University of Bergen
Department of Informatics

Development of a system for example-driven software language engineering

Author: Simen André Lien

Supervisors: Mikhail Barash, Anya Helene Bagge

UNIVERSITETET I BERGEN

Det matematisk-naturvitenskapelige fakultet

June, 2022

Abstract

Domain-specific languages (DSL) are programming languages designed with a particular application domain in mind. DSLs are used in financial, automative, engineering, biological, and medical industries, among others. However, building domain-specific languages is both time-consuming and requires knowledgeable developers. In order to use a DSL in a reasonable manner, one needs an Integrated Development Environment (IDE) that supports it. *Language workbenches* are tools to facilitate the development of DSLs and accompanying IDEs that support them. In this thesis, we present a web-based tool that allows for defining DSLs in an *example-driven* way. There, for each language construct, a user can specify its syntax and validation rules by *annotating* sample code snippets written in the language being designed. From such a language specification, we generate a grammar for a text-based language workbench Langium, which further outputs a VSCode-based IDE for the language. With the prototype implementation of our tool, we have demonstrated the viability of an example-driven approach to defining software languages.

Acknowledgements

I would like to thank my supervisor, Dr. Mikhail Barash, for all the support he has given me throughout my time working on this thesis. His eagerness to share his wealth of knowledge in domain-specific languages, and the countless hours of video calls, words of encouragement, and continuous support throughout my time working on this thesis has truly been invaluable to me.

I would also like to thank my co-supervisor Assoc. Prof. Anya Helene Bagge, for her valuable feedback on the thesis, and for sharing her extensive knowledge of programming with me.

I am also grateful to Knut Anders Stokke for helping me with implementation issues I have faced, and for his words of encouragement.

Finally, I would like to thank my friends and family for all the motivation and support they have given me.

Simen Lien
Wednesday 1st June, 2022

Contents

List of Figures

Listings

Chapter 1

Introduction

Domain-specific languages (DSLs) play an important role in software language engineering. DSLs are used in medical [28], process automation [17], engineering, biological, and financial[1] industries. Most industries require digitalization of business processes, and using DSLs can be highly valuable to improve the system's software quality, readability and development efficiency. One of the most extensive examples of applications of domain-specific languages in practice has been developed and used by the Dutch Tax Administration [24]. They switched their systems to use DSLs, allowing them to efficiently describe complex rules for determining government taxes and payout rules.

In general, domain-specific languages can allow for a significant increase in development productivity, software quality, and communication with domain experts [15]. In spite of these advantages, the level of adoption of domain-specific languages among users remains relatively low. This oftentimes is due to the lack of adequate tool support for a language, that is, the lack of a tailored Integrated Development Environment (IDE) that would be aware of the language in question. Normally, IDE support for a DSL is implemented manually; this requires a considerable skillset from developers of the IDE.

Recently, specialized tools, called *language workbenches*, have gained some popularity for their ability to generate high-quality IDE support for a DSL [13, 5]. Using a language workbench allows for many benefits, such as a specification of the language concepts, the specification of an IDE tailored to the DSL, and specification of the execution semantics, through interpretation and code generation. However, using language workbenches is often difficult, as it requires extensible development knowledge [13, 26]. In this thesis, we

[1] `https://dslfin.org/`

have explored a low-code/no-code [9] way to make it easier to use language workbenches by creating a tool that converts low-code/no-code example-driven programming language specification into a language workbench syntax. With this, we aim to make it easier for beginner-developers to create DSLs.

In this thesis, we implement a recently introduced *example-driven* approach to defining software languages [4]. In this approach, we specify a language by giving samples of code written in the language being defined; this is called *illustrative syntax definition* [4]. After declaring the syntax of a language, a language engineer can further annotate the samples to define validation rules, typing rules, formatting rules, and dynamic semantics[2]. After a user of our tool has implemented a language specification in an example-driven way, our tool generates an equivalent specification for that language for the language workbench *Langium*[3]. Precisely, from an example-driven specification of a language made in our tool, we generate a Langium grammar and a validation code. Langium is a Typescript-based language workbench similar to Eclipse Xtext [7, 14]. From a language specification (a grammar and imperative code that defines validations), it generates a plugin for Visual Studio Code, allowing for IDE support with tailored editor services, such as syntax highlighting, code navigation, simple refactorings, and simple static analysis for the DSL.

We have implemented a prototype of a web-based *front-end* [5] language workbench to explore the example-driven approach to creating DSLs with industry-level IDE support. To showcase an example of a DSL implementation in our tool, we have implemented a language to specify cooking recipes [6].

The thesis is structured as follows. In Chapter 2, we introduce the notion of a domain-specific language and describe when to use them. We also discuss the notions of the semantic model, language engineering, and language workbenches, following M. Fowler's book on domain-specific languages [15]. In Chapter 3, we present the concepts behind example-driven language engineering [4] and give an overview of our tool. Then, in Chapter 4, we showcase the tool by implementing a cooking recipe language in it. Here we explain how the tool works, and how to implement a DSL, with step-by-step instructions. In Chapter 5, we discuss the details behind the implementation of our tool. We discuss *Electron*[4], which is a tool that allow us to create a web-based desktop application. We also discuss *Monaco Editor* [5], which is a web-based editor that Visual Studio Code is built on top of. We use it in our tool to define the syntax of language constructs and

[2]In this thesis, our focus is on an example-driven approach to defining syntax and validation rules.
[3]`https://langium.org/`
[4]`https://www.electronjs.org/`
[5]`https://microsoft.github.io/monaco-editor/`

give visual feedback as to which properties and validations have been applied to each token. We then discuss Langium, which is the language workbench we use to generate VSCode-based IDEs with tailored IDE support. We also discuss how we use XML and XSL transformations to present and transform language specifications done in our tool to an equivalent Langium-based language specification. In Chapter 6, we present related work, where we mention some of the language workbenches, and tools related to our implementation. Finally, in Chapter 7, we discuss our implementation and what we envision as directions for future work [5].

All in all, our web-based tool is one of the first attempts to bring language engineering to users who are non-developers. Our tool is a prototype; however, with this prototype, we have demonstrated that an example-driven approach to language engineering is viable. It should be noted that we are in the early beginning of this implementation and that there is an array of directions to explore still; some of these include type systems, user-defined (custom) validations, and quick fixes.

Chapter 2

Background: Domain-Specific Languages

In this chapter, we follow chapter 2 in the book *Domain-Specific Languages* [15] by M. Fowler. Some of the things Fowler talks about in this book are what DSLs are, some of the advantages of using them, and some of the potential problems with using them. Furthermore, in this chapter we present Fowler's view on the lifecycle of a DSL, and what makes a good DSL design and DSL design process.

We start by introducing the notion of general-purpose programming languages (GPLs). Those are Turing-complete programming languages, designed with the intent of being able to build software applications for a wide variety of application domains. Examples of GPLs include Python, C, C++, Java, Ruby, and JavaScript. Domain-Specific Languages (DSLs) are the opposite of GPLs—they are small (*little*) languages built with a specific domain in mind and have limited capabilities. DSLs are not Turing-complete and oftentimes do not have variables or the ability to define subroutines. Furthermore, they usually avoid imperative control structures such as loops and conditions. Fowler classifies DSLs into two categories: external DSLs and internal DSLs. An *external* DSL is a stand-alone language which is parsed independently. Some examples of external DSLs include: CSS, Cucumber, and regular expressions. Regular expressions, for instance, have matching text as their domain focus and have limited features built-in with the goal of matching text efficiently.

Internal DSLs, also often referred to as "*embedded DSLs*", are a type of API written in a GPL. It is a certain way of using a GPL—which is in this context called "a host language"—to handle a particular aspect of the system. Importantly, an internal DSL should imitate the feel of a custom language rather than the GPL used to create it: indeed, another term for internal DSL is *fluent interface*. This term emphasizes the fact that an internal DSL is simply a type of API in a GPL with a focus on having a fluency quality to it, i.e., reading the code should have a sense of flow to it. Conversely, a term for a non-fluent API is *command-query API*. Examples of internal DSLs include jQuery [1], JOOQ [2], and JMock [3]. Another thing to note is that one is not able to build a whole program with a DSL, however, one can use multiple DSLs in a system that is mainly written in a GPL.

DSLs have a wide variety of use cases and are used across industries such as healthcare [28], finance[4], process automation [17], and legislation[5]. Moreover, they are fundamental for *language-oriented programming* [12], which is a software development programming paradigm, where programming languages are part of the software building blocks alongside objects, modules, and components. When solving problems in language-oriented programming, the programmers first create one or more DSLs, which can then be used to solve the problem at hand.

2.1 Advantages

When talking about DSLs, it is also important to talk about the *semantic model*. A semantic model is a library or framework that the DSL populates. A DSL is simply a thin layer on top of the semantic model. There are numerous advantages to using DSLs, although it is important to remember that a DSL is simply a thin layer on top of the model, where the model is a library or a framework that is being used in a project. Therefore, when considering the advantages and disadvantages of using a DSL, it is important to distinguish between the advantages and disadvantages that come from the model and the advantages and disadvantages that come from the DSL.

Improving development productivity A reason to use DSLs is that they are created with the intent of more clearly communicating the intent of a part of a system. We should

[1]`https://jquery.com/`
[2]`https://www.jooq.org/`
[3]`http://jmock.org/`
[4]`https://dslfin.org/`
[5]`https://resources.jetbrains.com/storage/products/mps/docs/MPS_DTO_Case_Study.pdf`

encourage using DSLs for the same reasons that we encourage good variable names, clear coding constructs, and documentation [21]. The easier it is to read code, the easier it is for developers to find mistakes, modify code, and reason about the program. Another problem that sometimes arises is that one is using an inconvenient third-party library. By creating a DSL for the library, one can lower the difficulty of using the library, which makes it easier and more convenient to use.

Limited expressiveness Since DSLs are created with a specific domain problem in mind, they naturally have limited expressiveness. This makes it harder to write incorrect code and easier to find errors, which improves development productivity for developers and domain experts. Furthermore, due to their limited expressiveness, DSLs are often easier to learn.

Helping design the model An advantage to DSLs is that building them can help with designing the semantic model for the DSL. When designing the DSL, one is building an abstraction over the model to use it effectively, and as such, building the DSL can help the developers find ways to improve the design of the model.

Changing execution context An advantage of DSLs is that one can use a DSL to change the execution context of a program by shifting the logic from compile-time to runtime. A relevant example of this is the extensible markup language XML. By using XML to express some state that one would usually write in a language that evaluates at compile-time, one can move the evaluation to runtime instead.

Improving communication with domain experts A reason to use DSLs is that they make it easier to collaborate with non-developer domain experts. Insufficient communication between domain experts and developers is a common source of projects failing. Providing a DSL that uses precise language, created with a specific domain in mind, allows for developers and domain experts to communicate more clearly. It also allows domain experts to read code in the language, which in turn allows them to spot mistakes in it, and communicate more effectively with the developers.

2.2 Problems with using DSLs

There are some problems too with using DSLs, however. If one does not see any of the benefits of a DSL apply to the project one is working on, then perhaps using a DSL is not worth it. Below we summarize some of points that Fowler presents as reasons that developers choose not to use DSLs in their projects.

Building cost Implementing software languages requires a considerable amount of knowledge from software engineers (such as parsing), thus making it difficult to implement them, and oftentimes not worth the effort. Most developers are not used to building DSLs, and some argue that because of this it will take developers new to creating DSLs much more time and resources to develop and maintain them. This might be true in some sense, however, it is important to consider the benefits of using the DSL anyways. Using a DSL can significantly ease development, and as such should be considered. Another point is that creating a DSL for the semantic model can help the developers learn how the semantic model works.

Need domain experts One problem is that DSL developers need domain experts to help them build and maintain the DSL since the developers usually are not domain experts in the domain they are developing the DSL for. Domain experts lack the programming knowledge to build the DSL themselves, and as such, excellent communication between the domain experts and developers is of great importance. This is not so easy to ensure in practice, however, and oftentimes there is miscommunication between them. Another point is that there might be few domain experts in a particular domain, which can raise labor costs for those trying to develop a DSL for it.

Language cacophony Oftentimes, there is a worry that languages are hard to learn, and that using multiple languages will be much more complicated than using only one. The argument is that needing to know multiple languages in order to work on a system is harder, and makes it more difficult to introduce new developers to the project. There is some truth to this argument, however, many fail to realize that DSLs are much easier to learn than GPLs. The nature of DSLs' simplicity and limited expressiveness allows users to learn them quickly. When hearing this, some argue that even though DSLs are easy to learn, having multiple languages in a project makes it more difficult to understand what is going on in the project. It is important to realize, however, that projects always have complicated areas in them that are difficult to learn. The developers would have to figure out how the underlying model in their system works anyways, and so using DSLs should make it easier to understand some of these complicated areas, ultimately reducing the learning cost. After all, the point of a DSL is to make it easier to understand and manipulate the model, and as such, using DSLs in a project should help ease development.

Evolving the DSL A problem with DSLs is that they can end up evolving into a GPL if one is not careful. A DSL should be designed with a specific domain in mind and should have a limited expressiveness aimed at the specific domain. What sometimes happens is that the DSL developers add more and more features to the DSL to the point where it essentially becomes a GPL, which ruins the core idea of DSLs. If new features are needed that require complicating the DSL, one should consider creating a new DSL for that purpose, or see if the features can be implemented in a host GPL instead. Also, sometimes it can suffice to just use some of the language processing techniques that we normally use with a DSL, without actually creating a DSL.

