JIRAE, Vol. 4, No. 1, April 2019, 26-32
e-ISSN 2407-7259
DOI: 10.9744/jirae.4.1.26-32

Creating Domain-Specific Language and Syntax Checker Using Xtext

Billy Jonathan[1,2,a], Rafayel Avetyan[2,b], Stan Abeln[3,c]
[1]Department of Informatics, Petra Christian University, Surabaya, Indonesia
[2]Information and Communication Technology, Fontys University of Applied Sciences, Eindhoven, The Netherlands
[3]SW DE SICS Scanner Facilities, ASML Holding, N.V., Veldhoven, The Netherlands
[a]billy.jonathan97@gmail.com, [b]r.avetyan@fontys.nl, [c]stan.abeln@asml.com

Abstract. ASML is a company that manufactures the TWINSCAN machine that can produce semiconductor chips. This machine has a TWINSCAN software installed inside it and that software needs mapping configuration files to keep it running properly. The configuration files are developed by developers from many departments within ASML. However, the development process of the configuration files is ineffective, as the developers will know if there is any defect in the source code only in the late part of the development after all files have been committed into the TWINSCAN software's source code archive. It would be better if the developers know if there is any invalid syntax in the configuration files when they are still creating or editing the files, so they can fix the defects immediately before the files are uploaded into the source code archive. The main purpose of this research is to develop a Domain-Specific Language (DSL) based on the structure of the configuration files, and a syntax checker application that can check the mapping configuration files for the TWINSCAN machine during the creation or editing phase of those files by the developers. The development of the DSL and syntax checker would be done using Xtext framework installed in Eclipse Integrated Development Environment (IDE). The final results show that the DSL and the syntax checker developed using Xtext can detect any invalid syntax during the development phase of the configuration files, so the developers can fix the defects directly, thus solving the company's problem.

Keywords: Domain-Specific Language, Syntax Checker, Parser, Xtext, Eclipse.

1. Introduction

ASML Holding N.V. (ASML) is currently the largest supplier of lithography systems for semiconductor industries in the world. It is the creator and manufacturer of the TWINSCAN machine, which can maximize the productivity of the lithography systems as well as its accuracy. In ASML's TWINSCAN machine, there is a software that drives the machine, and the software needs configuration files to give commands to the hardware inside the machine.

Figure 1. ASML TWINSCAN machine

However, because the configuration files are developed by many developers, and the files have to be compiled, installed, and tested first, any error in the configuration files are not reported until they are committed into the TWINSCAN software's source code archive and the TWINSCAN software is started, so the errors are detected late. This kind of workflow is not effective and time-consuming. The developers prefer to check the configuration files while they are being created or edited in the Eclipse IDE instead of checking it later when the files are already in the source code archive and the TWINSCAN software starts.

To solve the problem, first, a domain-specific language (DSL) based on the configuration files' format has to be made as the formal language definition for the configuration files' syntax. Secondly, a syntax checker that corresponds to the DSL has to be developed for the parser so that it can check for any syntax error when the files are still being created or edited. The DSL and the syntax checker can be created using Xtext framework.

2. Research Overview

2.1 Initial Situation and Problem

The TWINSCAN software's configuration files are developed by many developers in ASML using Eclipse IDE. However, whenever the developers develop the configuration files, the parser cannot detect if there is an invalid syntax in the code because Eclipse IDE cannot recognize the format of the configuration files. Usually, they know that there is any defect after they commit the configuration files into the TWINSCAN software's source code archive installed into the software. When an error is found, the code has to be edited and committed again into the source code archive and they have to start the software again. This kind of workflow is not effective as it is time-consuming to start the software and find some errors in the code, then repeat it again and again. It will be much better if the parser can detect and report any syntax error during the creation or editing of the configuration files, so the developers can find out the error and fix it immediately.

2.2 Research Description and Objectives

The aim of this research is to create a Domain-Specific Language (DSL) that will become the formal language definition for the configuration files and to develop a syntax checker application that can be used during the editing phase of the files so that there will not be any more syntax error during the testing phase. The syntax checker will be deployed into two forms, first as an Eclipse plugin that can be installed in Eclipse IDE, and second, as a standalone command-line tool that can be run from a console or a terminal. The DSL and the syntax checker application will be created using Xtext.

In order to support the DSL and syntax checker development process, scrum is used as the development methodology. Scrum is a fast and flexible approach for software development. It is easy to implement this methodology in a team and it has low risk since everyone is aware of any new updates. In scrum, the development process is split into sprints and in ASML, each sprint consists of 10 working days. It is the department decision in ASML to use sprints of two weeks, so it allows each team in the department to synchronize their activities.

