independent variables. In section 7.1.1 it is as well defined as sub goal to evaluate the usability of the DSL according to participants with and without prior experience in ML. The variable *prg* was used to distinguish participants of both groups. Also, the task order as well as the tool order were used as independent variables by creating experiment iterations for each combination of those two parameters. The resulting task matrix is illustrated in table 7.2. Participants are assign to direction-columns to the left or the right of the matrix, making sure that each sequence of each group was run through only once. For instance, the participant with the id "ls8" first processes the brain task using python and then the clickbait task with the DSL. The value displayed next to participant ids is the *ml* variable.

$\longrightarrow$		Task Matrix		$\longleftarrow$	
yb1 (1,0)	ls8 (2,7)	$brain_{python}$	$clickbait_{dsl}$	gn3 (4,3)	vk5 (1,3)
ez1 (1,3)	ck6 (1,3)	$brain_{dsl}$	$clickbait_{python}$	ck6 (2,7)	ax3 (1,0)

Table 7.2: Task Matrix for the Experiment Design: task and tool order are defined by the row and the direction column. Participants are referred to by ids and *ml* value is displayed to each id.

7.1.6 *Procedure*

Introduction & Pre Questionnaire (approx. 15min)

The experiment took place at two different locations. Depending on what was easier for the participant to reach, it took place in a room at the University of Weimar or in a private apartment. In both places there was the same setup with the materials as described in section 7.1.3. Participants were being welcomed and introduced to the

topic of the thesis (including a brief example of ML) with a prepared introduction (see C.1.1). They were informed on the outline of the experiment and that participation was voluntary and could be canceled at any time. A consent was signed by the participant in order to agree to the collection and processing of data. The pre questionnaire (see 7.1) was filled and the *ml* score calculated in order to assign the participant to a group (with or without prior experience). Within a group only one member could process the respective experiment iteration. Otherwise, the assignment was done arbitrarily.

Task Execution (approx. 45min per task)

The task was assigned and the introduction to the first tool given. The supervisor performed the tutorial task on a laptop computer and the participant followed the procedure on the desktop computer. After the instruction, the participants were given time to read the complete task and question were answered. When the participant was ready, the first tasks was started. The supervisor annotated the procedure on a time-sheet notating error that occurred, task-durations, unexpected behavior, and comments of the participant. The think-aloud technique was used to get immediate feedback and to be able to listen to it afterwards if annotation have not been made in time. The supervisor did not respond to questions regarding solutions of tasks but answered comprehension questions during the task. After 30 minutes the task was disrupted. The participant was asked to take a break as long as required before starting the task execution all over again with the tutorial of the second tool.

Post Questionnaire (approx. 15min)

The participant was given the post questionnaire in order to assess the compliance of the tools with the 10 heuristics. Also, the three main advantages and disadvantages were surveyed. In a half structured interview participants were asked on their opinion towards the concept of sections, the nesting of statements, and the usage of projections. Additionally, participants were asked about unclear answers to the questionnaire and any further comments on the DSL. Finally, the supervisor thanked once more for the participation and handed over the present.

7.1.7 *Deviations during Experiment Execution*

During the execution of the experiment we observed following deviations from the standard procedure:

- During the first session, considerable problems with projectional editing were identified. The output below the sections was editable and instead of new sections, additional output fields were accidentally created. The problems could be solved with minor changes to the DSL and were removed before further experiments were executed.
- On the first Python task, even after half of the processing duration, the data were not properly loaded due to unknown parameters. Thus, the supervisor gave advise on how to solve the problem in order to avoid frustration. The same happened to everyone but one participant.
- A participant could not access parts of the MLE documentation. At this point, the supervisor tried to give hints to encourage exploration, instead of reproducing the contents of the documentation.

7.2 ANALYSIS

7.2.1 *Pre Questionnaire*

Table 7.3 shows the variables extracted from the pre questionnaire. When analyzing the data, we divided the participants in subgroups based on *ml*. We were surprised that the groups were not of equal size, since we expected all members of the Chair of Intelligent Software Systems being familiar with ML. Regarding *prog* the scores were relatively similar. One noteworthy exception is "yb1" with a very low score. Regarding *py*, it is interesting that there is one experienced python programmer in the group without *ml* experience.

$part_{id}$	ml	prog	py
ck6	4,33	2,05	5
ez1	2,67	2,05	4
ax3	2,67	2,05	5
ls8	1,33	1,90	2
vk5	1,33	2,62	1
gn3	1,33	2,62	4
ms2	1,00	2,05	1
yb1	1,00	1,17	1

Table 7.3: Prior experience.

7.2.2 *Experiment*

The data from the experiment is shown in table 7.4. Here each column corresponds to one participant. The variables *task*1 and *task*2 denote the respective task with *cb* abbreviating "clickbait" and *br* "brain data". The next group of rows (#*Suc*) reveals the number of successfully solved subtasks solved for the respective tool. Regardless of the tasks

order, the DSL always solved at least the same number of subtasks. This also applies when comparing the task between participants (it even applies $min(\#Suc_{dsl}) \geq max(\#Suc_{py})$). Accordingly, $\#Sear$ contains the number of web searches where $\#Sear_{py}$ is always at least the same number or bigger. The remaining rows contain the response times for the completion of a specific task. Since the maximum of successfully completed subtasks is three, we removed the other three rows per task from the table.

variable	ck6	ez1	ax3	ms2	ls8	yb1	vk5	gn3
*task*1	cb_{py}	cb_{py}	br_{dsl}	br_{dsl}	br_{py}	br_{py}	cb_{dsl}	cb_{dsl}
*task*2	br_{dsl}	br_{dsl}	cb_{py}	cb_{py}	cb_{dsl}	cb_{dsl}	br_{py}	br_{py}
$\#Suc_{py}$	1	1	1	1	0	1	0	1
$\#Suc_{dsl}$	2	2	1	3	1	1	2	2
$\#Sear_{py}$	7	3	7	5	5	4	8	6
$\#Sear_{dsl}$	3	3	4	2	2	3	4	2
t^1_{py}	12:00	20:40	13:20	14:50	-	22:50	-	19:00
t^2_{py}	-	-	-	-	-	-	-	-
t^3_{py}	-	-	-	-	-	-	-	-
t^1_{dsl}	11:20	14:00	27:00	8:10	14:10	12:10	4:40	4:50
t^2_{dsl}	18:10	26:30	-	13:30	-	-	22:20	19:40
t^3_{dsl}	-	-	-	26:50	-	-	-	-

Table 7.4: Performance parameters recorded by annotations during the experiment.