DSL lifecycle There are different approaches to building a DSL, and there is not necessarily one that is the correct one. One way is to start with the model and build the DSL on top of it. Here one has an existing semantic model that needs to be populated by a DSL. Another way is to build both the model and the DSL at the same time. And another way is to build the DSL first and then the model afterward. With all of these, it is a good idea to design while communicating with domain experts. This allows DSLs to be a good communication point for the developers and domain experts.

2.3 DSL design

When designing a DSL it is important to focus on the DSL readability. Developers and domain experts need to be able to read the code both clearly and quickly. A good way to go about designing a DSL is an iterative approach. One can experiment with multiple designs, showcase them to others and get feedback on what works and what does not. It is important to not be afraid of trying out different designs, as the process of trying and failing is ultimately what puts one on the path to a good DSL.

Additionally, if the users of a DSL are familiar with the domain, then using domain-specific technical language in the DSL will help enhance communication. Trying to make the DSL read like a natural language is not advised, because it leads to a lot of syntactic sugar which complicates understanding the semantics. It is important to remember that a DSL is a programming language and that using it should feel like programming, with the greater conciseness and precision that programming languages have over natural languages. Moreover, it is advised to take advantage of common conventions, such as for a Java-style language, to use `//` for comments, { } for brackets, and so on.

2.4 Syntax and semantics

The semantic model is a library or framework that the DSL populates. When discussing programming languages, one distinguishes between syntax and semantics. The syntax is the legal expression of the program, and the syntax for a DSL is captured by the grammar. The semantics of a program is what it means, i.e., what it does when it executes. For a DSL, it is the model that describes the semantics.

Why use a semantic model Fowler argues that we should almost always use a semantic model when designing a DSL. There are multiple reasons for this. A semantic model allows for a clear separation of concerns between the result semantics and parsing of a language. One is also able to reason about the model, as well as debug and make changes to it without thinking about the DSL. Furthermore, one is able to evolve the semantic model and the DSL independently, allowing one to build new features in one before figuring out how to expose them in the other. One is also able to test the semantic model separately from the DSL itself, which can help the developers reason about the correctness of the program.

An important question to answer is if a DSL is needed at all. A DSL is simply a thin layer on top of the semantic model, and many times our model is a library or framework that works well enough without a DSL. Although, it is important to note that a DSL can enhance the capabilities of a model. As previously noted, a DSL can make it easier to understand what the code is doing, and some DSLs allow you to configure the model at runtime. Some libraries or frameworks are awkward to use, so creating a DSL on top of it can make it easier to work with.

2.5 What is software language engineering

Software language engineering is the concept of engineering languages and their tools required for creating software. When creating a new language, some fundamental choices have to be made, such as the syntax of the language and which paradigms to follow. One needs to decide if it should be a compiled or interpreted language, if it should implement a standard library, and if it should support tools such as editors and build systems. When engineering new DSLs, we use language workbenches since they are designed to help developers develop new DSLs, alongside the tools required to use the DSLs effectively, such as high-quality IDE support.

2.6 Language workbenches

A software language needs an Integrated Development Environment (IDE) that can interpret it, and have all the standard features such as syntax highlighting, auto-completion, code validation, and refactoring, such that the language can be used practically. For a newly generated DSL to have IDE support, an IDE with support for the specific DSL needs to be generated. This can be done using a *language workbench* [13], which is a tool or set of tools typically allowing for the specification of a meta-model or language concepts, editing environments for the DSL, and specification of the execution semantics [13]. Many different language workbenches exist; some examples include JetBrains MPS[6], Spoofax[7], Eclipse Xtext[8], and Langium[9].

Historically, a substantial disadvantage for external DSLs has been that they lacked sufficient IDE support. An external DSL is a new language that needs auto-completion, syntax highlighting, support for refactoring, etc., in an IDE, and this would normally have to be implemented manually. However, this issue is not as prominent anymore, since now we have language workbenches that can create new DSLs as well as tailored IDEs to support them, allowing these DSLs to be used effectively.

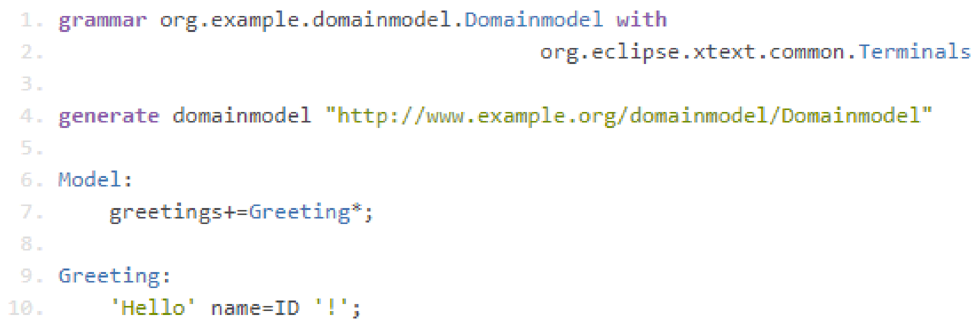
Using a language workbench requires developers to know about language development. One needs to learn how to create the grammar, as well as the validations for the DSL,

[6]`https://www.jetbrains.com/mps/`
[7]`https://www.spoofax.dev/`
[8]`https://www.eclipse.org/Xtext/`
[9]`https://langium.org/`

which can be time-consuming to learn how to do. Ideally, domain experts would be able to specify the grammar and validations themselves, as they know the domain best, however, this does not happen in practice, since domain experts lack the developer knowledge required to do so.

2.7 Language workbenches Eclipse Xtext and Langium

Eclipse Xtext [7] is a language workbench that can be used to build DSLs with high-quality IDE support. Eclipse Xtext uses a carefully designed grammar language, which itself is a domain-specific language, in order to describe textual languages. To showcase how Xtext grammars are structured, let us look at an example of a simple Xtext grammar from the website of Eclipse Xtext [10]. Here we have a grammar where one can say `Hello` to various persons; an example of a valid statement in this language is: `Hello bob!`. Figure 2.1 shows an Xtext grammar declaration for such a language.



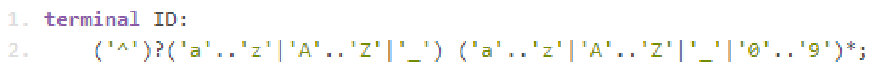
```
 1. grammar org.example.domainmodel.Domainmodel with
 2.                                       org.eclipse.xtext.common.Terminals
 3.
 4. generate domainmodel "http://www.example.org/domainmodel/Domainmodel"
 5.
 6. Model:
 7.     greetings+=Greeting*;
 8.
 9. Greeting:
10.     'Hello' name=ID '!';
```

Figure 2.1: An Xtext grammar for greeting people.

On line 1 in Figure 2.1 we define the name of the grammar to be `Domainmodel`, and on line 2 we use a mechanism called *grammar mixin* [7] to allow terminals to be reused in our language. Terminals are a set of already defined grammar rules in Xtext, such as `ID` used on line 10. Instead of defining the lexical rules of `ID` again, we inherit it from `org.eclipse.xtext.common.Terminals`. Line 4 is an easy way to have Xtext infer the Ecore model from the grammar. The ID rule defined in Terminals can be seen in Figure 2.2.

On line 9 in Figure 2.1, we define a new parser rule called `Greeting`. This rule has the syntax of `'Hello'` followed by an identifier and then an exclamation mark. On line

[10]`https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html`



```
terminal ID:
    ('^')?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

Figure 2.2: Terminal rule for ID defined in org.eclipse.xtext.common.Terminals.

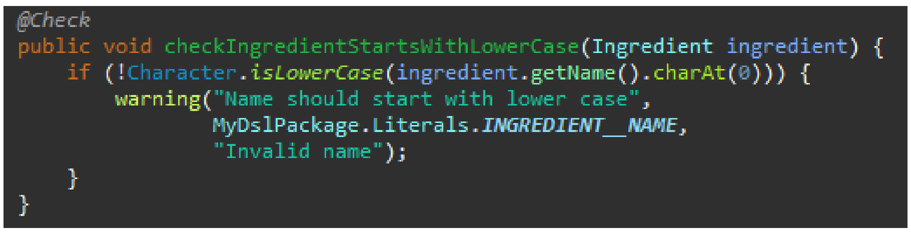
6 we define the entry rule for the grammar. The asterisk symbol (`*`) specifies that we can have zero or more `Greetings` in the DSL.

After the grammar has been defined, Xtext is able to generate a working IDE with high-quality IDE support for the DSL. An IDE provides many important features for a language to be viable to use, such as syntax highlighting, auto-completion, validations, and quick fixes.

Syntax highlighting allows code recognized by the program to be displayed in a different manner such as displaying using various coloring and formatting options. Syntax highlighting enhances the experience for the user of reading and writing code in the language. As an example, a DSL with the keyword `Hello` in it would display `Hello` differently to non-keywords, to easily showcase that it is a keyword.

Auto completion can give suggestions as to what to write when writing code in an IDE. It can make coding easier by giving the user the different possibilities for what they can write, which in turn can allow them to gain a better overview of what can be written in the language.

Validations allow for checking that certain conditions hold when using a language in an Integrated Development Environment. In Eclipse Xtext, one is able to define custom validations in a (partly) declarative way. Validations are declared together in one file, and each validation is declared as a method and annotated with the `@Check` annotation such that it will be invoked automatically when validation takes place.



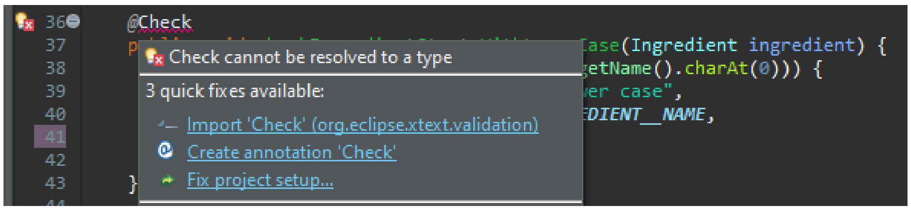
```
@Check
public void checkIngredientStartsWithLowerCase(Ingredient ingredient) {
    if (!Character.isLowerCase(ingredient.getName().charAt(0))) {
        warning("Name should start with lower case",
                MyDslPackage.Literals.INGREDIENT__NAME,
                "Invalid name");
    }
}
```

Figure 2.3: An example of an Xtext validation to check if an ingredient starts with a lower case letter.

Figure 2.3 shows an example of an Xtext validation. The name of the method is `checkIngredientStartsWithLowerCase`. It is conventional to give the method a descriptive name that explains to the reader what the method intends to validate. The method takes in an object `Ingredient`, which has been defined in the DSL grammar, and methods for accessing its properties have been created, such as `.getName()`. One can use these methods to access the properties and do validation checking on them while the user is using the DSL in the IDE. In the case of this example, we access the `name` property of an `Ingredient` instance to figure out if the first letter is lower case or not. If it is not lower case, we throw a warning with a fitting error message by using the `warning` method.

Quick fixes are another feature that Eclipse Xtext provides support for. Quick fixes (also known as *intentions*) allow for fixing warnings and errors in the IDE quickly. It is also possible to define custom quick fixes that can fix the custom validations one has created. An example of such a quick fix is to solve a warning for a name starting with an upper case letter when it should start with a lower case letter. Applying this quick fix will allow the name to be fixed such that it starts with a lower case letter.

Figure 2.4 shows an example of an issue that could be resolved with a quick fix. `Check` is not able to be resolved, and the IDE has a built-in quick fixes to try to solve the issue.