2.3 Tools and Programming Language

The following tools and programming language that will be used for the development process:

1. Xtext framework is used for the DSL and syntax checker development
2. Xtend programming language is used to develop algorithms for the syntax checking process.
3. Eclipse IDE is used as the development environment.
4. BitBucket/Git is used as the version control system.
5. Jira software is used as the communication board in scrum implementation.
6. Collaborators is used for the Code Review process.

3. Research Results

3.1 Comparison of Parser Generators

There are many parser generator applications and frameworks beside Xtext are available. However, Xtext was recommended at first for this research because it is Context-Free Language-based, which is the grammar used to design the DSL grammar, it can run on Java machine, which can be integrated easily with Eclipse IDE, it has an IDE that would make the DSL development easier instead of using console, and it has Extended Backus-Naur Form (EBNF) as the basic notation form, which is a popular and common grammar notation used by many language developers.

There are three other parser generators that meet those criteria, they are ANTLR, Beaver, and JavaCC. The comparison between those three parser generators and Xtext can be seen in Table 1. The table compares if the features needed in this research are available or not in the parser generators. "YES" means that the feature is available and "NO" means that the feature is not available and requires a third-party program. Based on the research on other syntax checkers for other types of configuration file in the company, there are 5 basic features that are needed for the syntax checker development in this research, these features are needed to develop the syntax checker in the easiest way and it would be easy to be developed further by other developers later. The necessary features are:

- Parser: The main feature used to identify any patterns of the syntax and keywords to build the AST.
- Generator: Generates the syntax of the DSL.
- Lexer: Analyzing the source code to split the syntax into keywords or tokens based on the DSL rules and components.
- Linker: Provides "cross-reference" feature that enables a component in a rule has a reference to another rule in the DSL.
- Continuous Integration (CI): This would be used to store the final products so that they can be taken care of easily by the developers later.

Table 1. Comparison of parser generators

	ANTLR	Beaver	JavaCC	Xtext
Parser	YES	YES	YES	YES
Generator	YES	YES	YES	YES
Lexer	YES	NO	YES	YES
Linker	NO	NO	NO	YES
CI	NO	NO	NO	YES

As can be seen in Table 1, Xtext has the parser, generator, lexer, linker, and CI support without the needs of a third-party program. This would make the DSL & syntax checker development become more efficient as only Xtext is needed to be installed. Based on the comparison, Xtext had been decided as the tool would be used to develop the DSL & syntax checker.

3.2 Installing Xtext and Doing Experiments

Xtext is a framework that depends on Eclipse IDE. It should be downloaded from Eclipse website and installed as a plugin on Eclipse IDE. After the installation, a new Xtext project has to be created and the file extension for the language has to be specified, then Xtext will generate a default starting grammar in the text editor. An experiment was done by creating a dummy DSL and saw the result whether it worked or not. Errors in the syntax were also created, so that the way Xtext generates the error alert can be viewed. Figure 2 shows the default Xtext grammar code.

Figure 2. Default Xtext grammar code

3.3 Configuration Files' Structure Analysis

All versions of configuration files were downloaded from the source code archive and analyzed. There are 3 main

versions of the files based on the major difference. Some important components in the configuration files that exist in all versions were found, like the header keyword that must exist in every configuration file and the function name or interface name as the ASML developers called. These components are grouped based on their role (e.g. as a keyword, function name, etc.) and the basic characters like a semicolon or brackets are also grouped together as terminals. The major difference from one version to another is the existence of new lines and white spaces between some components, but the basic code structure still similar. All these components and specific rules are the results of the analysis and they were noted to be used in the next step. However, because of CONFIDENTIALITY ISSUE, the detail structure of the configuration files cannot be mentioned in here.

3.4 The Design of the DSL Grammar

Since the configuration files had been analyzed and the result was put in a note, the grammar design could be created. Based on a recommendation from an ASML employee, a research about Context-Free Grammar (CFG) was done. It is a type of grammar that is used widely by language developer to convert the rules to a basic grammar for programming language. Eventually, CFG was decided as the type of grammar that would be used for the design.

CFG has many notations and forms. A popular notation in CFG that is often used is the Backus-Naur Form (BNF) notation. BNF notation is the basic form of the CFG, it is easier to create a grammar from scratch in BNF notation than in any other form, so it was decided to design the grammar in BNF format. The grammar was designed based on the analysis note created previously. All basic characters were put into terminals and the rules are put into variables. It was tricky to design the grammar because there are different versions of configuration files, but with help from fellow ASML colleagues, the grammar design could be finished.