7.2.3 *Post Questionnaire*

In table 7.5 the aggregated assessments of single heuristics (codes A1 to A10, compare section 7.1.4) are shown: the column all_{py} contains the aggregation of the assessments of all users with respect to the control condition (python) and all_{dsl} the respective values for the assessment of the DSL. Remarkable are the values for A3 ("minimize memory load"), A4("consistency"), A5 ("visibility"), and A9 ("error prevention"), since here a clear difference in the assessment of the tools becomes visible (compare figure 7.1). With respect to A3 (), A5, and A9, MLE outperforms the control condition.

In contrast, A4 is the only heuristic in which the DSL is rated worse than the control condition. The difference is even larger when only participants with prior knowledge regarding ML are considered: Here, A4 is assessed with an arithmetic mean of 2.33 versus 3.50 in the group without prior knowledge.

In table 7.6, the characteristics mentioned most often as greatest advantage (above midrule) or disadvantage (below midrule) are listed.

Code	all_{py}	all_{dsl}	exp_{dsl}	$inexp_{dsl}$
A1	3.00	4.00	3.67	4.20
A2	2.86	3.17	3.00	3.25
A3	2.57	4.38	4.00	4.60
A4	4.13	3.00	2.33	3.50
A5	1.63	3.75	3.67	3.80
A6	3.29	3.63	3.67	3.60
A7	3.50	3.83	4.33	3.33
A8	2.75	2.86	2.33	3.25
A9	1.75	4.14	4.33	4.00
A10	2.63	3.38	3.00	3.60
All	2.81	3.61	3.43	3,71

Table 7.5: Arithmetic mean for the assessment of heuristics grouped by participants (all, experienced, inexperienced) and tools.

The remaining characteristics not included in the list either were empty (i.e. the participant only wrote one or two advantages/disadvantages) or related to a very specific problem more related to the experiment than the DSL (e.g. missing documentation files).

count	description
6	Simple, unified, task-specific experience (not a collection of frameworks)
4	Clean interface, clear visual separation through sections
3	Code Suggestions enable fast results
6	Bugs, system stability
5	Inconsistency of names, syntax (shuffle)
3	Projectional editing too constrained, slow down

Table 7.6: Clusters of most liked/disliked aspects of MLE. The count is the number of participants who mentioned a

7.3 DISCUSSION

In this section we discuss the results presented in the analysis. Additionally, we include observations from the usability experiment in the discussion that we could not represent in numbers.

7.3.1 *Efficiency*

Participants could at most complete one and on average only 0.75 tasks when using python. In contrast, they completed at least one

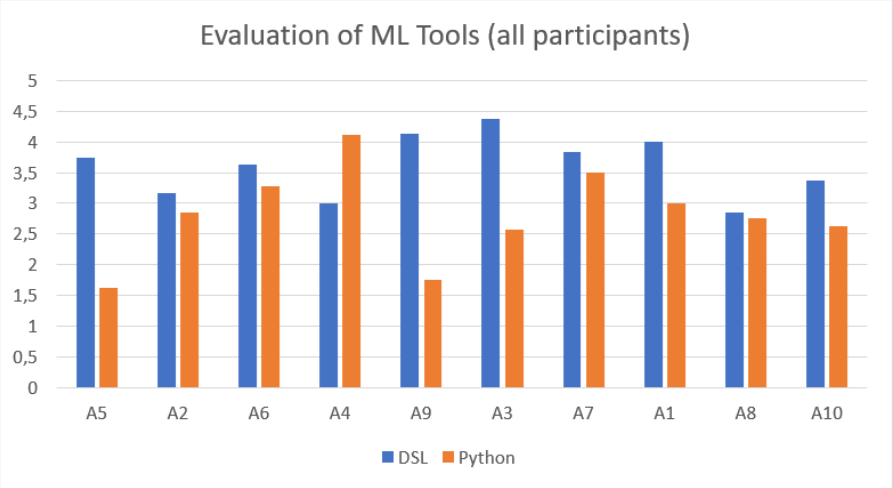

Figure 7.1: The usability assessment of all participants (arithmetic mean) for both tools, DSL and Python (Control Condition). The order of heuristics corresponds to the order on the questionnaire.

task with MLE on average 1.875. From that numbers, it is apparent that MLE can improve the efficiency. However, the context needs to be considered: the main problem on python was the process of reading in data. From the tutorial, participants suggested that the `.read_*()` function (i.e. *read_json*, `read_csv`, etc.) would simply accept all data as long as the correct file-extension is selected. Nevertheless, the files in which data was stored in the task required addition parameters to the function.

In contrast, MLE currently does not accept parameter on the file picker. Both datasets where used beforehand to test the DSL; thus, the file picker were configured to accept exactly those file-configurations (i.e. one parameter per task). The file picker in MLE is considered to automatically select correct parameters in the future.[2] However, for this experiment we did not expect the loading of files to have such impact on the tasks.

The efficiency of DSL must be considered in relation to this context. As described in section 7.1, the assessment of efficiency was a secondary objective. However, we believe that the trouble-free reading of data in the DSL has played a role in the assessment – especially in direct comparison with the control condition.

