TU WIEN Informatics

# A Langium-based approach to BigER

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

**Bachelor of Science**

in

**Business Informatics**

by

**Tobias Jordan, Sebastian Zib**
Registration Number 11902127, 11907072

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dr. Dominik Bork
Assistance: Philipp Glaser, BSc

Vienna, 7th February, 2024

| Tobias Jordan, Sebastian Zib | Dominik Bork |
|---|---|

# Erklärung zur Verfassung der Arbeit

Tobias Jordan, Sebastian Zib

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 7. Februar 2024

Tobias Jordan, Sebastian Zib

# Acknowledgements

We would like to express our deepest gratitude to our advisor, Assistant Prof. Dr. Dominik Bork, for making this project possible by helping us on every step of the way: From recommending the topic over pointing us in the direction of related projects to giving feedback to our work.

Furthermore, we are indebted to Philipp-Lorenz Glaser, BSc, whom we could count on for support with technical problems and answers to questions concerning the BIGER extension.

Lastly, we would like to thank our families for their continued support during the development of the project.

# Abstract

Although the Language Server Protocol (LSP) enables the division of language smarts and user interaction, a lot of language engineering tools do not use its full potential. Among them is the BIGER modeling tool, which uses technologies and programming languages not well suited to the client-server structure. This problem can be overcome by updating its foundation to a framework that was made with the web-stack in mind. This thesis therefore presents a reworked version of the BIGER tool, which is no longer based on Xtext, but instead on Langium. For this, features are translated from Java to TypeScript, including the code-generation, validation and graphical representation. The intention is to gain an insight into whether moving the project onto this new base is viable. During the practical work, there are some difficulties associated with lack of features as well as documentation for both Langium and Sprotty, which is due to them being new technologies. Despite these issues, the results overall suggest that this move can be achieved with further work.

# Contents

CHAPTER 1

# Introduction

Many of the current approaches to language engineering, like the Eclipse Modeling Framework[SBMP08], are fairly heavyweight in terms of disk-space and memory used. This is due to the fact that they are designed to be used locally on each machine and must therefore be standalone products, which naturally shifts focus from lightweight applications to better integration. While this means they can offer an unparalleled depth of features, it also means that the number of supported languages or meta-languages has to remain low, as language smarts have to be implemented for each and every one of them. Not only is this partly responsible for the aforementioned bloating of the software, but it also means developers of different frameworks working on support for the same language are implementing the same features over and over again without being able to reuse what has already been done [BK20].

This problem can be solved by strictly dividing editor and language features with a standardised interface between them [REIWC18]. However, even those approaches intended to use such a server-client structure, including the subject of this thesis, the BIGER Modeling Tool, primarily use technologies not made for, but rather fitted to the use of language servers. This introduces all kinds of avoidable constraints, like the inclusion of memory-intensive programming languages like Java, which is running in its own Java Virtual Machine.

In this sense, the aim this thesis is to overcome these problems by reworking the current BIGER tool to use Langium[1] as its technological foundation, a technology created with distributed systems in mind, replacing Xtext. While Xtext is a well-established language engineering framework with more than 15 years of incremental development behind it, the fact that it was not designed for a client-server system from the start becomes apparent when considering the programming languages used. The current BIGER tool is

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

dominated by Java and Xtend, the latter of which is a Java dialect, meaning it is still compiled to Java at runtime [Foub].

Naturally, it follows that one of the goals we want achieve by transitioning to a new, Langium-based framework is to replace Java with TypeScript. As a scripting language compiled to JavaScript at run-time, it is more lightweight, faster and, as we will see during the practical implementation, adds flexibility. This makes it an overall better fit for a language server.

At the same time, it is important to maintain the modularity of language services achieved by the original bigER. The reason behind that is to keep the code readable and subsequently maintainable, but also to efficiently use the Language Server Protocol, which connects client and server through standardized categories of requests[BL23]. In this, we are aided by the numerous LSP-features pre-defined in Langium as well as the ease of declaring and registering new ones.

The overarching purpose of doing all this is to investigate whether it would be viable as well as worthwhile to bring the whole bigER project over to Langium. Answering this question involves explaining both the technologies we build upon and the work that has been done so far, before then detailing our part of translating and improving the current state. Therefore, this thesis starts by giving some theoretical background, which includes explaining the terms used in this Introduction, and continues on to discuss papers related to the creation of bigER or similar projects. The main chapter will then go into detail about how each component functions, what changes we have made, what peculiarities we encountered and how they affect the extension. Lastly, the thesis will present an answer to the central question and the nuances behind it as well as offer some points of interest that may be used as guideposts for future research.

CHAPTER 2

# Background

## 2.1 Xtext

Xtext is a language engineering framework published in 2006 [EV06] and at present constitutes the base of the BIGER project. The main IDE XText was developed for is Eclipse, since it is still being maintained by the Eclipse Foundation, although there are also extensions available for other environments like VSCode. While there are still regular updates being released, they mainly bring Xtext up to date with newer versions of dependencies and there have been few further developments of the framework in the last couple of years. The reason for this given in the release notes is the declining number of contributing users [Foua]. On the bright side that also means that since major features have been around for a while with only small changes made to them, they are thoroughly documented and there is a lot of community material, such as tutorials, example projects and articles, as well.