```
@Check
p                                          ase(Ingredient ingredient) {
                                          etName().charAt(0))) {
                                          er case",
                                          DIENT__NAME,

}
```

Check cannot be resolved to a type

3 quick fixes available:

- Import 'Check' (org.eclipse.xtext.validation)
- Create annotation 'Check'
- Fix project setup...

Figure 2.4: Quick fixes for resolving that `@Check` does not exist/is not imported.

Here one can use quick fixes to quickly resolve the issue. Clicking on `Import 'Check' (org.eclipse.xtext.validation)` imports `Check` from Xtext, and fixes the issue immediately without the user having to do anything else. Ideally, we would be able to define such quick fixes in our tool.

With the goal of allowing the IDE support to be implemented and distributed independently from any editor or IDE, Xtext supports *Language Server Protocol* (LSP) [16]. The LSP is used between an editor or IDE and a language server, where a language server provides programming language-specific features such as auto-completion, go to definition, find all references and more.

Langium Recently, a tool very similar to Xtext has appeared on the market under the name of *Langium*. While Xtext is Java-based and built upon Eclipse and ANTLR, Langium is Typescript-based and built upon Visual Code and Chevrotain[11]. Both Xtext and Langium use the same strategy of having a grammar declaration of the DSL that everything depends on, and they roughly provide the same functionality as language workbenches. Xtext is a tool that allows one to build traditional programming languages as well as DSLs. Xtext can be used to create ecosystems of languages of different complexity. Langium on the other hand, is more focused on creating simple individual DSLs or low-code languages [1].

[11] `https://chevrotain.io/docs/`

Chapter 3

Defining languages in an example-driven way

Contrary to many other language engineering tools, the tools that we have implemented in this thesis uses an example-driven approach to define languages. In an example-driven approach [4], language engineers can start prototyping the language by writing samples of code written in it, instead of the usual approach where language engineers have to define a metamodel first. When designing a DSL using an example-driven approach, one would define each rule in the DSL by giving an example of it. The first step is to give an *example* of a rule written in the DSL. This is used to define the concrete syntax. The next step is to *annotate* the tokens in the syntax. This allows for defining abstract syntax, formatting, validation rules, as well as other aspects of language definiton (such as typing rules or dynamic semantics).

3.1 Why create DSLs in an example-driven way

Implementing DSLs can require a lot of implementation work, and can be difficult for beginner programmers to develop. However, defining DSLs in an example-driven way can shorten development time and make it easier for developers and domain experts to develop DSLs. An example-driven approach generates the DSL based on examples of the language given by the user, which removes the need to implement the DSL from scratch themselves. Defining DSLs in an example-driven way can lead to improved communication between domain experts and developers, since the examples are easier for domain experts to understand as well. In an ideal scenario, the domain experts are able to define (simple) languages themselves.

3.2 Example-driven DSL development

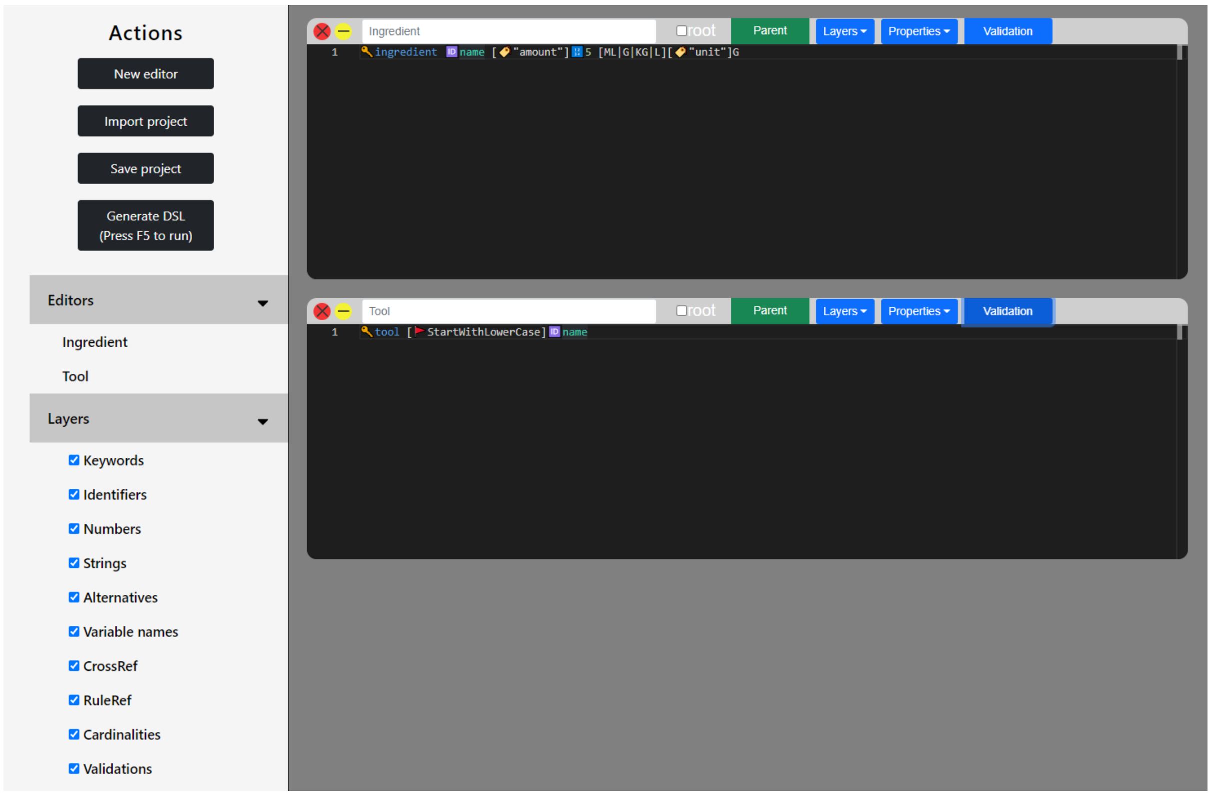
Specification of software languages in an example-driven way is done by giving examples of code written in the language using an illustrative syntax definition [4]. For each rule, the abstract syntax is defined, and each token is annotated such that an equivalent grammar rule can be generated and used in a language workbench, allowing a working DSL to be generated and used in a tailored IDE. An example-driven approach to DSL development may allow beginner engineers and domain experts to create DSLs more easily.

3.3 Overview of our tool

We have developed a web-based language workbench that allows one to define language constructs such as syntax, properties of the individual tokens, and validations to get the desired behavior of the IDE [1]. In order to generate an IDE for our language, we use Langium, which is a language workbench that allows for creating new DSLs by defining the grammar and validations for the DSL.

In our tool, in order to specify various aspects of the language definition, the user annotates each token in every rules with properties. For example, when defining the concrete syntax of a token, a user may use the following properties: *keyword* (to mark that a token is a keyword and should have an according syntax highlighting in the IDE), *identifier* (to mark that a token is an identifier), *number* (to mark that a token is an integer number), *string* (to mark that a token is a Java-like string), *alternatives* (to mark that a concrete value of a token should be one among the list of given alternatives), *variable name* (to specify the name of the feature—in Xtext terms—associated with the token), *cross reference* (to mark that the value of a token should come from one of tokens associated with a cross-refered token), and *cardinality* (to mark the multiplicity of a token). The user can drag and drop properties and validations between tokens in the same editors and across them. When dragging one token to another, the user gets prompted to choose which properties they want to copy onto the other token, allowing the user to choose exactly which properties and validations should be applied to the other token.

[1]The implementation of our tool is available at `https://git.app.uib.no/Simen.Lien/example-driven-software-language-engineering`.



```
ingredient name ["amount"] 5 [ML|G|KG|L]["unit"]G
```

```
tool [StartWithLowerCase] name
```

Figure 3.1: An overview of our tool. On the left, the user can create new editors, import projects, save projects, and generate a DSL based on the rule specifications in the editors. The user is also able to turn on and off the property and validation layers in each editor.

In our tool, the user is able to define validations for tokens in a rule, which our tool will use to create Langium validations such that the IDE will give warnings and errors when using the DSL depending on the validations defined by the user. An example of a validation is that a token should start with an upper case letter. Then, when using the DSL in an IDE, if the token does not start with an upper case letter, a warning will be shown in the IDE. We have a set of predefined validations in our tool (shown in Figure 4.8) that the user can choose between, and it is possible for a token to have multiple validations applied to it. Some of the possible validations for a token are: that it should start with an upper or lower case letter, that the word should be upper or lower case, that the length of a word should be of a given size, or that is should be equal or not equal to a word. There are also validations for numbers, such as the number being between two values, or greater than, less than, or equal to a certain amount.

After all language constructs have been defined by the user, our tool is able to generate Xtext/Langium grammar for the language, and TypeScript code for Langium, which implements the syntax validations defined in our tool for the IDE. Following this, Langium is invoked such that it generates an extension for Visual Studio Code[2] from the grammar and the TypeScript validation code that we generated. The user is then able to utilize this extension to create programs in the DSL in Visual Studio code. Thus, our tool is a *front-end language workbench* [5], since the language specification done in our tool is converted to an equivalent language specification in a traditional language workbench, in our case Xtext/Langium.

[2]`https://code.visualstudio.com/`

Chapter 4

Case study: Cooking Recipe Language

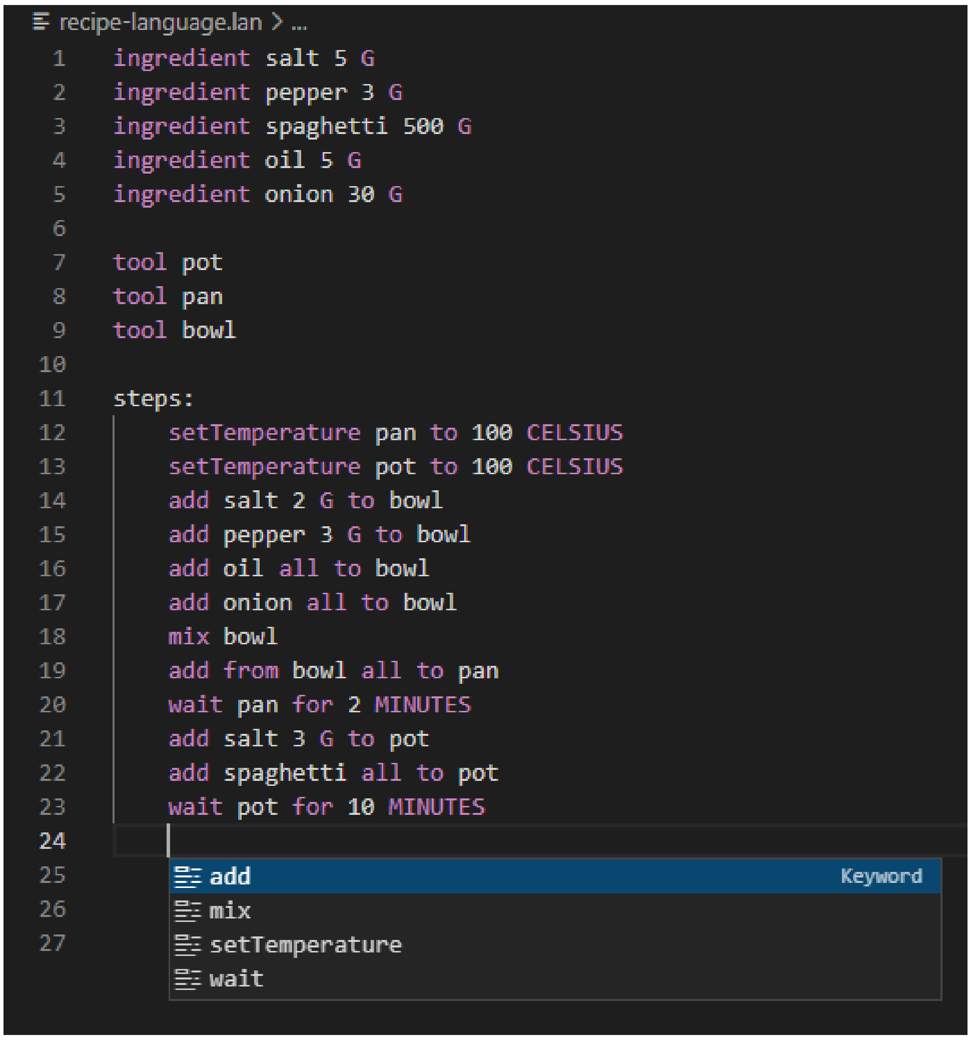
To give a concrete example of a language definition in our tool, we create a DSL for a cooking recipe language. In our language, we want to be able to define: ingredients, tools, and steps for a cooking recipe. It should be possible to specify how much of each ingredient we need for a recipe, which tools are going to be used in the recipe, and the steps that need to be taken in order to create the meal. Figure 4.1 shows this DSL working in VS Code.

We showcase now in detail how this DSL can be defined in an example driven way with the use of our tool.

First, we create a new editor and start by defining our first rule, `Ingredient`. We give the rule the name `Ingredient`, and define its *illustrative syntax* [4] to be `ingredient salt 5 G`, which is supposed to declare an ingredient `salt` is available in the amount of up to 5 grams to be used in the recipe. as shown in Figure 4.2

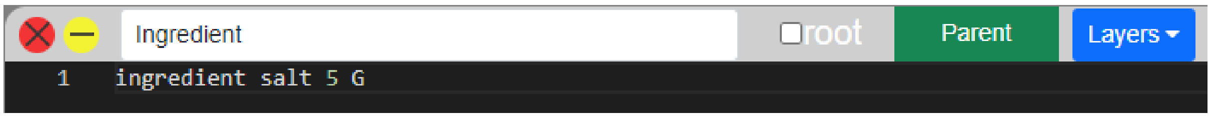
We then click on our first token (`ingredient`) and make it a keyword by clicking on *Properties* at the top of the editor, and then selecting `keyword`, as can be seen in Figure 4.3 We have now added the property "to be a keyword" to the token `ingredient`.

Now, we move on to the second token (`salt`), which we want to define as an identifier; later, when referencing an ingredient, we will use its name as the handle. We make `salt` into an identifier by clicking on `salt`, then clicking on the *Properties* button and choosing `identifier`. We then move on to the token (`5`), to which we will add a `number` property


recipe-language.lan > ...