2 For instance, on csv files the server could check common separators on the first two non-header lines. It would then select the first separator for which the number of features is greater than one and equal for both lines.

7.3.2 *Usability*

Table 7.5 reveals that participants with prior knowledge in ML assess the usability slightly stricter. We consider only the deviations in A4 (Consistency) noteworthy, with a value of 2.33 from experienced versus 3.5 from inexperienced participants. For all other heuristics, we use the assessments aggregated from all participants in order to discuss the experiment results. As stated in the analysis, the heuristics relevant for the discussion are A3 ("minimize memory load"), A4("consistency"), A5 ("visibility"), and A9 ("error prevention").

Consistency

For this heuristic, we think that experienced users were better qualified to notice inconsistencies such as confusion between standardization and normalization and or the term featureset (which was expected to be only the header of the data)[3]. However, table 7.6 shows that inconsistencies were mentioned five times as disliked characteristic of the DSL; thus, at least two participants without prior knowledge discovered similar problems.

We believe, that the introduction to the statement "sort instances at random" explains a part of this feedback. The supervisor stated at least[4] in two tutorials that the operation might be confusing and inconsistent since a name such as "shuffle" might be preferred.

Minimize Memory Load

The DSL was assessed with an average value of 4.38 out of 5. In contrast, the control condition was rated 2.57. We believe, that the positive value for MLE is caused to some extend on the code suggestion of projectional editing, in general. This is also indicated by the advantage mentioned third most which is summarized as "Code Suggestions enable fast results". However, we observed immediate feedback to up-to-date features in the featureset (compare 6.3.1) by one user who was excited that features were addressable as a single unit after grouping: "Sind jetzt alle weg? Ah, cool!" ("Are they all gone? Ah, cool!"). Although only four participants reached tasks where items could be targeted in expressions, everyone experienced the use of them during the tutorial.

3 In fact, only the header is loaded into MLE. However, one participant did not expect the editing of instances in the featureset.

4 Since we started the voice recording only at the beginning of the task, we cannot check the exact number of tutorial walkthoughs effected by this comment.

Error Prevention

Due to the number of errors and glitches that occurred during the experiment, we were surprised that A9 ("error prevention") was still rated with an average of 4.14 out of 5. In comparison, the control condition received only 1.75. We believe that the trouble caused by the load function is responsible for the bad rating of python. Moreover, the python error messages did not help to find the responsible line in code. We suspect that the cluttered presentation of errors prevented users from reading them carefully. Further, we believe that the early state of MLE was considered during the assessment of A9.

7.3.3 *Study Results*

The performed experiment revealed that MLE performs well in comparison with the control condition: The DSL achieves a higher task completion rate than python, even though the context needs to be considered which may questions the reliability of this result. However, regarding the usability heuristics used to rate the system, MLE outperforms the control condition. The experiment proves that the objectives of the usability driven language design are perceived by the participants of the study.

On the contrary, inconsistencies in terminology and the nesting of statements were made apparent. The other usability heuristics not directly addressed in the language design are still rated at least as good as the control condition. We are confident that further improvements can be made in these aspects by optimizing MLE's usage of the tooling provided by MPS.

8

CONCLUSION

The objective of this thesis was the development of a Domain-Specific Language (DSL), which opens the Machine-Learning (ML) for domains distant to computer science. With Machine Learning Evolves (MLE) we strive at developing a language that places itself orthogonally to existing frameworks and libraries. It is intended to provide experts from other domains with a well-structured environment that makes the sophisticated ML tools accessible without overwhelming the users with their complexity.

In MPS we have found a platform on which we can achieve this goal. It provides a comprehensive and extensible tool set and forms the basis for modular language development. Most importantly, however, it is the only development environment to our knowledge that enables projectional editing. This main characteristic of MPS prevents users from writing incorrect code and promotes exploration due to the context-aware code-suggestions.

Our main requirements for MLE are the adaptability to domain notations and vocabulary, the scalability of the language to the required functional scope and an efficient extensibility to handle the fast development in the field of ML. We explained in detail how an interactive component model can be used to specify individualized dialects from MLE. Moreover, we demonstrated how classes can be efficiently expanded and integrated into the DSL.

Regarding the usability of the DSL it was our goal to give the user an overview of his possibilities and of the system status at any time. We pursued this goal by implementing the visibility and feedback paradigms. Two resulting distinctive features of MLE are the referencability of items, which, for example, enables the use of individual features as variables in expressions, and prompt feedback inside the IDE which takes the current object state into account.

By means of a qualitative usability study, we were able to prove the benefit of the language characteristics mentioned earlier. Participants of the study confirmed the minimization of the user's memory load, the visibility of the program state, as well as effective error prevention. However, several weaknesses of the DSL also became apparent: Often there were problems with the use of projectional editing, which are not solely due to the unfamiliar handling. The inconsistency of terminology in MLE was justly criticized in particular by participants with

prior experience with ML. Finally, many minor errors were noticed which, combined, clouded the overall impression of the language.

With only one exception, the study was attended by participants from the field of computer science. Half of the participants stated that they enjoyed the experience but would hesitate to use it in their day-to-day work for reasons such as fear of missing functions, constraints of projectional editing, or being forced to use this specific IDE. Since MLE is intended to be tailored for non-programmers, the question arises to what extent the results of the study can be applied to them.

The study has encouraged us to continue on the path with MLE and we were able to derive the next steps from it: First, MLE must be brought to a more stable base. On the one hand, this means a more consistent implementation of the variability mechanisms in the concepts of DSL that were previously developed. On the other hand, comprehensive testing of the language concepts is necessary in order to guarantee reliability even with increasing complexity. Second, with regard to the expandability of DSL, the server side must also offer variability mechanisms. In general, the interaction between DSL and server requires more attention, since currently neither waiting times are indicated nor error messages are transmitted. Third, design decisions currently made by a small team must be measured against the needs of potential users. We believe that an exchange with both, ML experts and DSL users is essential for a meaningful requirements analysis.