## 2.2 Language Server Protocol

The Language Server Protocol (LSP) is a protocol used for communication between development tools (such as VSCode or Eclipse) and language servers. LSP is developed and maintained by Microsoft. The protocol operates using a JSON-RPC. It supports a wide range of features to enhance the usage of development tools. These features range from validation checks to quality of live features such as refactoring[Corb].

The development tool sends a request (for instance, for validation of a piece of code) to the language server. The language server then processes this request and sends a reply to the development tool, which can act accordingly (in this example: notify the user about the error). The advantage of language servers in conjunction with LSP is that the same server can be used by multiple tools, provided both the development tool and the language server are capable of LSP.

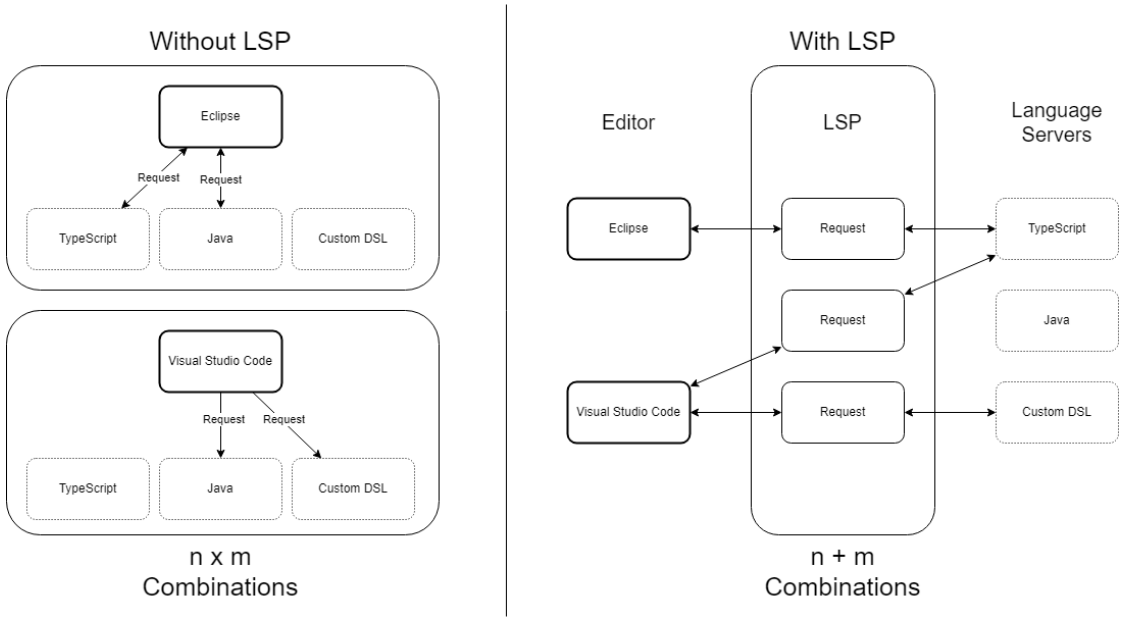


Figure 2.1: Usage Language Server Protocol

Without LSP each combination of editing software and programming language has to have an individual language server. Meaning that, by default, a language server will only understand one language for one software. With LSP each editor and each language server needs to understand LSP. However once an editing tool fulfils this requirement, it can make use of each language server that also supports LSP and vice versa.

For the development of the bigER Langium Tool, LSP will provide the communication between VSCode and the language server based on Langium.

## 2.3 Langium

Langium is an open source meta modelling tool. Similarly to Xtext it can be used to create Domain-Specific Language (DSL). The framework is written in the TypeScript programming language [Gmbc]. The grammar language, also called Langium, is recursively-defined in Langium [Gmbd]. For readability purposes, we will refer to the grammar language as Langium grammar language in this thesis to avoid confusion with the Langium tool used to generate language servers. The tool was first released to the public in August 2022 by TypeFox GmbH [Gmbb]. It should be noted that Langium is still in active development and for the practical part release version 1.1.0 was used. The source code for Langium is open and can be found on GitHub [Gmbb].

Langium was chosen to eliminate the need for Java dependencies in the bigER Visual Studio Code extension [1]. Langium uses TypeScript, which incorporates well with VSCode.

[1] https://marketplace.visualstudio.com/items?itemName=BIGModelingTools.erdiagram

Langium generated language servers support communication through the Language Server Protocol. The LSP functionality is partially predefined. This includes the rename and document symbol features. Further LSP features can be implemented using TypeScript and the abstract syntax tree generated by the Langium grammar language.

This allows for a variety of customization options which can be employed depending on the desired DSL. For this thesis the following LSP features will be implemented:

- validation
- hover
- completion suggestions
- go-to
- rename
- document highlight

It should be noted in advance that due to the in-development status of Langium, some features of the framework are not fully implemented or may contain bugs. Furthermore, the documentation is very sparse and much of the information required for this thesis was either discovered through testing or GitHub discussion pages where the developers actively engage with issues from the community.

## 2.4 Sprotty

While Langium will be used to implement the language server which will be directly used for textual editing, Sprotty will be used for the graphical representation.

Sprotty is a web-visualization tool made by the Eclipse Foundation, released in 2017 [Fouf]. It was also used for the original BIGER Tool [GB21]. To work in conjunction with Langium and VSCode further integration code is required. This comes in the form of the "sprotty-vscode" package, which allows Sprotty to work inside VSCode windows.