```
 1  ingredient salt 5 G
 2  ingredient pepper 3 G
 3  ingredient spaghetti 500 G
 4  ingredient oil 5 G
 5  ingredient onion 30 G
 6
 7  tool pot
 8  tool pan
 9  tool bowl
10
11  steps:
12      setTemperature pan to 100 CELSIUS
13      setTemperature pot to 100 CELSIUS
14      add salt 2 G to bowl
15      add pepper 3 G to bowl
16      add oil all to bowl
17      add onion all to bowl
18      mix bowl
19      add from bowl all to pan
20      wait pan for 2 MINUTES
21      add salt 3 G to pot
22      add spaghetti all to pot
23      wait pot for 10 MINUTES
24      |
25
26
27
```

Figure 4.1: An example of our cooking recipe language running in VS Code, with support for auto-completion, code validation, syntax highlighting, and more. Line 24 shows an example of suggestions in the IDE.

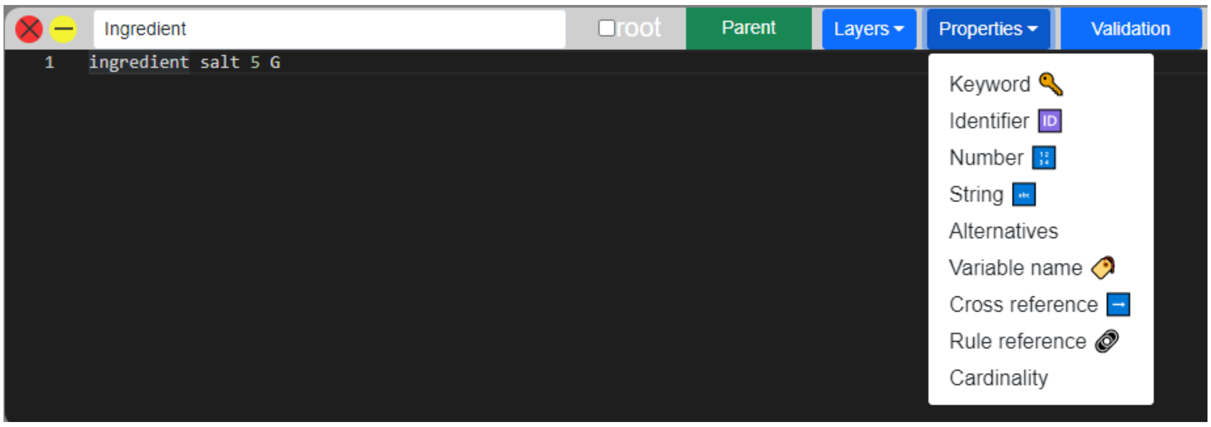


```
ingredient salt 5 G
```

Figure 4.2: The syntax for the rule `Ingredient`. It is currently not annotated.

by following the same process as previously and choosing `number`. We then give it a variable name `amount`.

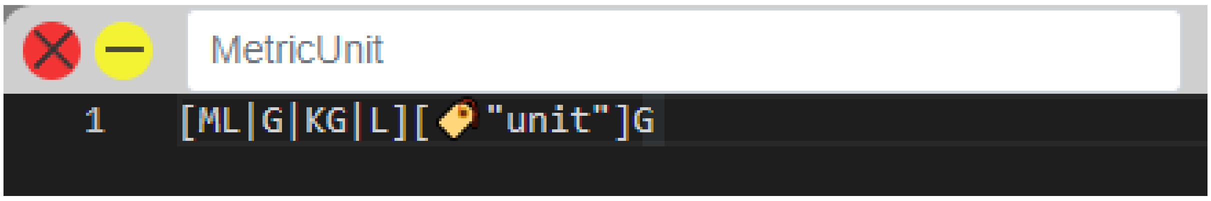
Finally, we want to define the different measurement units that token associated with example `G` can be. Since we will be referencing these units from multiple rules, we create a new rule for it called `MetricUnit`, define its syntax to be `G` and add property alternatives `ML|G|KG|L` to it, and give it a variable name `unit`, as seen in Figure 4.4. Now, in the `Ingredient` rule, we click on token (`G`), then *properties*, and selecting *rule reference* and



```
ingredient salt 5 G
```

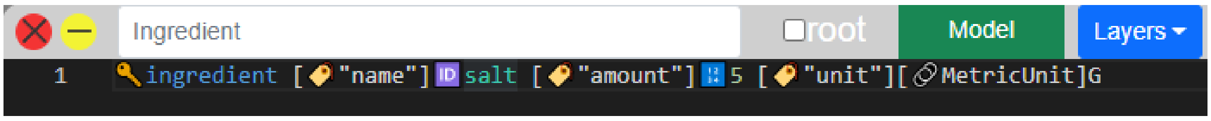
Figure 4.3: A view showing the properties available that can annotate a token in the syntax.

choose `MetricUnit`.



```
[ML|G|KG|L]["unit"]G
```

Figure 4.4: The definition of the rule `MetricUnit`.

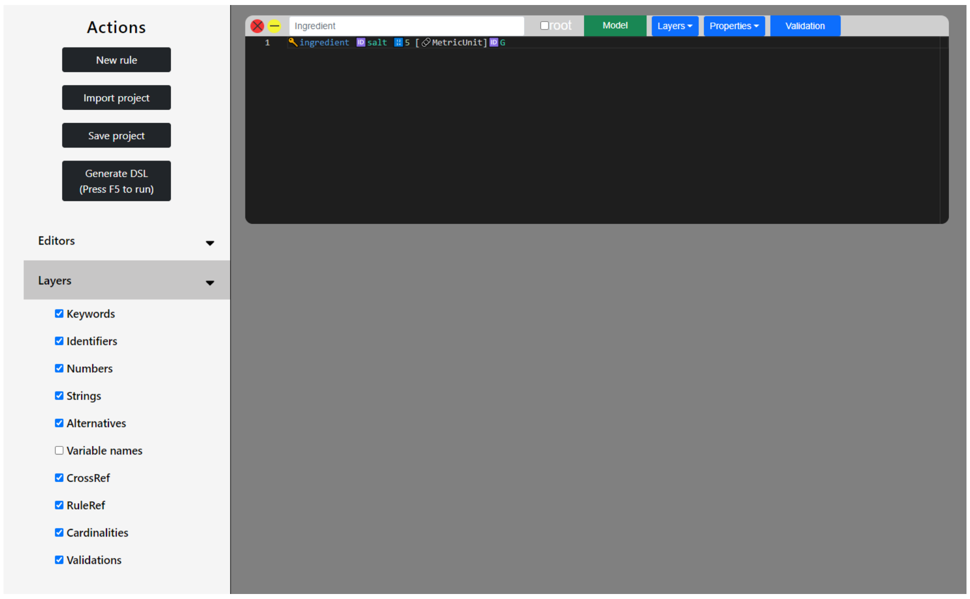


```
ingredient ["name"]salt ["amount"]5 ["unit"][MetricUnit]G
```

Figure 4.5: The definition of the rule `Ingredient`.

It is also possible to turn on and off layers for the properties, such that if one only want to see certain properties, or none at all, one can choose to do so. This can be seen in the left-hand side of Figure 4.6. In terms of Xtext grammar language, we have now defined two nonterminal symbols with the following rules, as can be seen below. They use `ID` and `INT`, which are terminal rules that are already predefined in the tool.

```
Ingredient:
    'ingredient' name=ID amount=INT unit=MetricUnit;

MetricUnit:
    unit=('ML'|'G'|'KG'|'L');

terminal ID: /[_a-zA-Z][\\w_]*/;
```

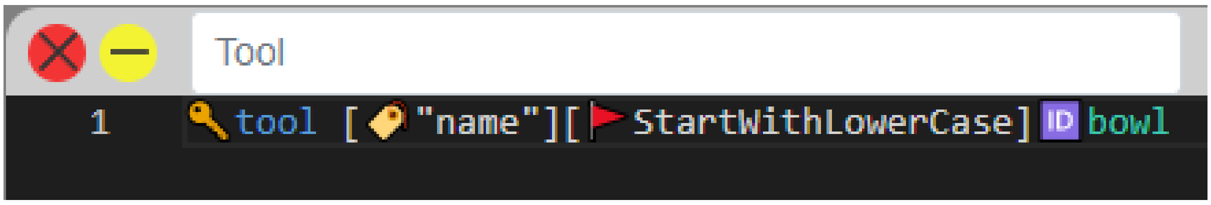


```
ingredient salt 5 [MetricUnit] G
```

Figure 4.6: Layers menu in our tool.

```
terminal INT returns number: /-?[0-9]+/;
```

Continuing the definition of our recipe language, we create a rule for `Tool`. We want each tool to have a name that we can reference it by. We create a new editor and define its rule name to be `Tool`, and its illustrative syntax to be `tool bowl`. Then, we define the token (`tool`) to be a keyword and the token (`bowl`) to be an identifier, and give it a name `name`. There are numerous validations to choose from, as seen in Figure 4.8. For our tool rule, we want to add a validation that (`bowl`) should start with a lower case letter. Figure 4.7 shows our implementation of the rule, `Tool`.



```
tool ["name"][StartWithLowerCase] bowl
```

Figure 4.7: The definition of the rule `Tool`.

A corresponding Xtext grammar rule for this illustrative syntax definition is as follows.

```
Tool:
    'tool' name=ID;
```

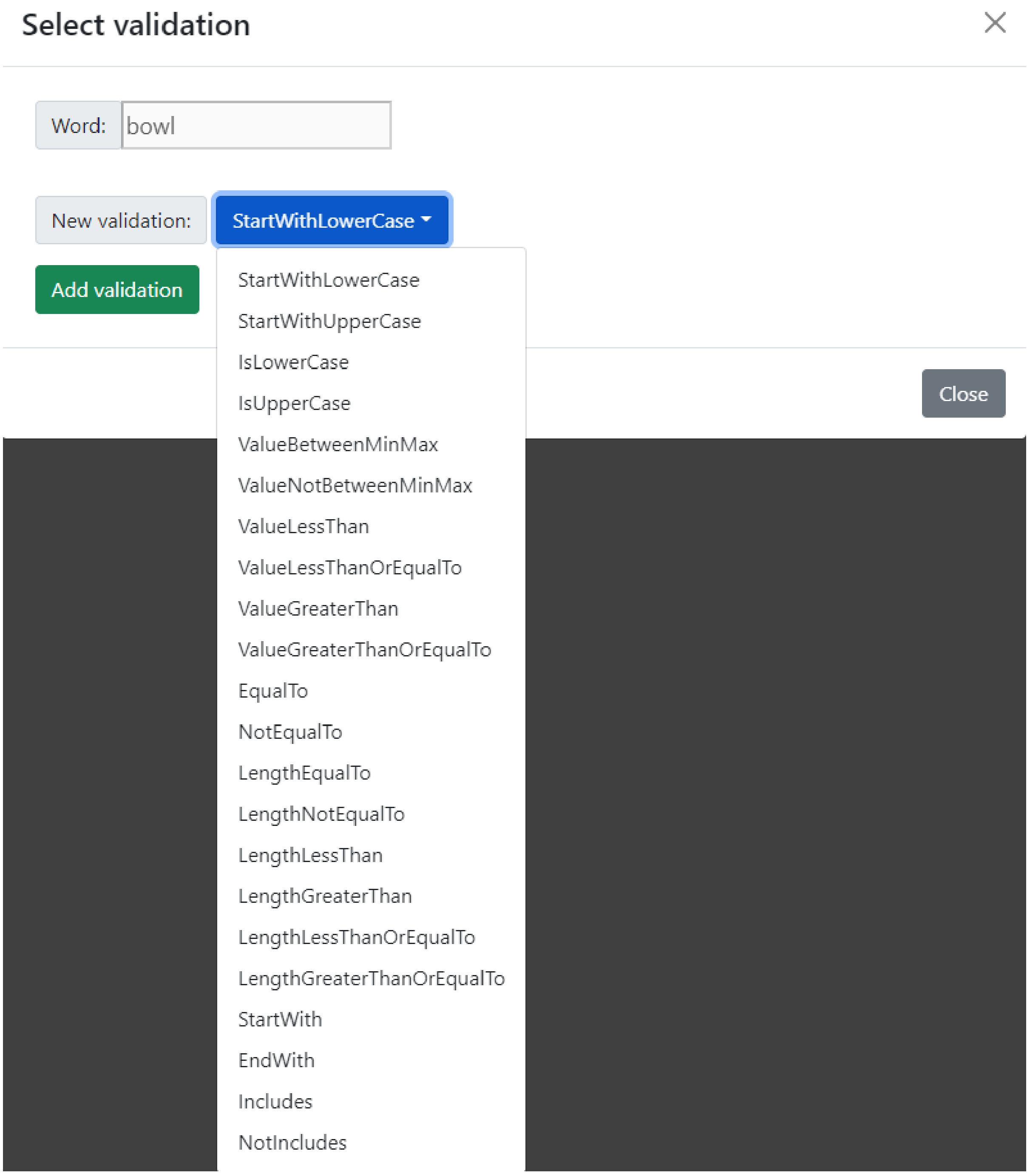
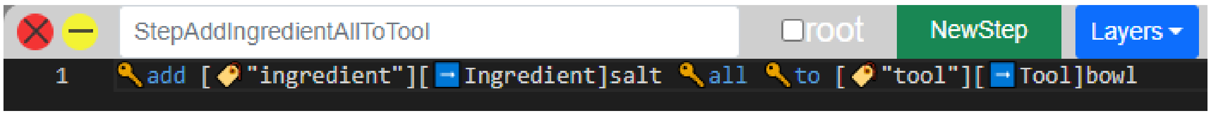


Figure 4.8: Predefined validations available in our tool.

Now that we have both `Ingredient` and `Tool` defined, we need to create some rule definitions for how these should be used in a recipe. We create a new editor for the rule `StepAddIngredientAllToTool`, as shown in Figure 4.9. We define its illustrative syntax to be `add salt all to bowl`, and then we annotate the tokens (`add`, `all`, `to`) to be a keyword. For the token (`salt`), we annotate it to be an identifier with a variable name `ingredient`, and add a cross-reference to the rule `Ingredient` which we created earlier. A cross-reference is a reference to an object already defined somewhere else, i.e., when we have a cross-reference, the referencing object has a link to the referenced object[1]. Then we click on token (`bowl`), and add a cross-reference to the rule `Tool`, and give it a variable name `tool`.



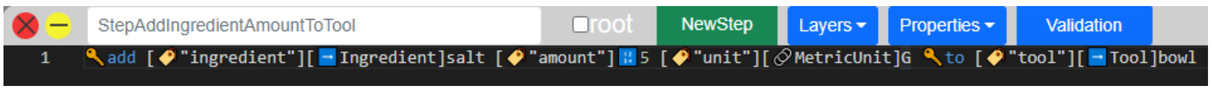
```
add ["ingredient"][Ingredient]salt all to ["tool"][Tool]bowl
```

Figure 4.9: Illustrative syntax definition for rule `StepAddIngredientAllToTool` with various annotations.

A corresponding Xtext grammar rule for this illustrative syntax definition is as follows.