After the grammar had been designed in BNF format, the grammar was converted to a variant of BNF notation which is called EBNF notation. The reason for this conversion is because Xtext uses EBNF as its grammar format, so it is easier to convert the grammar from EBNF to Xtext grammar than from BNF to Xtext grammar. After some research on a language developer's website [9], it was possible to convert the grammar from BNF into EBNF notation. However, the first version of the grammar still had many flaws, like an ambiguous component and incorrect rules for expressions, so the grammar had to be revised by re-analyzing the configuration files and re-designing the BNF grammar. Figure 3 shows the grammar in EBNF format.

The grammar in EBNF notation had to be converted into Xtext grammar. Since Xtext grammar based on EBNF notation, there is only a minor difference between both notations, so it was easy to do. It was decided to call this DSL as "VPM DSL", this name was taken from the extension of the configuration files.

A research was done based on a research journal [11] to convert the grammar. Because of the minor difference in between the EBNF and Xtext grammar, there were some rules that had to be changed again and the EBNF grammar had to be extended to make it possible to implement in Xtext. This included separating the EBNF grammar into 3 types of rule, these 3 rules are the rules category defined in Xtext:

- Parser rule: The rule for the variables or non-terminals. This rule contains keywords or combination of terminals.
- Enum rule: The rule for enumerations. This rule contains the operators and signs for the expressions.
- Terminal rule: The rule for the terminals. This rule contains the basic characters that is impossible to derive further.

```
start = {mapping};
mapping = section-header, section;
section-header = "mapping", EVP-interface-symbol, [condition];
condition = "[", or-expression, "]";
or-expression = and-expression | or-expression, "||", and-expression;
primary-expression = "(", or-expression, ")" | "!", primary-expression | comparison;
and-expression = primary-expression | and-expression, "&&", or-expression;
comparison = PVP-identifier, ("==" | "!="), PVP-variant-symbol;
PVP-identifier = PVP-interface-symbol, ":", PVP-name-symbol;
section = "{", {EVP-assignment}, "}";
EVP-assignment = EVP-identifier, "=", EVP-variant-symbol, [condition], ";";
EVP-identifier = EVP-interface-symbol, ":", EVP-name-symbol;
```

Figure 3. The DSL grammar in EBNF notation

The reason for this rule separation is because it made the grammar cleaner and easily showed the role of each component and this also made it easier when writing the rules in Xtext project later.

3.5 Syntax Checker's General Workflow

Syntax checker depends on the rules and components created in a programming language. Therefore, in the case of a DSL, it has to be created first before the syntax checker creation. A syntax checker usually already installed in a programming text editor. When the user types something in the text editor, the syntax checker will automatically start checking what the user typed by matching the what the user types with the DSL. The general workflow of a syntax checker application can be seen in Figure 4.

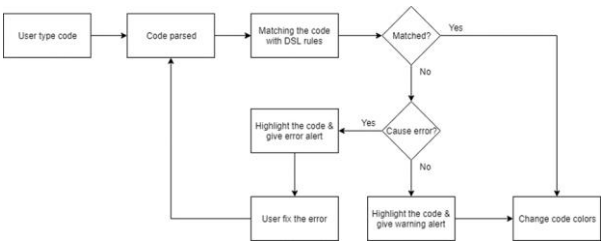

Figure 4. The general workflow of a syntax checker

3.6 The Design of the Architecture View

After the general workflow of a syntax checker has been found, the architecture view could be designed. The system in

ASML environment was analyzed, it connects Eclipse IDE to its plugin, the source code archive, and its location in the shared environment in ASML network. This was done to design a possible architecture view. An architecture view of the syntax checker that can be applied to ASML environment where the developers will develop the configuration files has been designed. The design of the architecture view can be seen in Figure 5.