Much like Langium, Sprotty is written in TypeScript and has strong native support for VSCode. It is a web-based visualization tool. It uses Scalable Vector Graphics which can be styled with CSS. Sprotty is open source and the code is available on GitHub [Fouc]. The main advantage of Sprotty is that it can be used as a visualization tool in a web environment. It works by binding input values to graphical models. These models are defined using HTML and rendered as SVGs. Sprotty is also able to make the generated graphical models interactive, allowing for features such as drag and drop or in-graph editing. Some of the interactive features are predefined, while others have to be expanded by developers who employ the tool.

Sprotty, by default, is used for web-applications, however through the additional sprotty-vscode glue code it is also able to run within VSCode webviews [Foud].

CHAPTER 3

# Related Work

This chapter will focus on discussing papers related to the original BIGER tool, as well as papers relating to Langium, Sprotty and the Language Server Protocol in general. The aim is to explore state-of-the-art approaches to problems faced when developing language servers, as well as to provide insight on the background of the original BIGER tool and how it differs to the implementation using Langium.

## 3.1 BigER Tool

The paper "The bigER Tool-Hybrid Textual and Graphical Modeling of Entity Relationships in VSCode" deals with the implementation and functionality of the original BIGER tool [GB21]. The original BIGER tool is also the basis of the Langium-based approach. Therefore, a lot of similarities can be seen between the functionalities of both tools. Both allow for textual editing and the creation of a visual representation based on the code. BIGER and BIGER Langium also both allow for code generation, however the original BIGER Tool only allowed for SQL generation. Later updates to the tool introduced the additional languages, which are also supported by BIGER Langium. The biggest difference between the original modeling tool and the Langium-based approach is that in-graph editing is not possible in the Langium version. This is due to the lack of documentation for the Langium and Sprotty integration, which could not be resolved otherwise during the development of the tool for this thesis.

Graphical editing allows for further accessibility and a broader usage. Meaning that also providing a WYSIWYG type editor for graphical modeling has major advantages over only being able to edit the model with the textual WYSIWYM editor [Fuh11, p. 7-8].

The architecture explained in the paper by Glaser and Bork is similar to the architecture in the Langium-based approach. Both use a client-server architecture, where the client communicates to the server via JSON in the form of the Language Server Protocol. The

client in this case is VSCode, whereas the server is a language server hosting the Entity Relationship language.

The implementation of the original bigER Tool uses Xtext for the realization of the language server. This creates an issue compared to Langium as Xtext requires Java to run the language server. Langium on the other hand requires only TypeScript. VSCode also requires TypeScript, but not Java, reducing the amount of dependencies to run the language server. In terms of LSP features, the paper by Glaser and Bork mentions that the original bigER Tool supports a variety of rich editing features [GB21]. The same is true for the Langium-based approach. A big difference between language servers implemented with Xtext compared to Langium is that both frameworks use a different parser and lexer. Xtext uses ANTLR while Langium uses Chevrotain [Gmbd]. The lexers use the grammar provided by the Xtext or Langium grammar language respectively and create an Abstract Syntax Tree (AST). The AST is then extended with the contents of files (.erd for bigER, .er for bigER Langium) by the parsers. How the lexers and parsers behave and which rules they follow dictate how the grammar has to be written. ANTLR uses the ALL(*) algorithm [PHF14], while Chevrotain by default uses A LL(k) algorithm (with a default of k=4)[Gau]. The amount of lookaheads for Chevrotain can be increased in the configuration file, however this comes at an extreme performance cost since the algorithm the lexer uses is not intended for higher lookahead amounts. Thus, the algorithm effectively dictates how many lookaheads the lexer can make. Lookaheads are necessary to distinguish between different options within EBNF grammar.

```
lookahead = "1" "2" "3" "4" "5" | "1" "2" "3" "4" "10";
```

In the case of the grammar given above, an ALL(*) lexer would be able to parse the grammar while a LL(4) lexer would not. This is because the LL(4) grammar would only take into account the first 4 terminals of each option before making a decision. An ALL(*) lexer on the other hand would consider all terminals.

This would be a big disadvantage for Langium as this would severely limit the grammars possible with the language. However TypeFox, the developers of Langium modified Chevrotain to also use the ALL(*) algorithm. This change was implemented in version 1.0 of Langium [Spo].

## 3.2 LSP functionality and evaluation

The paper "Decoupling language and editor - the impact of the language server protocol on textual domain-specific languages" by Hendrik Bünder aims to explore the functionality and possibilities of the language server protocol [Bün19].

At the time the paper by Bünder was written, Xtext was the only language modeling toolkit able to support the usage of LSP. Now, Langium is also capable of utilizing the

protocol. Furthermore, there are now over 40 [Cord] different editing tools supporting LSP and over 200 supported languages [Corc] according to the official LSP documentation, with some having different language servers for the same language to choose from. These lists contain almost every major language and tool, showing that the language server protocol certainly has had success with the functionality it provides. In comparison in 2019 when the paper by Bünder was published only about 50 languages supported LSP.