With these points being followed, we believe that MLE can open machine learning to a broad array of domain.

Part IV

APPENDIX

A

SOURCE-CODE OF CLASS: TREEBUILDER

```
1 public class TreeBuilder {
```

TreeBuilder.build()

```
3   public static node<AbstractComponentNode> build() {
4     nlist<ComponentDescription> descriptions =
5       Components.getAll();
6     nlist<AbstractComponentNode> vertices =
7       createVerticesFromDescriptions(descriptions);
8     node<AbstractComponentNode> root =
9       findEquivalentVertex(
10        findRootInDescriptions(descriptions), vertices);
11    connectVertices(root, vertices, descriptions);
12    restoreCrossTreeRelations(root, descriptions);
13
14    return root;
15  }
```

TreeBuilder.connectVertices()

```
17  private static void connectVertices(
18    node<AbstractComponentNode> treeRoot,
19    nlist<AbstractComponentNode> vertices,
20    nlist<ComponentDescription> descriptions)
21  {
22    queue<node<AbstractComponentNode>> parentList =
23      new linkedlist<node<AbstractComponentNode>>;
24    parentList.addLast(treeRoot);
25
26    while (!parentList.isEmpty) {
27      node<AbstractComponentNode> parent =
28        parentList.removeFirst;
29      foreach child in
30        findChildrenInDescriptions(descriptions, parent)
31      {
32        node<AbstractComponentNode> childNode =
33          findEquivalentVertex(child, vertices);
34        parent.childNodes.add(childNode);
35        parentList.addLast(childNode);
36      }
37    }
38  }
```

TreeBuilder.restoreCrossTreeRelations()

```
39   private static void restoreCrossTreeRelations(
40     node<AbstractComponentNode> treeRoot,
41     nlist<ComponentDescription> descriptions)
42   {
43     foreach cd in descriptions {
44       node<AbstractComponentNode> vertex =
45         findCorrespondingVertex(cd.path, treeRoot);
46
47       foreach mandatory in cd.mandatory {
48         vertex.mandatory.add(
49           restoreReference(mandatory.ref.path, treeRoot));
50         vertex.mandatory.last.ref.isMandatoryChild = true;
51       }
52       foreach required in cd.required {
53         vertex.required.add(
54           restoreReference(required.ref.path, treeRoot));
55       }
56       foreach excluded in cd.excluded {
57         vertex.excluded.add(
58           restoreReference(excluded.ref.path, treeRoot));
59         vertex.excluded.last.ref.excluded.add(
60           restoreReference(vertex.path, treeRoot));
61       }
62     }
63   }
```

TreeBuilder.restoreReference()

```
65   private static node<ComponentNodeRef> restoreReference(
66     string targetPath,
67     node<AbstractComponentNode> root)
68   {
69     node<ComponentNodeRef> ref =
70       new node<ComponentNodeRef>();
71     ref.ref = findCorrespondingVertex(targetPath, root);
72     return ref;
73   }
```

TreeBuilder.createVerticesFromDescriptions()

```
75   private static nlist<AbstractComponentNode>
76     createVerticesFromDescriptions(
77       nlist<ComponentDescription> descriptions)
78   {
79     nlist<AbstractComponentNode> result =
80       new nlist<AbstractComponentNode>;
81     foreach fd in descriptions {
82       result.add(createVertex(fd));
83     }
84     return result;
85   }
```

TreeBuilder.createVertex()

```
87    private static node<AbstractComponentNode>
88      createVertex(node<ComponentDescription> node)
89    {
90      node<AbstractComponentNode> graphNode = null;
91
92      if (node.childrenType.equals(
93        enum member value/children_type : independent/))
94      {
95        graphNode = new node<ComponentNodeIndep>();
96      } else if (node.childrenType.equals(
97        enum member value/children_type : 1-of-m/))
98      {
99        graphNode = new node<ComponentNode1M>();
100     } else if (node.childrenType.equals(
101       enum member value/children_type : n-of-m/))
102     {
103       graphNode = new node<ComponentNodeNM>();
104     }
105     graphNode.initFromArtifactDescription(node);
106
107     return graphNode.copy;
108   }
```

TreeBuilder.findEquivalentVertex()

```
110   private static node<AbstractComponentNode>
111     findEquivalentVertex(
112       node<ComponentDescription> target,
113       nlist<AbstractComponentNode> vertices)
114   {
115     return vertices.findFirst({
116       ~it => it.path.equals(target.path); });
117   }
```

TreeBuilder.findRootInDescriptions()

```
119   private static node<ComponentDescription>
120     findRootInDescriptions(
121       sequence<node<ComponentDescription>> nodes)
122   {
123     return nodes.findFirst({
124       ~it => it/.getModel().getModule() == module/core/ })
125   }
```

TreeBuilder.findChildrenInDescriptions()

```
127   private static sequence<node<ComponentDescription>>
128     findChildrenInDescriptions(
129       nlist<ComponentDescription> nodes,
130       node<AbstractComponentNode> parent)
131   {
132     return nodes.where({
133         ~it => it.parent.isNotNull
134                && it.parent.path.equals(parent.path); });
135   }
```

TreeBuilder.findCorrespondingVertex()

```
137   private static node<AbstractComponentNode>
138     findCorrespondingVertex(
139       string targetPath,
140       node<AbstractComponentNode> root)
141   {
142     string[] aliases = targetPath.split("[.]");
143
144     node<AbstractComponentNode> curNode = root;
145     for (int i = 1; i < aliases.length; ++i) {
146       curNode = getChildWithAlias(
147         aliases[i],
148         curNode.childNodes);
149     }
150
151     return curNode;
152   }
```