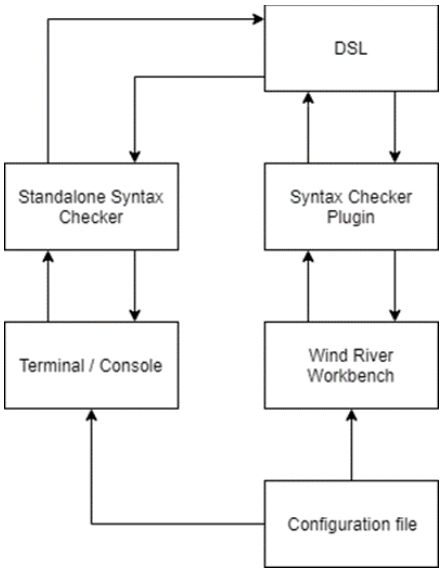

Figure 5. Architecture view of the syntax checker

4. Implementation and Test Results

4.1 DSL Implementation

A new Xtext project in Eclipse IDE was created specifically for this research, then the default Xtext grammar was replaced with the converted VPM DSL grammar created before. In this Xtext project, there is a grammar called "Terminals grammar" stored in Eclipse server and accessed by Eclipse IDE through Eclipse API. This Terminals grammar already provides default terminals including comments, white spaces, and new lines for the existing VPM DSL grammar, so the basic terminals and rules again for the VPM DSL did not have to be created.

Since the new Xtext project had been created and ready to be used, the converted grammar code was written. The grammar was written in the main Xtext file in the Xtext project. After the implementation finished, the grammar had to be tested.

4.2 DSL Unit Test

Before testing the DSL, a test scenario document was created to list all kinds of tests for the DSL and for the syntax checker later. Xtext provides an Xtend file that had been integrated with JUnit library dedicated for unit testing. For the unit test, the test cases that were related to the DSL were used. All 26 tests cases for the DSL were written in the Xtend file and Junit was run. There were several tests that did not pass at first. From 26 test cases, there are 15 test cases that did not pass, at the second test, there are 7 test cases that did not pass, and finally, in the third test, all test cases passed. After all test passed the Xtext project was compiled and tested manually with the real configuration files.

4.3 DSL Test with Real Files

After all unit tests were passed, real configuration files were used to test the grammar. Different versions of the real configuration files were used to check if the DSL could match all versions and it worked. If the code did not match the DSL, an error would occur, prompting the user to fix it. This error was generated automatically by Xtext because the typed code from the user did not match the DSL rules. For every version of the configuration file, the DSL detected the defects correctly. Since the VPM DSL passed all unit tests and tests with the real configuration files, the development of the DSL has finished and it becomes the formal language definition for the configuration files and the implementation of the syntax checker can be started. Syntax Coloring Implementation

The developed syntax was decided to be called "VPM Checker". The first feature of the syntax checker that would be created is the syntax coloring. It is needed to give each syntax a color based on its role in the VPM DSL. For instance, green color for syntax that has a role as a comment or purple color for syntax that has a role as a keyword. This will make the developers would know if the code they typed has the correct role as they expected or not and they could recognize the code easier when they are debugging or error checking.

To be able to create this feature, Xtend programming language is used. Xtend is a programming language specially used in Xtext which is interoperable with Java programming language. It has a syntax structure that is similar to Java with only some minor difference, but it is still understandable. Xtend will also be used for the other features of the syntax checker.

In order to configure the syntax colors, there are 3 Xtend files that had to be created. The first file is for the color and style configuration of each syntax role, the second file is for setting the color and style for each syntax using lexical or syntactic analysis, and the third file is similar with the second file, but it uses semantic analysis to determine the role of each syntax. The lexical analysis is used because it is the basic and easiest way to determine the role of each syntax by only checking its rule and position within the DSL. However, the semantic analysis is also necessary because there are some syntax roles that could not be determined by lexical analysis because the syntax is a combination of some roles with a unique meaning, so they require the parser to iterate through the AST in order to find their specific meaning.

The second file and the third file correspond to the first file in order to find out what color to give to each syntax. Later, these files would be integrated into Xtext so that the color and style will appear when the developers open a .vpm file in Eclipse IDE.

4.4 Syntax Validation Implementation

The next feature is syntax validation. This feature is extremely important because it would be used to check if the syntax is valid or not, if it is invalid, they would check the type

of error or warning in Eclipse IDE, then they would show it to the user.

The algorithms for checking the syntax validity were developed by checking the syntax roles against the DSL structure. When implementing the algorithms, if an invalid syntax was found, a function to show the error or warning alerts was called and Xtext would automatically generate a highlight in the form of a red line under the invalid syntax. This highlight along with the error or warning alert would appear automatically in Eclipse IDE when the developers were typing in the text editor whenever there is an invalid syntax. After implementing the algorithms correctly in the Xtend file, the algorithms could recognize the invalid syntax and show the highlight and the correct error or warning alert.