The paper points out that a big advantage of the usage of LSP over individual language support built into the editor is the avoidance of tool-lock in. Different tools have different advantages and disadvantages, and being able to work on the desired language in the desired tool improves the quality of live for developers. It allows developers to choose their configuration according to their needs and can better accommodate niche cases. The paper points out that the increased variety of supported tools is also a issue of accessibility [Bün19]. Seasoned IT-developers may be used to working with different development tools or in different IDEs. However casual users or users with a non-IT-background may have difficulties understanding tools they are not used to. This becomes more apparent when looking at the context of DSLs and graphical interaction with models. A person working in management may be able to create a UML diagram, but might not be knowledgeable on how to use a complex IDE such as Eclipse. However the usage of a DSL might require them to use Eclipse since the language is only supported for that tool as the developers prefer Eclipse.

In larger companies where many people of different backgrounds work with the same DSL, this problem will inherently arise without the usage of LSP or the continuous support of multiple editors. However, developing for and continually supporting multiple editors is very cost-intensive. Using LSP, the development effort for the DSL stays the same while users still are able to choose between their preferred editing tools (provided that the software supports LSP).

According to the paper, LSP has two big issues: the performance issues with running multiple language servers on a client machine and the requirement of access to the same file system for both the server and the client. Bünder points out that the name "language server" is misleading, as in most cases the language server runs on the same machine as the client, since both need access to the same file system. When an editor requires the interpretation of more than one language, multiple language servers need to be hosted simultaneously. This can lead to performance problems, as language servers can be resource-intensive [Bün19].

There are however ways of solving these issues. One would be to run both the client and language server on a server with better performance capabilities using web-based editing software such as Theia [Fouh]. Since the paper was written, Microsoft has also introduced an addition to LSP called LSIF. LSIF (Language Server Index Format) is a way for using LSP without the requirement of both the client and server having access to the same file system [Core]. This would effectively solve both of the issues as the language specifications could be hosted on servers which can also be updated by the maintainers of the language and when multiple languages are required the client connects

to multiple different language servers. In the future, it would also be interesting to see what this would allow for graphical and textual editors such as bigER.

## 3.3 Arrangement of model elements with libavoid-js

A good graphical representation is essential for visual models [BDC23]. Having a model which is well distributed over the drawing area requires a good algorithm, especially with cyclic graphs [RESvH17]. In an ER-diagram, graphs may quickly become cyclical. In Chapter 4.1 a small example graph is shown, which already presents multiple cycles despite its small scope. Developing an algorithm that is both fast and able to output a good looking representation is difficult to develop. Sprotty already implements a layouting engine for the whole graph by default, the Eclipse Layouting Kernel (ELK) [Foug]. This engine may be replaced by another layouting engine to modify the way Sprotty arranges the elements inside graphs, depending on the use case.

Libavoid-js is the library used by the original bigER tool to increase the readability of diagrams. The library is a web-viable translation of the Adaptagrams libavoid library and is designed to be used in conjunction with Sprotty [Hna23]. For the graphical representation of the model, it is important that the readability is preserved. Making the model easily understandable and more readable at a glance is one of the major advantages of having a graphical model over a textual one. By default, Sprotty is adequate at aligning the default graph. It is also possible for the user to rearrange the graph and align it according to personal preferences. However, having a well aligned graph generated by default can remove the step of realigning the graph to be readable. The graph may also often change due to changes in the textual concrete syntax. This will require the generation of a new graph, which should then also be readable. Thus, having a good standard graph can save time during the development of a model.

Libavoid-js is a library designed to improve the standard representation of graphs provided by Sprotty. The original Adaptagrams libavoid library is written in C++. It is a library for interactive diagram editors to allow for object-avoiding orthogonal and polyline routing [Ada]. The tool is used to better space out lines and objects inside a diagram to make it more readable. However, the original implementation is not well suited for web applications as it is written in C++. The libavoid-js library solves this issue as it is a translation of the library in JavaScript with type mappings for TypeScript.

## 3.4 DSL readability and transformations

A similar project to the bigER Langium tool is the DSL for System-Theoretic Process Analysis by Jette Petzold [Pet22]. The paper is concerned with the development of a domain-specific language for a risk analysis technique. Langium and Sprotty are both used in the creation of the DSL, making the aim of the paper similar to that of this. For this thesis the main focus will not lie in defining the ER DSL as the language of

the original BIGER tool will be continued. In the paper by Petzold, the creation of the underlying DSL is the main focus.

A good DSL is able to model its domain in a concise and readable way. This is true for both the textual aswell as the graphical concrete syntax. In the case of the ER DSL much of the graphical components can be taken from other visual models of ER diagrams such as UML. Meanwhile, the textual syntax may take inspiration from class notations of object-oriented programming languages. In the BIGER tool the graphical notation can be defined to mimic a variety of common visual modeling languages and the textual notation is not dissimilar to Java or JavaScript.

In their section on future work, Petzold mentions that a filter function improves the readability for graphical models. For complicated models such as the System-Theoretic Process Analysis DSL described in their paper, a filter function will have an immediate positive effect on the readability of and ease of understanding for a model. For ER diagrams, a filter function may not have an effect on small models. However for larger models such a function may prove invaluable. Designing a large IT-infrastructure with an ER diagram will require a large amount of entities and relations. Having a way to filter them could allow for easier creation, modification and observation. This leads towards the topic of model transformation, highlighting parts of the model for the respective observer. In the BIGER tool, this is implemented on a small scale as the tool allows users to define their desired visual representation. In the future, it may be valuable to have a way to translate a BIGER ER diagram into a UML class diagram or any other common model.