TreeBuilder.getChildWithAlias()

```
154   private static node<AbstractComponentNode>
155     getChildWithAlias(
156       string alias,
157       nlist<AbstractComponentNode> children)
158   {
159     return children.findFirst({
160       ~it => it.shortname :eq: alias; });
161   }
```

End of Class: TreeBuilder

```
163 }
```

B

NIELSEN'S USABILITY HEURISTICS

The ten usability heuristics suggested by Nielsen (1994) serve as criteria for the system's usability. The descriptions of the individual heuristics are direct quotations from the appendix in "Enhancing the Explanatory Power of Usability Heuristics":

Code	Usability Heuristic
A1	Simple dialogue: Dialogues should not contain information which is relevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility. All information should appear in a natural logical order.
A2	Natural language: The dialogue should be expressed clearly in words, phrases and concepts familiar to the user, rather than in system-oriented terms.
A3	Minimize memory load: The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.
A4	Consistency: Users should not have to wonder whether different words, situations, or actions mean the same thing.
A5	Feedback: The system should not always keep users informed about what is going on, through appropriate feedback within reasonable time.
A6	Clearly marked exists: Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue.
A7	Shortcuts: Accelerators—unseen by the novice user—may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users.
A8	Good error messages: They should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.
A9	Error Prevention: Even better than good error messages is a careful design that prevents a problem from occurring in the first place.
A10	Help and documentation: Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, be focused on the user's task, list concrete steps to be carried out, and not be too large

C

EXPERIMENT MATERIAL

C.1 PROCEDURE

C.1.1 *Welcome*

Original

Hallo! Schön, dass Du da bist und an der Studie zu meiner Masterarbeit teilnehmen willst. In meiner Arbeit habe ich mich mit Maschinellem Lernen beschäftigt. Genauer habe ich eine DSL, das ist eine vereinfachte Programmiersprache, für Maschinelles Lernen entworfen, die nun getestet werden soll.

Zunächst einmal: Was ist Maschinelles Lernen? Es ist natürlich nicht so wie in SciFi-Filmen, dass Maschinen ein künstliches Gehirn programmiert bekommen, dass es ihnen ermöglicht selbstständig zu denken und zu lernen. Maschinelles Lernen funktioniert sehr viel einfacher auf Basis von Beispieldaten. Zum Beispiel könnten eine Menge von Texten, für welche bekannt ist, ob es sich um Lyrik handelt oder nicht, dazu dienen einer Maschine beizubringen, lyrische Texte von nicht-lyrischen zu erkennen. Das wäre dann Klassifikation, also die Einordnung, ob ein Objekt zu der gewünschten Klasse gehört oder nicht. Diese Art von Aufgabe soll heute auch bearbeitet werden.

Wichtig: Ziel der Studie ist es, die Programmiersprache zu evaluieren, nicht dich als Probanden – es gibt also kein gutes oder schlechtes Abschneiden. Es sollen Fehler und Schwächen deutlich werden und hoffentlich auch Stärken und Vorzüge. Insbesondere bin ich auch an deiner Meinung interessiert.

English Translation

Hello! Nice that you are here and want to participate in the study for my master thesis. In my work I have dealt with machine learning. More precisely, I designed a DSL, which is a simplified programming language, for machine learning, which will now be tested.

First of all: What is machine learning? It is of course not like in SciFi movies that machines are programmed with an artificial brain that allows them to think and learn independently. Machine learning works much easier on the basis of example data. For example, a lot of texts for which it is known whether they are poetry or not could serve to teach a machine to recognize poetry from non poetry. That would then be classification, i.e. whether an object belongs to the desired class or not. This kind of task should also be processed today.

Important: The aim of the study is to evaluate the programming language, not you as the participant - so there is no good or bad result. Flaws and weaker spots should become apparent and hopefully also strengths and advantages. I am especially interested in your opinion.

C.1.2 *Tutorials Tasks*

Machine Learning Evolves

Program sections:

```
section loadData uses vars << ... >> and provides vars iris
code
  create featureset iris from file[solution_root/iris.json]
  print iris
  provide iris
```

```
featuretable 'iris' loaded with 5 dimensions and 150 rows
iris[5 dimensions, 150 rows]:
```

name	type	#empty
petalLength	float	0
petalWidth	float	0
sepalLength	float	0
sepalWidth	float	0
species	string	0

```
section processData uses vars loadData.iris  and provides vars iris
code
  edit iris
    [preprocess feature species: encode labels using single column
    group features as features: petalLength .. sepalWidth
    preprocess feature features: standardize <group by>
    set truth species]
  provide iris
```

```
Feature 'species' of type string was label encoded and 3 labels were found:
  setosa (0), versicolor (1), virginica (2)
Values in feature 'features' were standardized.
```

```
section applyML uses vars processData.iris  and provides vars << ... >>
code
  create model model as new model
  edit iris [sort instances at random]
  split iris [instances]:
    part iris_1 takes 90 %
    part iris_2 takes 10 %
  edit model
    [find estimator for classification on iris
    train model on iris_1
    test model on iris_2]
```

```
The featureset 'iris' (150 items) is split into:
  Part 'iris_1' with 135 items
  Part 'iris_2' with 15 items
Three best estimators found for classification on iris:
  1. KNeighborsClassifier with f-score 0.9599673202614379 (selected)
```