```
StepAddIngredientAllToTool:
    'add' ingredient=[Ingredient] 'all' 'to' tool=[Tool];
```

Next, we want to be able to define a rule for expressing that in a recipe, one can add a certain amount of an ingredient to a tool. For that, we create `StepAddIngredientAmountToTool`, and define its illustrative syntax to be `add salt 5 G to bowl`. We annotate that the token (`add`) is a keyword, and the token (`salt`) is a cross-reference to the `Ingredient` rule, and give it a variable name `ingredient`. Then, we define the token (`5`) to be a number, and give it a variable name `amount`. We define the token (`G`) to be a rule reference to `MetricUnit`, and give it a variable name `unit`. Finally, we define the token (`to`) as a keyword, and the token (`bowl`) to have a cross-reference to the `Tool` rule, and give it a variable name `tool`. Figure 4.10 shows the complete illustrative syntax definition of the rule `StepAddIngredientAmountToTool`.



```
add ["ingredient"][Ingredient]salt ["amount"]5 ["unit"][MetricUnit]G to ["tool"][Tool]bowl
```

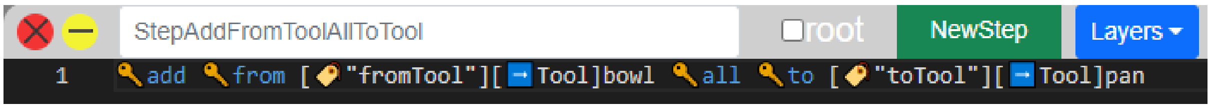
Figure 4.10: Rule `StepAddIngredientAmountToTool` defined in an example-driven way.

A corresponding Xtext grammar rule for this illustrative syntax definition is as follows.

```
StepAddIngredientAmountToTool:
    'add' ingredient=[Ingredient] amount=INT unit=MetricUnit 'to' tool=[Tool];
```

[1]https://nittka.github.io/2011/08/01/scoping1.html

We want the the cooking recipe language to support adding the entire contents of one tool into another tool. We create a new editor, and call it `StepAddFromToolAllToTool`, and define its illustrative syntax to be `add from bowl all to pan`. Then, we specify that the tokens (`add`), (`from`), (`all`), and (`to`) are keywords, and give the token (`bowl`) a variable name `fromTool`, as well as a cross-reference to the `Tool` rule. To the token (`pan`), we add a variable name `toTool`, and a cross-reference to the `Tool` rule, as is shown in Figure 4.11.



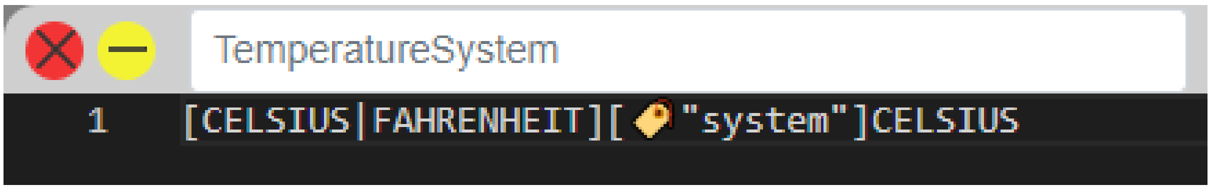
```
add from ["fromTool"][Tool]bowl all to ["toTool"][Tool]pan
```

Figure 4.11: The definiton of the rule `StepAddFromToolAllToTool`.

A corresponding Xtext grammar rule for this illustrative syntax definition is as follows.

```
StepAddFromToolAllToTool:
    'add' 'from' fromTool=[Tool] 'all' 'to' toTool=[Tool];
```

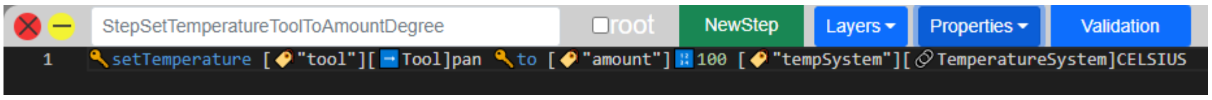
Furthermore, we want to create a rule for setting the temperature for a cooking tool, such as a pan or an oven. For that, we create a new editor, and call the rule `StepSetTemperatureToolToAmountDegree`, and define its illustrative syntax to be `setTemperature pan to 100 CELSIUS`. We define the tokens (`setTemperature`) and (`to`) as keywords, and the token (`pan`) to be a cross-reference to `Tool`, and give it a variable name `tool`. We define the token (`100`) to be a number, and give it a variable name `amount`. Then, for the token (`CELSIUS`), we want to be able to choose between two literal options: `CELSIUS` and `FAHRENHEIT`. We create a new rule `TemperatureSystem`, and define its illustrative syntax to be `CELSIUS`, and add the `alternatives` property to it with `CELSIUS` and `FAHRENHEIT` as alternatives, as can be seen in Figure 4.12. Then, for the (`CELSIUS`) token in the `StepSetTemperatureToolToAmountDegree` rule, we add a rule reference to `TemperatureSystem`, and a variable name `tempSystem`, as shown in Figure 4.13.



```
[CELSIUS|FAHRENHEIT]["system"]CELSIUS
```

Figure 4.12: The definition of possible values for `TemperatureSystem`.

Corresponding Xtext grammar rules for these illustrative syntax definitions are as follows.

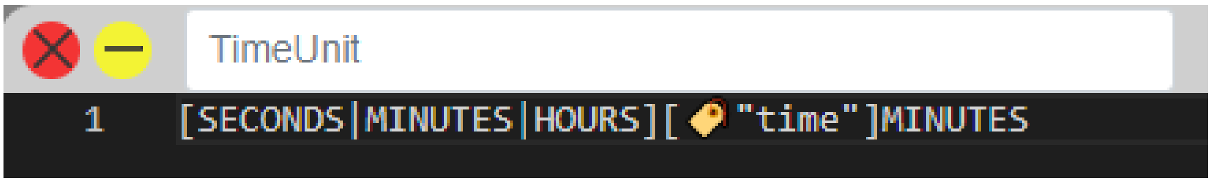


```
StepSetTemperatureToolToAmountDegree
setTemperature ["tool"][Tool]pan to ["amount"]100 ["tempSystem"][TemperatureSystem]CELSIUS
```

Figure 4.13: The definition of the rule `StepSetTemperatureToolToAmountDegree`.

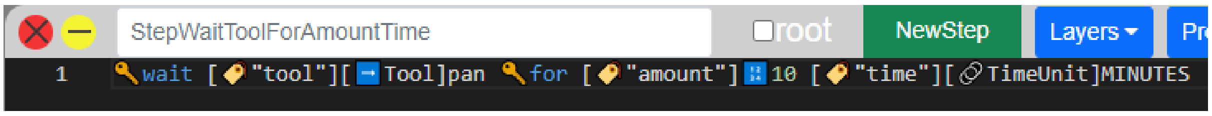
```
TemperatureSystem:
    system=('CELSIUS'|'FAHRENHEIT');

StepSetTemperatureToolToAmountDegree:
    'setTemperature' tool=[Tool] 'to' amount=INT tempSystem=TemperatureSystem;
```

Next, we want to formalize the notion of cooking something for a certain period of time. We create a new rule `StepWaitToolForAmountTime`, and define its illustrative syntax to be `wait pan for 10 MINUTES`. Then, we add the keyword property to the tokens (`wait`) and (`for`). Then, for the token (`pan`), we add a cross-reference to `Tool`, and set its variable name as `tool`. For the token (`10`), we add a number property, and a variable name `amount`. For the token (`MINUTES`), we create a new rule which we call `TimeUnit`, and define its syntax to be `MINUTES`. We give the (`MINUTES`) token a variable name `time`, and specify the possible alternatives for the value of this token (`SECONDS|MINUTES|HOURS`), as shown in Figure 4.14. Then, in the rule `StepWaitToolForAmountTime`, for the token (`MINUTES`), we add a rule reference to `TimeUnit`, and add a variable name `time`, as can be seen in Figure 4.15.



```
TimeUnit
[SECONDS|MINUTES|HOURS]["time"]MINUTES
```

Figure 4.14: The definition of the rule `TimeUnit`.



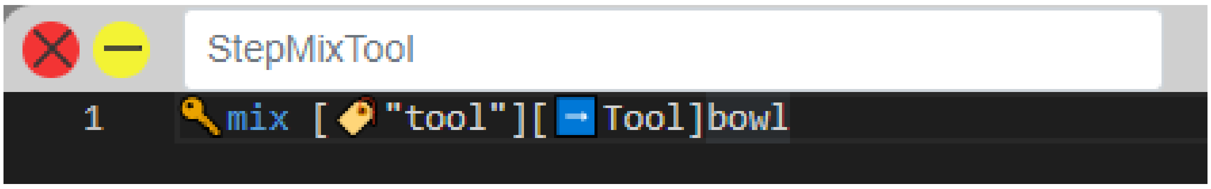
```
StepWaitToolForAmountTime
wait ["tool"][Tool]pan for ["amount"]10 ["time"][TimeUnit]MINUTES
```

Figure 4.15: The definition of the rule `StepWaitToolForAmountTime`.

Corresponding Xtext grammar rules for these illustrative syntax definition are as follows.

```
TimeUnit:
    time=('SECONDS'|'MINUTES'|'HOURS');

StepWaitToolForAmountTime:
    'wait' tool=[Tool] 'for' amount=INT time=TimeUnit;
```

Another step we would like to have in our DSL is to be able to mix a tool, such as a bowl, in order to have a rule to tell those reading the recipe to mix the ingredients that are in a tool. To do this, we create a new editor, and call the rule `StepMixTool`, and define its syntax to be `mix bowl`. We then add the keyword property to the token (`mix`), and then for the token (`bowl`) we add a cross-reference to `Tool`, and a variable name `tool`, as seen in Figure 4.16


StepMixTool

```
mix ["tool"][Tool]bowl
```

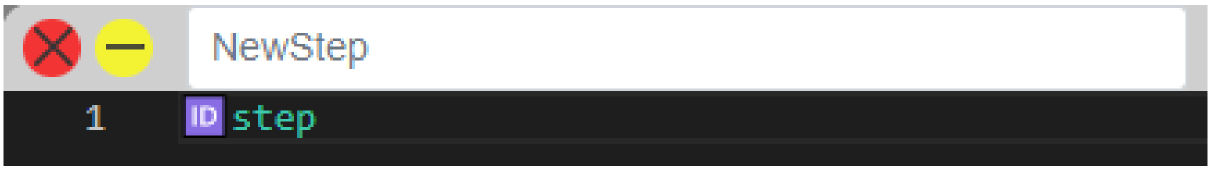
Figure 4.16: The definition of the rule `StepMixTool`.

A corresponding Xtext grammar rule for this illustrative syntax definition is as follows.

```
StepMixTool:
    'mix' tool=[Tool];
```

Now that we have defined the possible steps in our language, we want a new rule that will reference all the step rules. We create a new rule called `NewStep`, and define its syntax to be `step` and make it an identifier, as seen in Figure 4.17. We want this rule to be the parent of all the `Step...` rules, such that `NewStep` has a rule reference to all of them. In order to make `NewStep` the parent of these rules, we go into each `Step...` rule, namely `StepAddIngredientAllToTool`, `StepAddIngredientAmountToTool`, `StepAddFromToolAllToTool`, `StepSetTemperatureToolToAmountDegree`, `StepWaitToolForAmountTime`, and `StepMixTool`, and press the green `Parent` button and choose the parent to be `NewStep`.

A corresponding Xtext grammar rule for this illustrative syntax definition is as follows.

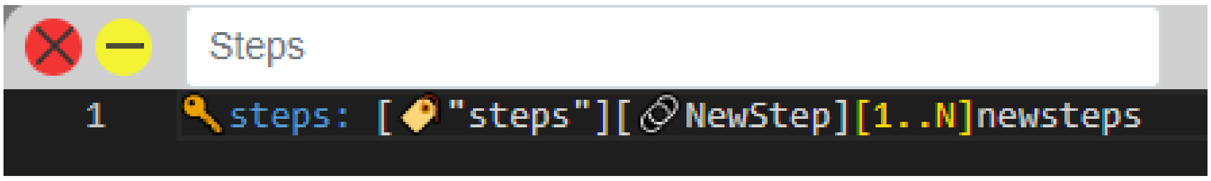


```
NewStep
step
```

Figure 4.17: The definition of the rule `NewStep`.