Another project related to BIGER Langium through a similar technology stack is the DescribeML tool, created by Joan Giner-Miguelez, Abel Gómez, and Jordi Cabot in 2022. Their paper, titled 'DescribeML: A Tool for Describing Machine Learning Datasets' [GMGC22], presents the tool as a model-driven approach based on a DSL designed for describing datasets for machine learning.

The aforementioned similarity in technologies used becomes apparent when looking at the generation process of the tool. It is available as a VSCode extension using its own Langium grammar with EBNF syntax and provides additional language services implemented in TypeScript as well as code generation for valid documents.

DescribeML's Grammar relates to the relevant aspects of a dataset. These include metadata like the title of the set and its version, the composition of the data, for example attributes of instances, and information regarding its provenance, such as details of the gathering and labeling processes. In addition, it allows for denoting social concerns with the data set, which are an issue across industries [BOJC16][CSM18][LP20] and may include sensitive attributes or protected groups.

The custom language services offered by DescribeML are syntax highlighting, which is done through a TextMate grammar like in BIGER, as well as validation and hints for correct usage of the provided features. Additionally, the paper mentions a preloader service, which automatically sets up a file by deriving basic information from the data.

Lastly, the code generator creates an HTML file documenting the description file using a predefined template compiled by Pug [Com]. The paper especially emphasizes the discoverablity, as in how well the content can be discovered by search engines, of the generated documentation, which it attributes to the vocabulary used in populating it.

CHAPTER 4

# Development of BigER Langium

In this chapter, we document and explain the practical work done to bring the BIGER tool to the Langium framework, describing both convenient and inconvenient aspects to deliver a balanced view meant for evaluating the feasibility of larger model engineering projects in Langium. We will go into detail about the translation of the grammar, the generation of SQL code from a textual model, the validation of element names or syntactic rules and the graphical representation of the model and toolbar via Sprotty.

Figure 4.1 shows a high-level overview of the project. Communication between the language server and the webview are handled by the VSCode extension. The extension communicates to the language server via, LSP while the webview and the extension communicate through VSCode and Sprotty actions. The marked components are the ones who differ most from the original BIGER and are further explained in this chapter.

## 4.1 Grammar

Translating grammar from Xtext to Langium is made easy as both meta-languages are based on EBNF-notation and they have only minor differences between themselves in most other aspects as well. One thing to note is that default terminals like ID are not imported but instead have to be defined in each file, though files created through the yeoman generator will contain them. This makes creating files without the generator tedious but also makes them entirely self-sufficient.

The most important distinction, however, is that Langium does not offer a construct for enumerating options that would be a functional equivalent to Xtext's 'enum'. For the grammar, this problem can be solved by defining rules which return a value from a series of alternative integers or strings. The drawback of this solution lies in the fact that these rules are not resolved to tangible elements in the abstract syntax tree and therefore cannot be referenced. The effects of this become apparent in later sections like 4.3

| **ER Concept** | **Concrete Textual Syntax in bigER Langium** | **Differing bigER notation** |
|---|---|---|
| entity | entity | |
| weak entity | weak entity | |
| inheritance | A extends B | |
| attribute | attribute: type | |
| optional attribute | attribute: type optional | |
| primary key | attribute: type key | |
| partial key | attribute: type partial-key | |
| derived key | attribute: type derived | |
| multivalued key | attribute: type multivalued | |
| public modifier | + (alt.: public) | |
| private modifier | - (alt.: private) | |
| protected modifier | ~ (alt.: protected) | |
| package modifier | # (alt.: package) | |
| relationship naming | rel relationship | relationship relationship |
| binary relationship | A -> B | |
| recursive relationship | A -> A | |
| ternary relationship | A -> B -> C | |
| left aggregation | A o- B | |
| right aggregation | A -o B | |
| left composition | A *- B | |
| right composition | A -* B | |
| one cardinality | A[1] -> B | |
| zero or one cardinality | A[0..1] -> B | |
| zero or more cardinality | A[0..N] -> B | |
| exactly one cardinality | A[1..1] -> B | |
| many cardinality | A[N] -> B | |
| one or more cardinality | A[1..N] -> B | |
| cardinality with role | A[1..N\|role] -> B | |

Table 4.1: Entity Relationship concepts and their bigER Langium concrete textual syntax translations