Python with Pandas and Scikit-Learn

```
# Load Data
import pandas as pd
data = pd.read_json("iris.json")

print(data.head())


# Split Into Values and Truth-Class
y = data.species

X = data[["petalLength", "petalWidth", "sepalLength", "
    sepalWidth"]]
X = data.iloc[:, :4]

print(X.head())

# Standardize Values
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()

X_std = sc.fit_transform(X)
X = pd.DataFrame(columns=X.columns.values, data=X_std)

print(X[:5])

# Split Into Test and Train
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.3, random_state=0)

print('There are {} samples in the training set and {}
    samples in the test set'.format(
    X_train.shape[0],
    X_test.shape[0]))

# Create Classifier and Train On Data
from sklearn.svm import SVC

svm = SVC()
svm.fit(X_train, y_train)

print('Accuracy of the svm classifier: ' + str(svm.score(
    X_test, y_test)))
```

C.2 TASKS

C.2.1 *Clickbait (Original)*

Intro

Maschinen-Lernen soll dazu genutzt werden zu erkennen, ob es sich bei Tweets (Twitter-Posts) um sogenanntes Clickbait handelt oder nicht. Wikipedia beschreibt Clickbait wie folgt:

Ein Clickbait besteht in der Regel aus einer reißerischen Überschrift, die eine sogenannte Neugierlücke (englisch curiosity gap) entstehen lässt. Sie teilt dem Leser gerade genügend Informationen mit, um ihn neugierig zu machen, aber nicht ausreichend, um diese Neugier auch zu befriedigen, ähnlich einem Cliffhanger.

Um einen Datensatz zu erstellen, mit dessen Hilfe ein Klassifizierer trainiert werden kann, wurden Tweets im Crowdsourcing-Verfahren annotiert. Der gleiche Beitrag wurde dabei von einigen verschiedenen Annotatoren bewertet und aus den Einzelentscheidungen ein Clickbait-Score berechnet. Für die Klassifizierung wurde zusätzlich ein Klassenlabel („truthClass“) erstellt, das für jeden Tweet eindeutig die Einstufung („clickbait“ oder „no-clickbait“) angibt.

Aus den gegebenen Daten müssen für ML-Verfahren verwertbare (d.h. numerische) Eigenschaften eines Tweets abgeleitet werden, aus denen gefolgert werden kann, ob es sich um Clickbait handelt oder nicht. Beispielsweise könnte man vermuten, dass Clickbait-Tweets besonders häufig auf visuelle Reize setzen und deshalb meistens Bilder enthalten sind. Ein Boolean-Feature „pictureIsPresent“ würde dafür sorgen, dass diese Eigenschaft beim Trainieren des Klassifikators berücksichtigt wird.

Task 1

Der Clickbait-Datensatz liegt wie oben beschrieben aufgeteilt in zwei Dateien vor. Als gemeinsames Feature zum Verknüpfen der Daten dient die Tweet-Id, die in beiden Datensätzen enthalten ist. Es sollen die Daten geladen und nicht benötigte Features entfernt werden. Benötigt werden postMedia, postText und id aus tweets.jsonl sowie id und truthClass aus truth.jsonl.

Task 2

N-Gramme können bspw. Wortgruppen der Länge n sein (aber auch Buchstabenfolgen, Bitfolgen o.Ä.). Tweets können in Form von N-Grammen repräsentiert werden, indem für die im Datensatz vorhandenen N-Gramme gespeichert wird, wie oft sie hier vorkommen.

Es wird angenommen, dass bestimmte (aber unbekannte) Wortgruppen in Clickbait-Tweets häufiger und in Nicht-Clickbait-Tweets selten vorkommen. Es soll deshalb die Häufigkeit (abs.) von 3-Grammen aus postText extrahiert werden.

Task 3

Der Flesch-Kincaid Score ist ein Maß für die Komplexität eines Textes. Der Minimalwert beträgt -3.4, eine obere Grenze gibt es nicht. Je höher der Wert, desto schwieriger ist der Text zu verstehen. Zur Berechnung wird folgende Formel verwendet:

<Formel>

Es wird angenommen, dass Clickbait-Posts in sehr einfacher Sprache verfasst werden. Es soll deshalb der Flesch-Kincaid Score aus dem postText extrahiert werden. Für Posts, die keinen Text enthalten, soll der Minimalwert verwendet werden.

Task 4

Die Anzahl von sogenannten Mentions (Verweise auf andere Twitter-Accounts) oder Hashtags (Twitter-Schlagwörter) könnte bei Clickbait-Tweets im Allgemeinen höher sein, um damit die Reichweite zu erhöhen. Etwas vereinfacht können sie durch die Zeichen „@“ bzw. „#“ ausgemacht werden. Es soll jeweils die Anzahl dieser Zeichen im postText als Feature extrahiert werden.

Task 5

Clickbait zeigt sich oft in Gestalt von Empfehlungslisten wie zum Beispiel: „5 Dinge, die du niemals essen solltest... Nr. 3 wird dich überraschen!“ Dies soll durch ein Feature „beginsWithDigit“ berücksichtigt werden, das einen entsprechenden Wahrheitswert enthält.

Auch wird in Clickbait häufig auf visuelle Reize gesetzt, weshalb ein weiteres Boolean-Feature „mediaAttached“ extrahiert werden soll, das angibt, ob ein Bild, Video, o.Ä im Post enthalten ist.

Task 6

Als letztes soll ein passender Klassifikator ausgewählt werden, dessen Aufgabe es ist, Clickbait von Nicht-Clickbait-Posts zu unterscheiden. Zum Trainieren sollen 75% der Daten als Input für ein 5-fold Cross-Validation-Verfahren dienen. Die anderen 25% der Daten sollen zum Testen des Klassifikators genutzt werden.

C.2.2 *Clickbait (English Translation)*

Intro

Machine learning should be used to detect whether tweets (Twitter posts) are clickbait or not. Wikipedia describes Clickbait as follows:

A clickbait usually consists of a lurid headline that creates a so-called curiosity gap. It gives the reader just enough information to make him curious, but not enough to satisfy this curiosity, similar to a cliffhanger.