```
NewStep:
    StepAddIngredientAllToTool |
    StepAddIngredientAmountToTool |
    StepAddFromToolAllToTool |
    StepSetTemperatureToolToAmountDegree |
    StepWaitToolForAmountTime |
    StepMixTool;
```

In our DSL, we want to be able to write multiple steps after each other, and we can do that by using a keyword and then have multiple `NewStep` after it. Now, let us create this rule. We call it `Steps`, and define its syntax to be `steps: newsteps`. We then give the token (`steps:`) a keyword property, and the token (`newsteps`) a variable name `steps`, a rule reference to our `NewStep` rule, and cardinality `1...N`, which corresponds to the symbol `+`. The rule can be seen in Figure 4.18. With this rule, we are defining that the user can write `steps:` followed by one or more of the `Step...` rules we defined earlier.



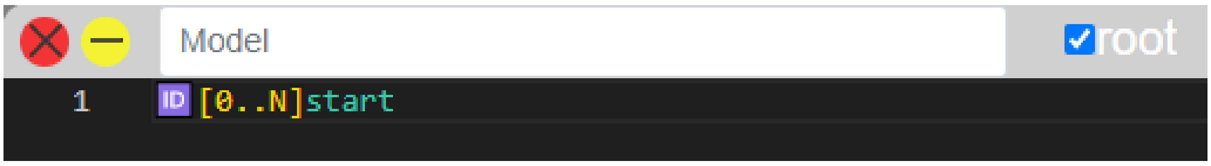
```
Steps
steps: ["steps"][NewStep][1..N]newsteps
```

Figure 4.18: The definition of the rule `Steps`.

A corresponding Xtext grammar rule for this illustrative syntax definition is as follows.

```
Steps:
    'steps:' steps+=NewStep+;
```

Every grammar needs a root/entry rule that defines what can be written in the DSL. In our cooking recipe DSL, we want to be able to define the tools to use, the ingredients to use, and the steps to do in the recipe. We create a new rule called `Model` and check the `root` checkbox at the top of the editor, since this will be our root rule. We define its syntax to be `start` (it could be any word, it does not matter) and then we give the token an identifier property, and a cardinality `0..N`, as seen in Figure 4.19. Then, we go into our `Steps`, `Ingredient`, and `Tool` rules, and click on the green `Parent` button, and choose `Model` for each of them.



```
ID [0..N]start
```

Figure 4.19: The definition of the rule `Model`.

A corresponding Xtext grammar rule for this illustrative syntax definition is as follows.

```
Model:
    (ingredients+=Ingredient | tools+=Tool | steps+=Steps)*;
```

Now we have defined a DSL in an example-driven way by using our tool. The language specifications done in our tool have been transformed into code that a language workbench understands, in our case this language workbench is Langium, and the code we have generated is for the Langium grammar and Langium validation files. The full cooking recipe language grammar generated by our tool is presented in Appendix A. We can now use Langium to generate a plugin that will allow us to use our DSL with high-quality IDE support in Visual Studio Code. This allows the user of our tool to simply define a language specification in our tool, and then click on `Generate DSL` and then they can use the DSL in Visual Studio Code. This simplicity could also make it easier for non-developers to do language specification and create a working DSL.

Chapter 5

Implementation

Language workbenches are powerful tools that can help one create DSLs with high-quality IDE support. However, learning to use language workbenches is difficult, and requires a certain amount of software development knowledge. Our tool aims to make it easier to create new DSLs with high-quality IDE support by making it easier to specify important parts of a language workbench such as the grammar and validations for a DSL. Our tool does this by letting the user specify the grammar and validations for a language workbench in an example-driven way in our tool. The language workbench that we use is called Langium and is very similar to the language workbench Eclipse Xtext. The user defines a language specification in our tool, and the tool transforms the specification into files for the language workbench by using XML to store the language specification data, and XSLT to transform it into Langium files. Once the Langium grammar and validations files have been generated, Langium is able to generate a Visual Studio Code plugin that gives the user access to a high-quality IDE with support for the DSL. In essence, our tool acts as a front-end for a language workbench, allowing the user an easier way to define grammar and validations for a DSL.

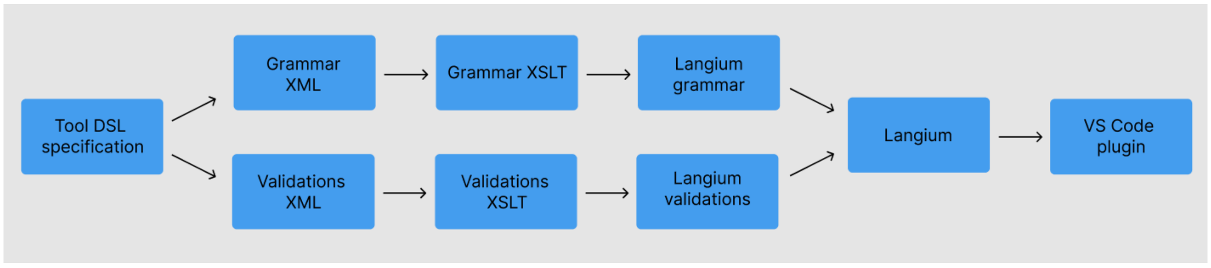


Figure 5.1: An overview of how we use the language specification in our tool to create a working DSL with IDE support by using Langium.

5.1 XML and XSLT

Extensible Markup Language (XML) is a markup language similar to HTML, where one can define tags and their data depending on specific needs. Extensible Stylesheet Language Transformations (XSLT) uses information stored in XML to transform the XML data into what we want. The tool transforms the language specification done in the tool into Langium grammar and validation files by using XML and XSLT. To give an idea of how we use XML and XSLT in our tool, we will show some examples from our implementation. Below is an example of XML generated for the rule `Tool` in our case study cooking recipe language in chapter 4. This is a rule for stating which kitchen tools to use in a cooking recipe, for instance a pot, a pan, or a bowl.

```
<rule id="Tool" code="tool bowl" isroot="false" parent="Model">
    <keyword id="tool"/>
    <token id="bowl" identifier="true" variablename="name"/>
</rule>
```

Here we have a rule called `Tool`, with the syntax `tool bowl`. It has two children, `<keyword id="tool">`, and `<token id="bowl" identifier="true" variablename="name">`. Using XSLT, we can access this information to transform it into something else. Below we explain how we transform the XML defined above into Langium grammar syntax by using XSLT.

```
<xsl:template match="rule">
    <xsl:value-of select="./@id"/>: <xsl:apply-templates/>;
</xsl:template>
```

Here we have some XSLT code. XSLT takes in some XML and goes through each rule one by one, starting with the outer parent and then going into all its children in a hierarchical way. First, the XSLT rule above matches on `<rule id="Tool" code="tool bowl" isroot="false" parent="Model">`, since it has a `rule` tag. We can then get the `id` attribute stored in the XML rule tag by using the built-in function `<xsl:value-of select="...">` in XSLT. This allow us to get the value of any attribute by specifying it in `"..."`, using the `@attribute-name` annotation. The XSLT rule above matches on the rule with `id="Tool"`, and writes `Tool:` and then `<xsl:apply-templates/>` tells it to carry on checking for other templates that will match any children rules. The children rules are `<keyword id="tool"/>` and `<token id="bowl" identifier="true" variablename="name"/>`. It will try in the order it's been declared in the XML, which means it will check for the keyword rule first.

```
<xsl:template match="rule/keyword">
    '<xsl:value-of select="./@id"/>'
</xsl:template>
```

The XML rule `<keyword id="tool"/>` matches on the rule above, and is transformed into `'tool'`. There is no further matching happening for this rule, and so we move on to the next rule, `<token id="bowl" identifier="true" variablename="name"/>`. This rule matches on the XSLT below.

```
<xsl:template match="rule/token[@variablename and
not(@crossreference) and not(@rulereference)]">
    <xsl:choose>
        <xsl:when test="not(*) and @identifier">
            <xsl:value-of select="./@variablename"/>=ID
            <xsl:apply-templates/>
        </xsl:when>
        <xsl:otherwise>
            <xsl:value-of select="./@variablename"/>=
            <xsl:apply-templates/>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

The XSLT above matches on an XML tag that is called `token` inside of a `rule` tag. The `token` tag has a variablename attribute, and not a crossreference or rulereference attribute. Then, with `<xsl:when test="not(*) and @identifier">`, we check if there is an identifier tag. The XML is `<token id="bowl" identifier="true" variablename="name"/>`, and here we have an identifier and variablename attribute, and the XSLT will output `name=ID`. It then tries to apply-templates, however, there is nothing else to check for, since this is the last of the children. It then returns to the first apply-templates call from the `<xsl:template match="rule">` and inserts a semicolon at the end.

```
Tool:
    'tool' name=ID;
```

We have now generated the output, and above is an example of the rule, Tool, now transformed into the Langium grammar syntax by the use of XML and XSLT. We are also converting validations from the programming language specification in our tool by using XML and XSLT, as shown in the Langium section in this chapter.

A somewhat non-trivial example of an XSLT implementation we had to implement was for the Langium validation file. It uses a different syntax for when there is multiple validations for a rule, and when there is only one validation for a rule. As an example, let us say we have two validations for the rule Ingredient. We have that the name of an ingredient should start with a lower case letter, and that the amount of an ingredient should be greater or equal to a certain number. Then, we also have another rule, Tool, which only has one validation, that the name should start with an upper case letter. The Langium validation file uses this syntax:

```
const checks: LangiumNameTestChecks = {
    Ingredient:
        [validator.checkIngredientStartWithLowerCase,
         validator.checkIngredientValueGreaterThanOrEqualTo],
    Tool:
        validator.checkToolNameStartWithUpperCase,
};
```

We need to generate this with the use of XML and XSLT. When there are multiple validations in one rule, we need to create an array that lists all the different validations that should be checked for that rule. The `validator.check...` is referencing methods that have been defined in the file from our XSLT. So, to figure out if there are multiple validations for one rule, we specify this in the XML, such that we can check for it in the XSLT. Below is an example of the XML:

```
<validations data-reactroot="">
    <validation name="StartWithLowerCase" filename="Ingredient"
     node="ingredient" property="name"
     rulename="checkIngredientStartWithLowerCase"
     multipleValidations="true">
    </validation>

    <validation name="ValueGreaterThanOrEqualTo" filename="Ingredient"
     node="ingredient" property="amount"
     rulename="checkIngredientValueGreaterThanOrEqualTo" min="5"
     multipleValidations="true">
    </validation>

    <validation name="StartWithUpperCase" rulename="Tool"
     node="tool" property="name"
     functionname="checkToolNameStartWithUpperCase">
    </validation>
</validations>
```

Below is a part of the XSLT for creating the correct syntax depending on if there are multiple validations or not.

```
<xsl:key name="validation-by-filename"
 match="validation[@filename]"
 use="@filename"
/>
...
<xsl:template match="validations">
    const checks: LangiumNameTestChecks = {
        <xsl:for-each select="//validation
        [count(. | key('validation-by-filename', @filename)[1]) = 1]">
            <xsl:if test="@multipleValidations">
                <xsl:value-of select="./@filename"/>: [
                <xsl:for-each select="key('validation-by-filename', @filename)">
                    validator.<xsl:value-of select="./@rulename"/>
                    <xsl:if test="position() != last()">
                        <xsl:text>, </xsl:text>
                    </xsl:if>
                </xsl:for-each>
                <xsl:text>&#10;</xsl:text>
                ]
            </xsl:if>

            <xsl:if test="not(@multipleValidations)">
                <xsl:value-of
                select="./@filename"/>: validator.<xsl:value-of
                select="./@rulename"/>
            </xsl:if>
            ,
        </xsl:for-each>
    };
</xsl:template>
```

The idea here is that we create an index/registry of elements by their filename attributes, as can be seen in `<xsl:key name="validation-by-filename" match=" validation[@filename]" use="@filename"/>`. Then we use `xsl:for-each` and select all the entries of this registry, but only the first one for those who repeat several times, such as `Ingredient`. This can be seen in `<xsl:for-each select="//validation[count(. | key('validation-by-filename', @filename)[1]) = 1]">`. Then, we check whether or not `@multipleValidations` is present on the validation. If it is, then we use a `xsl:for-each` loop to output the `@rulename` attribute values of those repeating elements.

Once all the programming language specification in our tool is converted to Langium syntax, we are able to use Langium to generate a working DSL with high-quality IDE support that the user of the tool can use.

5.2 Langium

We use Langium[1], which is a language engineering framework for creating DSLs. It has built-in support for the Language Server Protocol (LSP). The LSP works by defining the protocol between a language server and an editor or IDE. It provides features such as auto-completion, syntax highlighting, documentation on hover, go to definition, find all references, and more. Langium has a similar grammar declaration syntax to Eclipse Xtext[2] and allows for defining validations that should hold for the DSL when using

[1]https://langium.org/
[2]https://www.eclipse.org/Xtext/

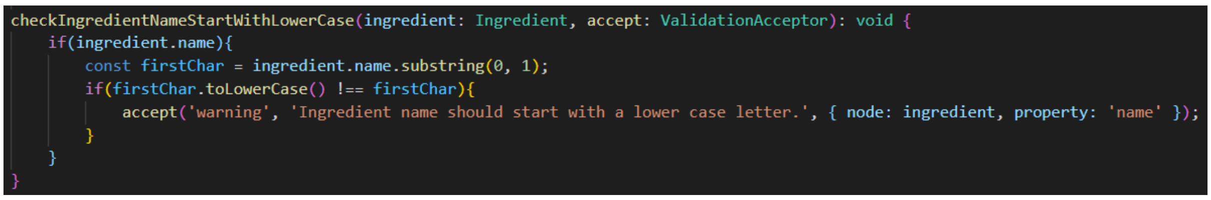
it in the IDE that it generates. In our tool we allow the user to choose between a set of predefined validations to apply to the language specification they are creating in the tool. Writing language workbench grammar and validation files can be difficult for beginner developers and domain experts that have no software development knowledge, and therefore, we have created a tool that aims to make it easier to define the grammar and validations for the language workbench Langium.