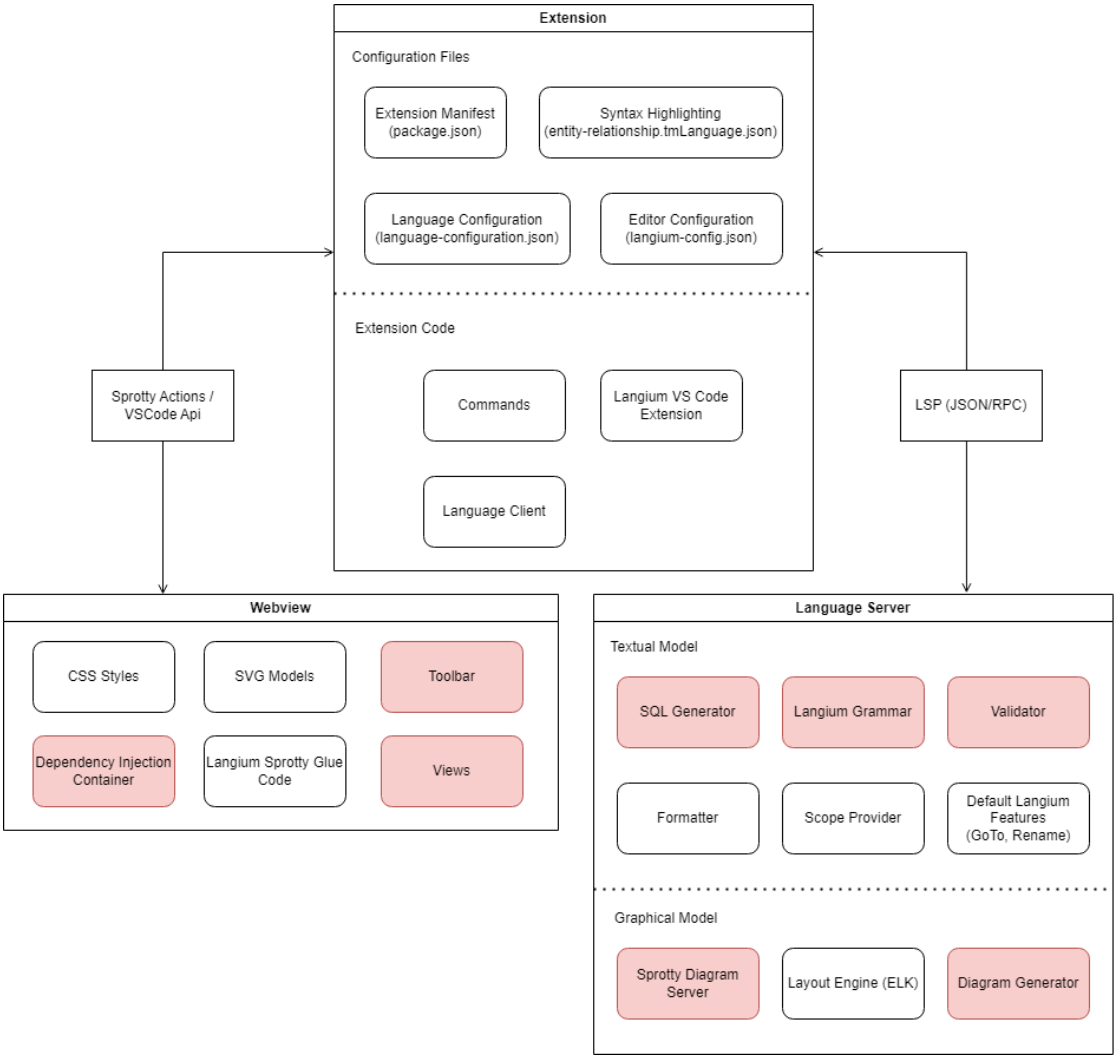


Figure 4.1: High-Level Overview of the BIGER Langium Structure (highlighted components are discussed in detail)

Table 4.1 shows how the ER concepts have been captured by the concrete textual syntax of the BIGER Langium textual notation. This syntax envelops the same concepts covered by BIGER Version 0.5.0. Even though the underlying technology is different, the textual representation is still largely the same, as seen in Table 4.1. The resulting textual concrete syntax supports the most important features of Entity-Relationship diagrams, such as aggregation and composition relations. Some features like enumerations and primitives are not yet supported [BJR00, cha. 2 p. 77-85].

Highlighting of the textual syntax, as in coloring for categories of keywords, is done through a TextMate grammar file located in the 'syntaxes' folder. TextMate is an open-source text editor that is highly customizable as it allows the user to create their

own grammar files [Ltd]. This makes it a quasi standard for projects related to word processing, such as VSCode [Cora] and its extensions [ARB+22] [GMGC22], LaTeX[SK07] and web IDEs [GL21]. Entries of a TextMate grammar use regular expressions to find keywords and link them to a highlighting style through the 'name' field before the expression. This style can be customized as well, although for this project, the predefined styles proved sufficient. Overall, the highlighting is largely the same as in the original bigER, although some changes have been made to incorporate lookaheads allowing for more complex keywords as in the case of relationships. Figure 4.2 and Figure 4.3 compare the differences in highlighting of cardinalities and relationship types. This applies in the same way for aggregation (-o/o-) and composition (-*/*-).

```
relationship Places {
    Customer[1..1] -> Order[N]
```

Figure 4.2: Highlighting in original bigER tool

```
rel Places {
    Customer[1..1] -> Order[N]
```