In order to create a data set that can be used to train a classifier, tweets were annotated using the crowdsourcing method. The same contribution was evaluated by several different annotators and a clickbait score was calculated from the individual decisions. In addition, a class label ("truthClass") was created for the classification, which clearly indicates the classification ("clickbait" or "no-clickbait") for each tweet.

From the given data, usable (i.e. numerical) properties of a tweet must be derived for ML procedures, from which it can be concluded whether it is clickbait or not. For example, one could assume that Clickbait tweets are particularly often based on visual stimuli and therefore usually contain images. A Boolean feature "pictureIsPresent" would ensure that this feature is taken into account when training the classifier.

Task 1

As described above, the Clickbait dataset is split into two files. The common feature for linking the data is the Tweet-Id, which is contained in both data sets. The data should be loaded and unneeded features should be removed. Required are postMedia, postText and id from tweets.jsonl and id and truthClass from truth.jsonl.

Task 2

N-grams can be, for example, word groups of length n (but also letter sequences, bit sequences, etc.). Tweets can be represented in the form of N-grams by storing how often they occur in the data set for the N-grams present.

It is assumed that certain (but unknown) word groups occur more frequently in clickbait tweets and rarely in non-clickbait tweets. Therefore, the frequency (abs.) of 3-grams shall be extracted from postText.

Task 3

The Flesch-Kincaid Score is a measure of the complexity of a text. The minimum value is -3.4, there is no upper limit. The higher the value, the more difficult it is to understand the text. The following formula is used for the calculation:

<Formula>

It is assumed that Clickbait posts are written in very simple language. Therefore the Flesch-Kincaid score should be extracted from the postText. For posts that do not contain any text, the minimum value should be used.

Task 4

The number of so-called mentions (references to other Twitter accounts) or hashtags (Twitter keywords) could generally be higher for Clickbait tweets in order to increase their reach. In a somewhat simplified way, they can be identified by the characters "@" or "#". The number of these characters in the postText should be extracted as a feature.

Task 5

Clickbait often comes in the form of recommendation lists such as "5 things you should never eat... No. 3 will surprise you! This should be taken into account by a feature "beginsWithDigit" which contains a corresponding truth value.

Clickbait also often focuses on visual stimuli, which is why another Boolean feature "mediaAttached" should be extracted to indicate whether an image, video, etc. is included in the post.

Task 6

Finally, a suitable classifier should be selected, whose task is to distinguish Clickbait from non-Clickbait posts. To train, 75% of the data should serve as input for a 5-fold cross-validation procedure. The other 25% of the data should be used to test the classifier.

C.2.3 *Brain Data (Original)*

Intro

Maschinen-Lernen soll dazu genutzt werden, anhand von Hirn-Aktivität-Scans zu klassifizieren, ob die Versuchsperson sich mit einer bestimmten Aufgabe befasste oder nicht. Die Aufgabe im Versuchsaufbau war, den Sinn eines vorgelegten Programmcodes zu verstehen. Um den Einfluss der visuellen Verarbeitung von Programmcode heraus rechnen zu können, sollten die Versuchspersonen Syntax-Fehler in anderen Codesamples suchen. Zwischen den verschiedenen Aufgaben gab es zudem Ruhephasen, für die es ebenfalls Scans gibt, die als Ausgangswert genutzt werden können.

Task 1

Die Daten sollen geladen und bereinigt werden: Die ersten drei Trials („Rest“, „Comprehension“, „Rest“) dienten als Warm-Up und die dazugehörigen Messungen sollen entfernt werden. Gleiches gilt für den letzten Task („Rest“).

Task 2

Die Messdaten sollen normalisiert und statistische Ausreißer bereinigt werden. Dabei ist es wichtig, die Daten der einzelnen Probanden voneinander separiert zu behandeln.

Task 3

Die Messdaten sollen für jeden Probanden und Task zusammengefasst werden, sodass es nur einen Wert pro Task und Hirnareal gibt. Für die Zusammenfassung soll das arithmetische Mittel verwendet werden.

Task 4

Der Task-Typ soll als Wahrheitsklasse für das Maschinen-Lernen verwendet werden. Für den Klassifizierungstask darf es nur zwei Klassen geben, weshalb Werte „S“ (Syntax) und „R“ (Rest) zusammengefasst werden müssen (z.B. „notC“). Außerdem soll der Task-Typ als numerisches Feature enkodiert werden (Label Encoding).

Task 5

Es soll ein passender Klassifikator ausgewählt werden, dessen Aufgabe es ist, Comprehension-Tasks von anderen Tasks zu unterscheiden. Benötigt werden dafür nur die Messwerte und der Task-Type als Wahrheitsklasse. Zum Trainieren sollen 75% der Daten als Input für ein 5-fold Cross-Validation-Verfahren dienen. Die anderen 25% der Daten sollen zum Testen des Klassifikators genutzt werden.

Task 6

Um auszuschließen, dass der Lernerfolg bei der Klassifizierung auf die Algorithmen und nicht den Task-Typ zurückzuführen ist, sollen die Snippets neu sortiert und als Wahrheitsklasse verwendet werden. Die einzelnen Task-Labels (die Namen der Algorithmen) sollen in zwei zufällige, gleich große Gruppen unterteilt werden.

Das Zwischenergebnis könnte (jedoch ohne zufälliger Zuteilung der Gruppen) wie folgt aussehen:

<Illustration>

Nun soll die erste der beiden Gruppen von der zweiten Gruppe und den "R"-tasks (in der Tabelle die Zellen unter "group_2" ohne Wert) unterschieden werden. Es müssen entsprechend Aufgaben vier und fünf mit "group_2" als Wahrheitsklasse wiederholt werden.