The structure of Langium is divided into two main parts, one for the CLI, and one for the language-server. The CLI folder contains files for the CLI such as `cli-util.ts`, `generator.ts`, and `index.ts`. The language-server folder contains the file for language grammar specification as well as the file for specification of validations, and code to start and run the language server. Furthermore, it contains a folder called generated which has the files `ast.ts`, `grammar.ts`, and `module.ts` in it. These are generated dependent on the grammar specification and contain an AST of the language, and code to generate it. The files that require the most change by the user are the file for defining the grammar specification and the file for defining the validation specification.

The grammar file is called `"[name of project].langium"`. The specification of the grammar itself is done in a domain-specific language that has been designed to describe textual languages. In this file, we define the grammar, consisting of the name of the grammar, the entry rule, terminal rules, and parser rules.

The file for specifying validations is called `"[name of project]-validator.ts"`. In the validations file, each validation is defined as a method. Each validation method is given a name that describes the validation, parameters for the objects that the validation should validate on, as well as a validation acceptor object that is used for creating a warning or error for the validation.

The objects' attributes can be accessed to check for conditions that should hold, and depending on these conditions we can use the acceptor object to throw a warning or error in the IDE. A bonus of it being a TypeScript file is that the type of each object can be specified, allowing for greater code robustness and readability.



```
checkIngredientNameStartWithLowerCase(ingredient: Ingredient, accept: ValidationAcceptor): void {
    if(ingredient.name){
        const firstChar = ingredient.name.substring(0, 1);
        if(firstChar.toLowerCase() !== firstChar){
            accept('warning', 'Ingredient name should start with a lower case letter.', { node: ingredient, property: 'name' });
        }
    }
}
```

Figure 5.2: Example code of a validation in Langium

Our tool generates Langium validation files depending on specifications done in our tool. As an example, in our cooking recipe language we specified that the token `name` of rule `Tool` should be a lower case letter (see Figure 4.7). In this case, it will generate a TypeScript validation file with this information for Langium, such that the validation is asserted when writing programs in the DSL. To generate this validation file, our tool generates XML for our validations and then transforms it into Langium validation files by using XSLT. For token `name` in our rule `Tool`, we would generate an XML such as:

```
<validation name="StartWithLowerCase"
rulename="Tool" node="tool" property="name"
functionname="checkToolNameStartWithLowerCase"/>
```

Then, in our XSLT, we check for each validation and create a rule depending on the XML received. XSLT has built in functions to access the values inside XML, an example of which is `<xsl:value-of select=""/>`. It can be used to access a node in the XML and return its value. For instance, `<xsl:value-of select="./@rulename"/>` returns the value of node `rulename` defined in the XML above, giving us the string "Tool". The child code of an `xsl:template tag` is what XSLT will generate, and we can use functions such as `<xsl:value-of select="...">` to specify what variables to insert into the otherwise static file. Below is an example of XSLT for generating a Langium validation file for a token that has the `StartWithLowerCase` validation.

```
<xsl:template match="//validation[@name='StartWithLowerCase']">
    ...
    const firstChar = <xsl:value-of select="./@node"/>.<xsl:value-of
    select="./@property"/>.substring(0, 1);
    if(firstChar.toLowerCase() !== firstChar){
      accept('warning', '<xsl:value-of select="./@rulename"/>
      <xsl:value-of select="./@property"/> should start with a
      lower case letter.', { node: <xsl:value-of select="./@node"/>,
      property: '<xsl:value-of select="./@property"/>'
      });
     }
    ...
</xsl:template>
```

The XSLT above uses the XML and generates the following validation code for Langium:

```
const firstChar = tool.name.substring(0, 1);
if(firstChar.toLowerCase() !== firstChar){
    accept('warning', 'Tool name should start with a
    lower case letter.', { node: tool, property: 'name'
    });
}
```

After updating the Langium grammar and validation files, Langium is able able to generate an IDE that supports our DSL. To do that, one can go into a terminal and go to the root folder of the Langium project and run the commands "npm run langium:generate", followed by "npm run build" (this happens when the `Generate DSL` button in the tool is pressed). Then, Langium generates a plugin for Visual Studio Code that allows the DSL specified to be run in VS Code with high-quality IDE support and support for validations specified in Langium. After running the two commands above, we press F5 to start the plugin such that we can use the DSL in VS Code. We can then create a new file that ends with `.lan`, and use our DSL in VS Code. Figure 4.1 shows an example of the DSL we defined in the case study in chapter 4 being used in VS Code.

5.3 Electron

Electron[3] is a framework using Chromium and Node.js[4] that lets one create native applications with the use of web technologies such as JavaScript, HTML and CSS. Electron allows for easy access to Node.js and the users' local filesystem. This allows one to run terminal commands for Langium, update the grammar and validation files in Langium, request Electron dialogs to show to the user, as well as save and import projects in our tool. In order to do this, we use Electron's 2 IPC (Inter Process Communication) modules, *ipcMain* and *ipcRenderer*.

Electron uses Event Emitters called ipcMain and ipcRenderer in order to communicate asynchronously between the main process and the renderer process. The main process

[3]https://www.electronjs.org/
[4]https://nodejs.dev/

can define methods for listening to events from the renderer processes. The methods defined in main process can use Node.js, initiate Electron's dialogs, and more. This allows any renderer processes to run Node.js commands by sending events to the main process. The main process can also optionally send data to the renderer process, allowing for intuitive communication between them. Many well-known applications use Electron, such as Visual Studio Code, Discord, Slack, and Facebook Messenger.

5.4 React

We have used React[5] for the front-end development. React is a component-based, declarative, object-oriented style library for JavaScript. It allows one to update only the parts of the web page that have been changed because of React's concept of reconciliation[6]. It takes the Document Object Model (DOM), which is a representation of what one sees in the browser, and creates a virtual DOM whenever a change happens. Then, it compares the actual DOM and the virtual DOM and uses a diffing algorithm in order to figure out if changes have been made such that the web page needs to be updated, and then updates that part of the DOM. React allows one to manage the state and view for each component, and send data from component to component through props [7].

5.5 Monaco Editor

We use Monaco Editor[8] to let the user define each rule in their grammar. Monaco Editor is an editor that VS code is built on, and it allows for the use of custom languages through Monaco Editor Monarch[9]. This allows defining custom keywords, identifiers, and other parts of the language syntax in each editor. We use this to give the user visual feedback on changes they make to the grammar rules. For instance, when a user adds the property "to be a keyword" to `tool`, such as in Figure 4.7, the custom language in the editor will be updated to have `tool` as a keyword and will showcase it with a distinct keyword color in the editor. In addition, we use the Monaco Editor to give visual feedback to the user depending on the different properties and validations applied to each token in their rules,

[5] `https://reactjs.org/`
[6] `https://reactjs.org/docs/reconciliation.html`
[7] `https://reactjs.org/docs/components-and-props.html`
[8] `https://microsoft.github.io/monaco-editor/`
[9] `https://microsoft.github.io/monaco-editor/monarch.html`

as seen in Figure 4.5. When adding a keyword property to a token, we show a `key` emoji next to it, and when adding the property “to be an identifier”, we show an `ID` emoji next to it, `numbers` emoji next to a number token, and so on, as seen in Figure 4.5. Monaco Editor has support for getting the token at a position clicked in the editor. This allows us to click on a token and then use the properties or validations button to annotate that specific token. It also allows us to drag and drop properties and validations between tokens. Furthermore, the editor allows us to customize right-click options such that one can right-click on a word and choose a property to apply to it from the right-click pane. We use interact.js[10] in order to move and resize the editors.

One issue that we have run into with Monaco Editor is that some parts of the CSS are troublesome to work with. When a token is annotated with properties or validations, we want to display some CSS before it that shows which annotations are applied to the token. For instance, if a token is annotated to be a keyword, it will have a keyword emoji in front of it. If it is annotated to have a variable name, it will have a name tag emoji followed by the variable name inside square brackets. The problem we encountered is that when we want to add CSS properties in front of a token that has an annotation with a name that can change depending on the user input, such as a variable name, alternatives, cross-reference, rule-reference, etc., we are not able to do so. This means that we cannot add CSS properties to them, such as a different color, size, or font.

[10] `https://interactjs.io/`

Chapter 6

Related work

In this chapter, we will present some of the language workbenches and other tools that are related to our implementation.

Using predefined concepts to assemble new languages

Here we present some tools that allow for building languages from components. The first tool, CBS, is a framework that allows for component-based specification of programming languages [23]. Its goal is to reduce the effort for developers to specify the formal semantics of their full languages. It provides a library of reusable components, where the semantics of each component is defined once and for all. Thus, translating a programming language to a set of components allows for defining the formal semantics of the language.

Another tool that uses a similar idea is Miksilo [2]. It is a language workbench built for a modular design that allows one to build new programming languages by mixing parts of existing languages on top of each other. Language transformations are defined as reusable components, and new languages are defined by stacking language transformations onto an empty language in the tool.

Furthermore, we have Déjà Vu, which is a platform for building applications by using self-contained and reusable concepts that include both front-end and back-end functionality [25]. The concepts are developed by experts and are defined in a catalog in the platform. The idea is to import these concepts and adjust them to fit the need of the application being developed, ultimately allowing developers to program more efficiently.

Live language development

Spoofax [18] is an example of a language workbench that supports live language development. For language workbenches supporting live language development, the user can see a language specification as well as an editor for the language in the same window. If a change is made to the language specification, then the editor is immediately updated to reflect the change. This allows for quick feedback to language developers. Some similar workbenches are Jetbrains MPS [10], MetaEdit+ [19], and Racket [27]. Microsoft Oslo is another example a language workbench with support for live language development. It allows for defining a grammar and an example input, and then showing a representation of parsing the input with the grammar [4].

Example-driven modeling

An approach to modeling is *example-driven modeling* [3, 11]. This approach uses explicit examples and abstractions to model complex business knowledge, where the abstractions are generated from explicit examples. Another approach to example-driven modeling is *example derivation* [3], which is the idea of generating examples based on abstractions. Using this approach, we could imagine a language engineer using a tool similar to our implementation to define a set of annotated examples of a language. This implicitly would define the abstract syntax, allowing examples to be derived from the abstractions. Such an approach could potentially help beginner language engineers spot mistakes and inconsistencies in their implementation.

Projectional editing

Projectional editing is an alternative to editing textual source code directly. It uses a model to hold the core definition of the system, which can be edited through projections. *MacGnome* environment [22] is an example of an environment that had an editing mode that allowed one to convert code sections into plain text, and then change the text and transform it back into a structural representation. Another example, *Greenfoot* [8], allows a program to be represented as frames, where the frames are created by the use of text- and mouse-based operations. With it, one can define expressions in a textual form, and then transform them into structured expressions, thus incorporating techniques from block-based editing. *Barista* [20] is an implementation framework that supports creating user interfaces that allow for embedded interactive tools, annotations, and alternative views in the code, allowing for a non-textual, visual representation.

Language Workbench Whole platform

The Whole Platform[1] is an open-source tool used for engineering software. It provides an Eclipse-based Language Workbench, allowing the development of new domain-specific languages for the system. The Whole Platform allows for the expression of business knowledge and the development of domain-specific languages independently of the technologies used to build the products. When using the Whole Platform, the developers and domain experts would want to work together to develop the software products in the system. The domain experts create the business knowledge, and the developers develop the generators. The Whole Platform has an economy of scale. The generators are linked to the DSLs and not to specific business knowledge, and as such, software development is about building a software product line, not just a single product.

Language Workbench Cedalion

Cedalion is a programming language that is bundled with a projectional editor. In Cedalion there is no syntax in the usual sense, and it is impossible to make syntax errors. It is impossible to make syntax errors in Cedalion, because instead of letting the user write code using a traditional textual editor, Cedalion uses a *projectional editor* called *The Cedalion Workbench* which only lets the user edit code through selecting syntactic elements already defined in the language. Cedalion's syntax is extensible, allowing language constructs to be defined by describing semantic definitions for them.