Figure 4.3: Highlighting in bigER Langium tool

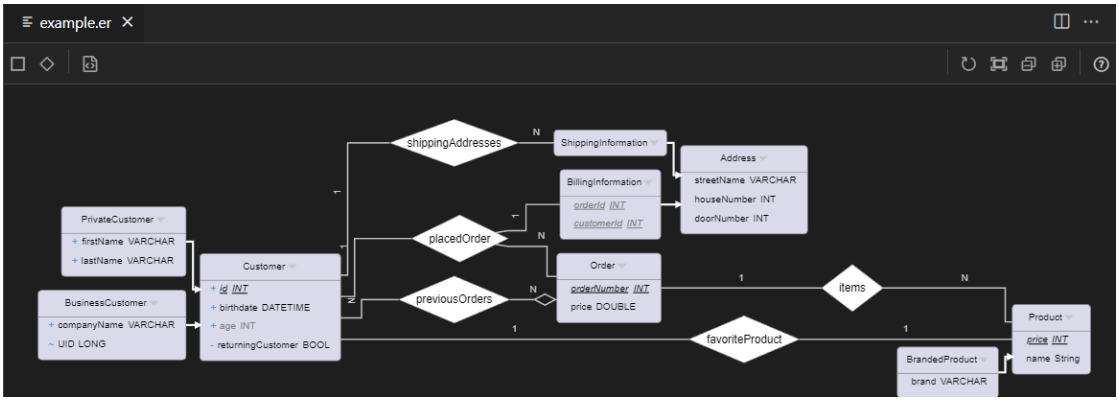


Figure 4.4: Graphical Concrete Syntax Example in bigER Langium

The textual syntax is then converted to a graphical syntax using Sprotty. Figure 4.4 shows how the generated graphical syntax is displayed. This representation is similar to standard UML. The full corresponding textual concrete syntax can be found in the appendix. Every entity, relationship, attribute, cardinality, visibilty and property that is defined in the textual concrete syntax is also shown in the graphical representation.

```
entity Customer {
  + id: INT key
  + birthdate: DATETIME
  + age: INT derived
  - returningCustomer: BOOL
}
```

Listing 4.1: Customer entity in the ER example

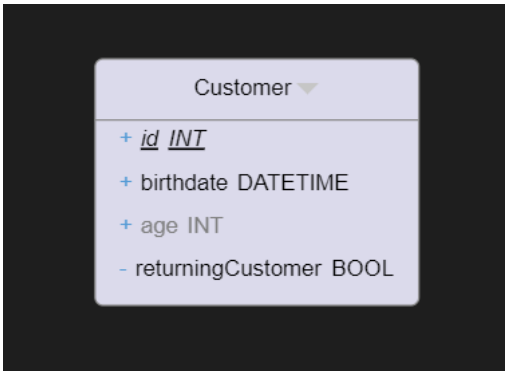


Figure 4.5: Customer ER Diagram

For instance, the customer entity seen in Listing 4.1 is shown as a rectangle with the title customer and a list of its attributes in the visual representation. The primary key is signified by the underscore while the derived attribute is displayed with a gray color. The relationship placedOrder shown in Listing 4.2 is signified by lines leading from each involved entity to a diamond shaped rectangle containing the name of the relationship. The lines leading to each of the entities also has a label showing their respective cardinality.

```
rel placedOrder {
  Customer[1]
  ->
  Order[N]
  ->
  BillingInformation[1]
}
```

Listing 4.2: placedOrder relationship in the ER example

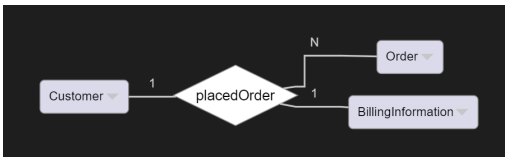


Figure 4.6: Placed Order ER Diagram

Like mentioned in Chapter 2 on Langium the grammar has to be defined in the Langium grammar language. When the Langium grammar is compiled, the lexer (Chevrotain) iterates through the grammar creating an abstract syntax tree in the process. This abstract syntax tree is the basis for all other modules and features the language server has. Once the AST has been created, the language server can already be run and will also have limited verification. This is not validation which will be created in a later step, but simply a check whether the content of the file matches the layout of the AST. For this process, the language server accesses the file and creates a concrete syntax tree using the parser (also Chevrotain). If the concrete syntax tree does not match the abstract syntax tree, an error is thrown in the form of an LSP response. This process can be seen in Figure 4.7. This also explains why LSP requires both the client and server to be able to access the same file system. The Concrete Syntax Tree is built on the server side only with information coming from the file that has been created or edited by the client.

The creation of the AST is part of the meta-modelling process and it is generated when

the grammar is compiled. It can then be used in code for creating modules. Therefore the AST can also not easily change during runtime and requires a new compilation of the grammar, meaning a new creation by the lexer. Meanwhile, CST is generated during runtime and used for any operations used in the language server modules. It is required for almost all LSP features as it is the basis on which the language server modules operate.