4.5 Auto-Correct Implementation

The auto-correct feature or "quickfix" as called in Xtext, provides fix suggestions to the user when there is an error or warning occurred in order to resolve it. The suggestion given by quickfix is placed under the error or warning alert and if the user clicks it, it will automatically implement the fix that would resolve the error or warning. An error or warning created in the syntax validation can have one or more quickfixes, but not all of them need it, depending on the circumstances.

To implement the quickfix feature, algorithms for each quickfix were created. These algorithms would enable the suggestions to appear below the error or warning alert in Eclipse IDE and allows the fix for that error or warning to be implemented automatically. For example, for an error where an interface name is not in capital letters, the quickfix for this error will provide suggestion to the user to change the interface name letters to uppercase, and when the user clicks this suggestion, the quickfix will automatically capitalize the interface name letters. This feature will make it easier for the developers to fix an error or warning as it provides an automatic fix, instead of fixing the defect manually.

The algorithms were created by analyzing the source code structure and find the possible outcome, and then wrote them in the Xtend source code. Next, a debugging was done to test the quickfix, this was done until the prefered algorithms were found. After the algorithms for the quickfixes had been implemented, whenever the user types an invalid syntax, an error or warning alert would show up, and if the error or warning had a quickfix implemented, the fix suggestion would show below the alert and can be used by the user.

4.6 Syntax Validation Unit Test

Since all features had been implemented, the syntax checker was ready to be tested. However, the test was only able to be done for the syntax validation feature, which is the mandatory feature from the company, due to lack of time. Unit test was used to test the syntax validation. Using JUnit, and the test scenarios related to the syntax validation that had been created previously, the unit test for the syntax validation was done. There are 15 unit test cases for the syntax validation. At first 6 test cases were failing, so the validation code was revised. At the second time, all test cases were passed.

4.7 Syntax Checker Test with Real Files

Since the syntax validation had passed all the unit test cases, it was decided to use a real configuration file to test the DSL and syntax checker. the real configuration file source code was copied to an Eclipse instance generated by Xtext, since the syntax checker had not been deployed yet. In this example, a syntax error was made intentionally.

As seen in Figure 10, the syntax coloring worked. It gave different colors to specific syntax based on their roles. The syntax validation also worked. It determined the error correctly and show the error alert along with the syntax highlighting (the red line below the invalid syntax). Below the error alert, two quickfixes were available for the user to resolve the error. When one of those quickfixes was clicked, the assignment without condition would move after the assignments with conditions and the error was gone. This means that the syntax checker works correctly according to the DSL.

4.8 Component Test

After all unit tests were done, a component test was done to make sure that the four syntax checker components, which are the DSL matching, the syntax coloring, the syntax validation, and the quickfix feature, do their job as expected. The component test was done by creating a new configuration file and then by typing the syntax to test its features one by one. For example, the syntax validation component was test by typing invalid syntax to see if the correct error alert would appear or not. When there was a mistake, the algorithms were checked again along with the unit test and then revised.

4.9 Code Review

Since the DSL and syntax checker passed the component test, a code review was done for all Xtext and Xtend source code using a tool called Collaborator. This tool is the default code review tool used in ASML. The source code files were uploaded and reviewed through Collaborator by two other developers in ASML and finally, they gave feedback to revised the source code, then the source code were changed based on that feedback. After all code passed the code review, the next step was the user test.

4.10 User Test

The user test is an important part of this assignment as feedback are needed from the developers who will use this syntax checker application later. The developers who understand about the configuration files from another department were contacted to do the user test. They tried to create some configuration files in Eclipse IDE using the syntax checker application.

The result of the user test was very good. The developers were satisfied with the syntax checker application and they also gave recommendation for further improvement. After that, the syntax checker could be deployed.

4.11 Eclipse Plugin Deployment

First, the syntax checker should be deployed as an Eclipse plugin so that the developers from the other departments can use it in Eclipse IDE. However, the deployment of the plugin has to be done by another team within ASML. The person in charge from that team was contacted and he agreed to do it. After the deployment finished, he installed in Eclipse IDE so that it can be used by ASML developers.

4.12 Standalone Command-Line Tool Deployment

Unlike the deployment of the Eclipse plugin in the previous sub-chapter, which has to be handled by another team, the deployment of the standalone command-line tool can be done without their help. Xtext allows the developed syntax checker to be exported as a standalone command-line tool in the form of a jar file. The compiled jar file was exported and could be used directly. However, using the jar file directly would be inconvenience for the developers, as it could not have multiple files checked together in one command and it also has an unclear message if there is no error in the file. Because of that, a bash file was created so that it would run the jar file with a nice customize messages shown to the developers when they wanted to run the jar file.