Language Workbench Más

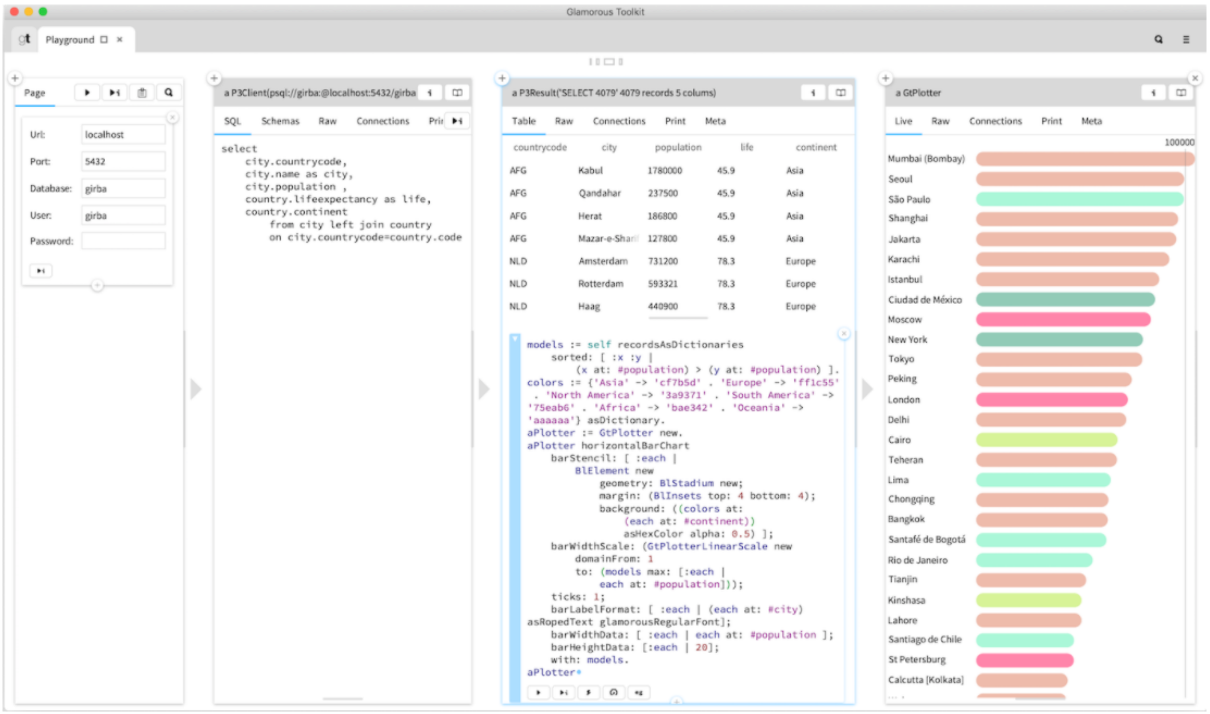
Más is a web-based workbench for projectional modelling languages. Más stands for Model-as-Service, and captures domain knowledge in a human-readable, machine-processable format and stores it in the Cloud. When using Más, one would first pick a

[1] `https://whole.sourceforge.io/`

modeling language to use, and then start using the Más model editor to do modeling. If a suitable modeling language is missing, one is able to develop their own custom modeling language. Custom languages are modeled in Más' own language definition language, which allows for it to be created, edited, and maintained just like a regular model [2].

Glamorous Toolkit

Another example of a related tool is Glamorous Toolkit. It is a tool with many features, created to help make software systems explainable. It acts as a code editor, a software analysis platform, a visualization engine, and a knowledge management system. It is a highly programmable tool with modular components, allowing its users to see a great amount of information about their systems in a neatly visualized manner.



Url:	localhost
Port:	5432
Database:	girba
User:	girba
Password:	

```
select
    city.countrycode,
    city.name as city,
    city.population ,
    country.lifeexpectancy as life,
    country.continent
        from city left join country
        on city.countrycode=country.code
```

countrycode	city	population	life	continent
AFG	Kabul	1780000	45.9	Asia
AFG	Qandahar	237500	45.9	Asia
AFG	Herat	186800	45.9	Asia
AFG	Mazar-e-Shari	127800	45.9	Asia
NLD	Amsterdam	731200	78.3	Europe
NLD	Rotterdam	593321	78.3	Europe
NLD	Haag	440900	78.3	Europe

```
models := self recordsAsDictionaries
    sorted: [ :x :y |
        (x at: #population) > (y at: #population) ].
colors := {'Asia' -> 'cf7b5d' . 'Europe' -> 'ff1c55'
 . 'North America' -> '3a9371' . 'South America' ->
'75eab6' . 'Africa' -> 'bae342' . 'Oceania' ->
'aaaaaa'} asDictionary.
aPlotter := GtPlotter new.
aPlotter horizontalBarChart
    barStencil: [ :each |
        BlElement new
            geometry: BlStadium new;
            margin: (BlInsets top: 4 bottom: 4);
            background: ((colors at:
                (each at: #continent))
                asHexColor alpha: 0.5) ];
    barWidthScale: (GtPlotterLinearScale new
        domainFrom: 1
        to: (models max: [:each |
            each at: #population]));
    ticks: 1;
    barLabelFormat: [ :each | (each at: #city)
asRopedText glamorousRegularFont];
    barWidthData: [ :each | each at: #population ];
    barHeightData: [:each | 20];
    with: models.
aPlotter
```

Figure 6.1: An example where a database is queried with SQL, and Glamorous Toolkit visualizes the result.

[2]https://mas-wb.appspot.com/site/information.html

Chapter 7

Discussion and future agenda

In this thesis, we have explored an example-driven way to define domain-specific languages.

We have implemented a tool where one is able to define syntax of language constructs together with simple validations in an example-driven way. The user defines the syntax of each rule and can attach properties and validations to each token in a rule. The tool can then transform the example-driven programming language specification into Langium grammar and validation syntax, and invoke Langium to generate a working DSL with IDE support in Visual Studio Code. From the end-user's perspective, our tool is effectively a low-code/no-code environment for software language engineering. The language workbench we have implemented in course of the present thesis is still at an early stage, and there are many improvements that we envision can be made in the future. Below we list some of the improvements that we would like to see.

Example-driven way to specify type systems, validations, and quick-fixes Currently, our tool does not support defining *type systems* for languages. In a simplified setting, one should be able to annotate tokens with the types that they have; this seems to be straightforward in cases when the type is primitive and fixed (i.e., not computed). For more practical cases, one would need to find a way to define custom object-oriented types *in an example-driven way*, i.e., by annotating examples. Moreover, we would want to see support for *custom validations* in our tool. Currently, our tool supports a set of predefined validations to show that it is possible to use them in an example-driven way. Creating support for custom validations would be ideal, as this would allow the user to specify the exact validations they want, instead of having to use a set of predefined validations. When we have implemented custom validations, it would also be ideal to be able to fix the errors and warnings quickly. To do so, we would want support for defining *quick fixes* in our tool. These can be used to solve validation errors and warnings specified in the tool. One would need to figure out a good way to define them, as our tool is supposed to be an example-driven no-code/low-code environment. Support for *quick fixes* would allow the users of the DSL to write code faster and to gain a better understanding of how the DSL works since quick fixes show the user how to solve issues in the program.

Example-driven code generation A much more intricate issue is finding a way to define the code generation process in an example-driven way. Using code generation, the user could translate the DSL into another programming language, such as JavaScript, Java, or Python. As an example, let us look at our cooking recipe language being translated into JavaScript by using code generation. We would write `add salt 5 G to bowl` which then would be translated into something like:

```
system.locateTool("bowl").add(
    system.localIngredient("salt"),
    system.createAmount(5, system.units.G)
);
```

Defining code generation in an example-driven, low-code/no-code way is a complicated task that needs further investigation.

API for interacting with existing models For the tool to be even more useful, we want it to be able to interact with other language workbenches. We envision implementing an API for interacting with existing models, such as languages and programs in those languages, and the ability to use them to create new ones.

Collaborative editing Furthermore, focusing on interaction, we want to work on collaborative editing. We envision an environment that allows for collaborative editing for language engineers, as well as end-users. We also envision support for version control systems, such as Git to store languages and programs defined in the tool.

Social media component In regards to user-friendliness and the low-code/no-code aspect of our tool, we could imagine having a social media component, where users could post implementations for others to see and use, where the users could give feedback on the implementation by voting and leaving comments.

Bibliography

[1] Quickly create DSLs with Langium. `https://tomassetti.me/quickly-create-dsls-with-langium/`. Accessed: 2022-05-29.

[2] Miksilo: A modularity first language construction workbench. `https://keyboarddrummer.github.io/Miksilo/`. Accessed: 2022-05-26.

[3] Kacper Bak, Dina Zayan, Krzysztof Czarnecki, Michał Antkiewicz, Zinovy Diskin, Andrzej Wasowski, and Derek Rayside. Example-driven modeling: model= abstractions+ examples. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1273–1276. IEEE, 2013.

[4] Mikhail Barash. Example-driven software language engineering. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, pages 246–252, 2020.

[5] Mikhail Barash. Vision: the next 700 language workbenches. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*, pages 16–21, 2021.

[6] Mikhail Barash and Václav Pech. Teaching MPS: Experiences from industry and academia. In *Domain-Specific Languages in Practice*, pages 293–313. Springer, 2021.

[7] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd, 2016.

[8] Neil CC Brown, Amjad Altadmri, and Michael Kölling. Frame-based editing: Combining the best of blocks and text programming. In *2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*, pages 47–53. IEEE, 2016.

[9] Jordi Cabot. Positioning of the low-code movement within the field of model-driven engineering. pages 1–3, 10 2020. doi: 10.1145/3417990.3420210.

[10] Fabien Campagne. *The MPS language workbench: volume I*, volume 1. Fabien Campagne, 2014.

[11] Hyun Cho, Yu Sun, Jeff Gray, and Jules White. Key challenges for modeling language creation by demonstration. In *ICSE 2011 Workshop on Flexible Modeling Tools, Honolulu HI*, 2011.

[12] Sergey Dmitriev. Language oriented programming: The next paradigm.

[13] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.

[14] Moritz Eysholdt and Johannes Rupprecht. Migrating a large modeling environment from xml/uml to xtext/gmf. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 97–104, 2010.

[15] Martin Fowler. *Domain-specific languages.* Pearson Education, 2010.

[16] Thibault Gagnaux. Developing a minimal language server for the frege programming language: an experience report.

[17] Desirée Groeneveld, Bedir Tekinerdogan, Vahid Garousi, and Cagatay Catal. A domain-specific language framework for farm management information systems in precision agriculture. *Precision Agriculture*, 22(4):1067–1106, 2021.

[18] Lennart CL Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and ides. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 444–463, 2010.

[19] Steven Kelly, Kalle Lyytinen, Matti Rossi, and Juha Pekka Tolvanen. Metaedit+ at the age of 20. In *Seminal contributions to information systems engineering*, pages 131–137. Springer, 2013.

[20] Amy J Ko and Brad A Myers. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 387–396, 2006.

[21] Björn Latte, Sören Henning, and Maik Wojcieszak. Clean code: On the use of practices and tools to produce maintainable code for long-living. 2019.

[22] Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann. Evolution of novice programming environments: The structure editors of carnegie mellon university. *Interactive Learning Environments*, 4(2):140–158, 1994.

[23] Peter D Mosses. A component-based formal language workbench. *arXiv preprint arXiv:1912.10631*, 2019.

[24] Betsy Pepels and Gert Veldhuijzen van Zanten. Model driven software engineering in the large: Experiences at the dutch tax and customs service (industry talk). In *Proceedings of the 1st Industry Track on Software Language Engineering*, pages 2–2. 2016.

[25] Santiago Perez De Rosso, Daniel Jackson, Maryam Archie, Czarina Lao, and Barry A McNamara III. Declarative assembly of web applications from predefined concepts. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 79–93, 2019.

[26] Daniel Ratiu, Vaclav Pech, and Kolja Dummann. Experiences with teaching mps in industry: Towards bringing domain specific languages closer to practitioners. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 83–92. IEEE, 2017.

[27] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 132–141, 2011.

[28] Markus Voelter, Bernd Kolb, Klaus Birken, Federico Tomassetti, Patrick Alff, Laurent Wiart, Andreas Wortmann, and Arne Nordmann. Using language workbenches and domain-specific languages for safety-critical software development. *Software & Systems Modeling*, 18(4):2507–2530, 2019.

Appendix A

Xtext grammar rules for cooking recipe language

Listing A.1: All Xtext grammar rules for the cooking recipe language in our case study.

```

Model:
    (ingredients+=Ingredient | tools+=Tool | steps+=Steps)*;

NewStep:
    StepAddIngredientAllToTool |
    StepAddIngredientAmountToTool |
    StepAddFromToolAllToTool |
    StepSetTemperatureToolToAmountDegree |
    StepWaitToolForAmountTime |
    StepMixTool;

Steps:
    'steps:' steps+=NewStep+;

Ingredient:
    'ingredient' name=ID amount=INT unit=MetricUnit;

MetricUnit:
    unit=('ML'|'G'|'KG'|'L');

Tool:
    'tool' name=ID;

StepAddIngredientAllToTool:
    'add' ingredient=[Ingredient] 'all' 'to' tool=[Tool];

StepAddIngredientAmountToTool:
    'add' ingredient=[Ingredient] amount=INT unit=MetricUnit 'to'
       ↪ tool=[Tool];

StepAddFromToolAllToTool:
    'add' fromTool=[Tool] 'all' 'to' toTool=[Tool];

StepStepSetTemperatureToolToAmountDegree:
    'setTemperature' tool=[Tool] 'to' amount=INT
       ↪ tempSystem=TemperatureSystem;

StepWaitToolForAmountTime:
    'wait' tool=[Tool] 'for' amount=INT time=TimeUnit;

StepMixTool:
    'mix' tool=[Tool];

TemperatureSystem:
    system=('CELSIUS'|'FAHRENHEIT');
```

```

TimeUnit:
    time=('SECONDS'|'MINUTES'|'HOURS');

terminal INT returns number: /-?[0-9]+/;
```