C.2.4 *Brain Data (English Translation)*

Intro

Machine learning will be used to use brain activity scans to classify whether the subject was involved in a particular task or not. The task in the experimental setup was to understand the meaning of a presented program code. In order to be able to calculate the influence of the visual processing of program code, the test subjects should search for syntax errors in other code samples. There were also rest periods between the various tasks, for which there are also scans that can be used as initial values.

Task 1

The data is to be loaded and cleaned up: The first three trials ("Rest", "Comprehension", "Rest") served as warm-up and the corresponding measurements are to be removed. The same applies to the last task ("Rest").

Task 2

The measurement data should be normalized and statistical outliers corrected. It is important to treat the data of the individual test persons separately.

Task 3

The measurement data should be summarized for each respondent and task, so that there is only one value per task and brain area. The arithmetic mean should be used for the summary.

Task 4

The task type is to be used as a truth class for machine learning. There may only be two classes for the classification task, which is why values "S" (syntax) and "R" (rest) must be combined (e.g. "notC"). In addition, the task type is to be encoded as a numeric feature (Label Encoding).

Task 5

A suitable classifier is to be selected whose task is to distinguish comprehension tasks from other tasks. Only the measured values and the task type as truth class are needed. For training, 75% of the data should serve as input for a 5-fold cross-validation procedure. The other 25% of the data should be used for testing the classifier.

Task 6

To exclude that the learning success in the classification is due to the algorithms and not to the task type, the snippets should be re-sorted and used as a truth class. The individual task labels (the names of the algorithms) are to be divided into two random groups of equal size.

The intermediate result could look like this (but without random assignment of the groups):

<Illustration>

Now the first of the two groups shall be distinguished from the second group and the "R"-tasks (in the table the cells under "group_2" without value). Accordingly, tasks four and five must be repeated with "group_2" as truth class.

Abadi, Martín et al. (2016). "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems." In: *CoRR* abs/1603.04467.

Apel, Sven, Don Batory, Christian Kästner, and Gunter Saake (2016). *Feature-oriented software product lines*. Springer.

Behringer, B., J. Palz, and T. Berger (2017). "PEoPL: Projectional Editing of Product Lines." In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 563–574.

C. Kang, Kyo, Sholom Cohen, James A. Hess, William Novak, and A Spencer Peterson (1990). "Feature-Oriented Domain Analysis (FODA) feasibility study." In:

Dmitriev, Sergey E (2004). "Language Oriented Programming The Next Programming Paradigm." In:

Drey, Zoé, Cyril Faucher, Franck Fleurey, Vincent Mahé, and Didier Vojtisek (2009). "Kermeta language." In: *Reference Manual*.

Efftinge, Sven and Markus Völter (2006). "oAW xText: A framework for textual DSLs." In: *Workshop on Modeling Symposium at Eclipse Summit*. Vol. 32, p. 118.

Feigenspan, Janet, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg (2012). "Measuring programming experience." In: *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*. IEEE, pp. 73–82.

Fisher, Ronald A (1936). "The use of multiple measurements in taxonomic problems." In: *Annals of eugenics* 7.2, pp. 179–188.

K. Sujeeth, Arvind, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun (2011). "OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning." In: pp. 609–616.

Kang, K. C., J. Lee, and P. Donohoe (2002). "Feature-Oriented Product Line Engineering." In: *IEEE Software* 19, pp. 58–65.

Low, Yucheng, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein (2014). "GraphLab: A New Framework For Parallel Machine Learning." In: *CoRR* abs/1408.2041.

Makarkin, Aleksey (2016). *Open API - accessing models from code*. Retrieved January 20, 2018. URL: https://confluence.jetbrains.com/display/MPSD20182/Open+API+-+accessing+models+from+code.

Nielsen, Jakob (1994). "Enhancing the Explanatory Power of Usability Heuristics." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '94. Boston, Massachusetts, USA: ACM, pp. 152–158. ISBN: 0-89791-650-6.

Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. (2011). "Scikit-learn: Machine learning in Python." In: *Journal of machine learning research* 12.Oct, pp. 2825–2830.

Portugal, Ivens, Paulo Alencar, and Donald Cowan (2016). "A survey on domain-specific languages for machine learning in big data." In:

Potthast, Martin, Tim Gollub, Matthias Hagen, and Benno Stein (Dec. 2018). "The Clickbait Challenge 2017: Towards a Regression Model for Clickbait Strength." In: *Clickbait Challenge 2017*. URL: https://arxiv.org/abs/1812.10847.

Siegmund, Janet and Jana Schumann (2015). "Confounding parameters on program comprehension: a literature survey." In: *Empirical Software Engineering* 20.4, pp. 1159–1192.

Voelter, Markus (2018). "The Design, Evolution, and Use of KernelF - An Extensible and Embeddable Functional Language." In: *Theory and Practice of Model Transformation - 11th International Conference, ICMT 2018, Held as Part of STAF 2018, Toulouse, France, June 25-26, 2018, Proceedings*, pp. 3–55.

Voelter, Markus and Eelco Visser (2011). "Product line engineering using domain-specific languages." In: *Software Product Line Conference (SPLC), 2011 15th International*. IEEE, pp. 70–79.

Völter, Markus (2014). "Generic Tools, Specific Languages." PhD thesis.

Völter, Markus, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth (2013). *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org. ISBN: 978-1-4812-1858-0. URL: http://www.dslbook.org.

Ward, Martin (1994). "Language Oriented Programming." In: *Software-Concepts and Tools* 15, pp. 147–161.

Weimer, Markus, Tyson Condie, and Raghu Ramakrishnan (2011). "Machine learning in ScalOps, a higher order cloud computing language." In: *NIPS 2011 Workshop on parallel and large-scale machine learning (BigLearn)*.

DECLARATION

I hereby confirm that I have written this paper independently and have not used any aids other than those indicated. The parts of the work which are taken from the wording or the meaning of other works (including internet sources) have been marked with the source.

Weimar, January, 2018

Aaron Solbach