Multiple files checking could also be done directly through one command. The command to run the bash file is also shorter than the command to run the jar file directly. In this way, the developers only need to run the bash file when they want to use the standalone command-line tool. In the end, the bash file was created successfully. A command option about the usage instruction is also written in the bash file so that the developers can see how to use the standalone command-line tool. Next, both forms of the deployed syntax checker were tested by creating new configuration files.

4.13 Deployed Syntax Checker Test

Next, some new configuration files were created in Eclipse IDE with the Eclipse syntax checker plugin installed. Some errors were made intentionally to check if the syntax checker plugin worked correctly or not. The result is good, it can check all the defects immediately when the syntax was typed.

The newly-created configurations files with some defects were saved and then he standalone command-line tool was run to check the configuration files. This would show if the standalone command-line tool worked correctly or not. The result is the same with the Eclipse syntax plugin, proving that it worked correctly.

4.14 Checking Process in the Source Code Archive

For the last step, the new configurations files were committed into the source code archive, then they were checked over there using a tool called CWBD plugin, which is a standalone command-line tool developed to check the uploaded .vpm files. It is usually used by the developers to check the configuration files in the late part of the development. CWBD plugin is much more advanced than the syntax checker developed in this research, as it not only checks the syntactic part of the configuration files, but also the semantic part while the syntax checker in this research can only check the syntactic part. Figure 6 shows the checking process of the configuration files syntax using CWBD along with the checking result.

Figure 6. The checking process using CWBD plugin

In Figure 6, there is an error: "VPME file is not a DATTARGET" generated by the CWBD plugin. DATTARGET usually generates and lists all configuration files for the TWINSCAN machine. This error occurs because the new files that were created and uploaded were not listed yet in the DATTARGET, but this error is not related to the syntax of the configuration files, so this is not a syntactic error. As can be seen, there is no other error beside the DATTARGET error, so this means that the syntax checker application in this research worked as the checking process is successful.

5. Conclusion and Recommendation

During the research and implementation process, the DSL and syntax checker application worked as expected. The DSL has become the formal language definition of the configuration files. The deployed Eclipse syntax checker plugin can detect any invalid syntax in the configuration files when the user create or edit the files. Also, the deployed standalone syntax checker command-line tool can check multiple files directly through the console or terminal.

After the configuration files were checked using the deployed syntax checker and committed into the source code archive, there was no more syntactically invalid syntax in the files as all syntactic error had been detected and fixed using the deployed syntax checker. This means that the company's problem has been solved by the DSL and syntax checker created in this research.

For further research, the DSL grammar can still be extended for supporting more advance syntax checking process. Also, a semantic check is also a good idea to be added, as the syntax checker in this research currently only able to do the syntactic check. Xtext is also a very good and powerful framework for language development and it can be used again in the future.

References

1. ASML: About ASML - Organization. (2018). Retrieved from https://www.asml.com/company/organization/en/s277?rid=51984
2. ASML: About ASML - Our history. (2018). Retrieved from https://www.asml.com/company/our-history/en/s277?rid=51985
3. Beaver - a LALR Parser Generator. Retrieved from http://beaver.sourceforge.net/index.html
4. Bettini, L. (2016). Implementing Domain-Specific Languages with Xtext and Xtend - Second Edition (2nd ed.). Packt Publishing.

5. Efftinge, S., & Spoenemann, M. Xtext - Language Engineering Made Easy!. Retrieved from https://www.eclipse.org/Xtext/
6. Efftinge, S., & Spoenemann, M. Xtend - Modernized Java. Retrieved from https://www.eclipse.org/xtend/
7. Electroiq.com. (2018). ASML increases its dominance of semiconductor lithography market in 2017 | Solid State Technology. [online] Available at: https://electroiq.com/2018/02/asml-increases-its-dominance-of-semiconductor-lithography-market-in-2017/ [Accessed 5 Jul. 2019].
8. JavaCC. (2019). Retrieved from https://en.wikipedia.org/wiki/JavaCC
9. Lee, X. (2018). What's the Difference Between BNF, EBNF, ABNF?. Retrieved from http://xahlee.info/parser/bnf_ebnf_abnf.html
10. Suranga, S. (2018). Build your own programming language with ANTLR. Retrieved from https://medium.com/@shalithasuranga/build-your-own-programming-language-with-antlr-5201955537a5
11. Yue, J. (2014). Transition from EBNF to Xtext.