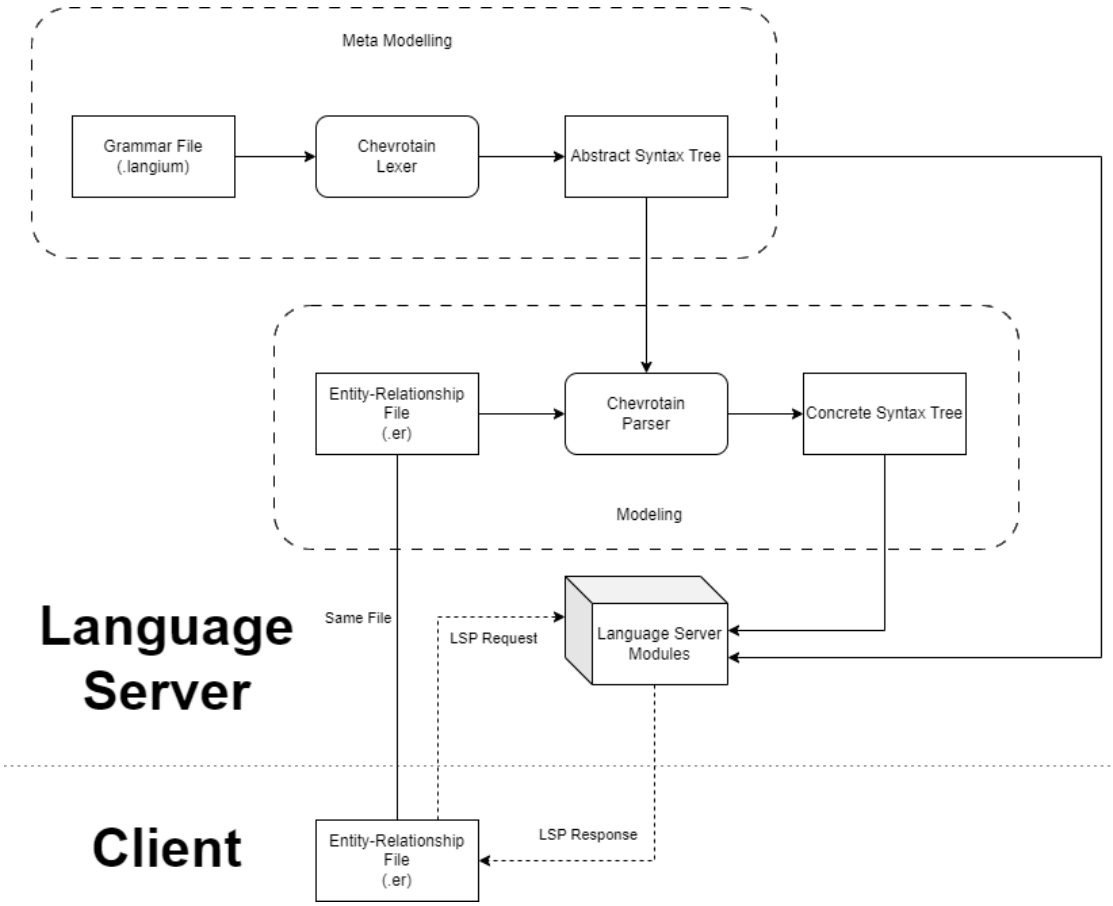


Figure 4.7: Interaction between AST, CST, Modules and the Client

## 4.2 Code Generation

This task was mainly focused on extracting the meaning of the original Java code and implementing a new version in TypeScript that matches it as closely as possible in terms of structure while improving the process where possible. The reason behind keeping a similar logical flow is to minimize the relearning effort for developers on one hand and to keep documentation of previous versions usable for gaining a high-granularity understanding of the generator. It should also be noted that, since this is primarily a proof of concept, some functionalities of the original are missing as of writing of this

thesis. Adding support for many different dialects of SQL as well as generation of different databases proved to considerably increase the amount of work and was therefore considered out of scope for this project. Instead we settled for a working prototype generating generic SQL code, which may be translated into the variants available for the original BIGER in the future.

Code generation is implemented as the command 'generateSQL', available via a command line interface such as PowerShell. After compiling the project with the 'yarn' command, it is always available inside the project folder. Global availability, however, requires adding the project to the system's path variable. The command needs to be supplied a path to the target file the code should be generated from and both relative and absolute paths are possible. In Addition, it can be supplied with two optional arguments. Option -d / –destination allows for supplying a desired file path for the newly generated .sql file. By default, the file is saved to a 'generated' folder in the same directory as the target file. Option -dr / –drop (without arguments) tells the program to create a series of drop table instructions for removing the tables instead of creating it. A default version of the command is also available through the right-click menu of the textual DSL as well as as a button in both textual and graphical representations.

The code works the same as the original BIGER, converting entities followed by relationships to table entries. The main process of creating these tables does this by writing a CREATE TABLE statement and inserting a list of attributes, the combination of primary keys and the selection of foreign keys. The process of dropping instead writes a DROP TABLE statement without further details. It is important to note that weak entities on their own are not persisted at all (since they have no unique identifier), but instead only when they are part of a weak relationship with a strong "owner" entity in which case they are entered along with the reference to their owner. The same is also true for dropping them, where they are automatically removed when their owner is deleted. The following paragraphs will give a detailed description focused on the main process and the functions involved.

Once a generation command is issued, the control flow enters through one of the exported functions, either generate() for creating or generateDrop() for dropping the table. These functions create a new SQL file at the given path, which includes creating any non-existent directories along said path. The helper function extractDestinationAndName() inside cli-utils.ts is responsible for parsing absolute and relative paths to get one uniform destination directory and file name. Generating the actual content for this file is handed off from the entry functions to generateFileContent() and the information whether a drop file is being created is from then on communicated through a boolean named 'drop' that is passed on to each function this is relevant for. GenerateFileContent() creates a string array of the elements in the model, converting entities and relationships to tables in the form of strings. It first loops through the strong entities, calling entityToTable() to get the actual content, before doing the same for relationships with weakRelationshipToTable() and relationshipToTable(). The order of these loops is significant, since relationships cannot be entered into a database before the entities they concern. The order is reversed

for drop statements, dropping relationships before entities to delete the database in an orderly fashion without relying on cascading deletes.

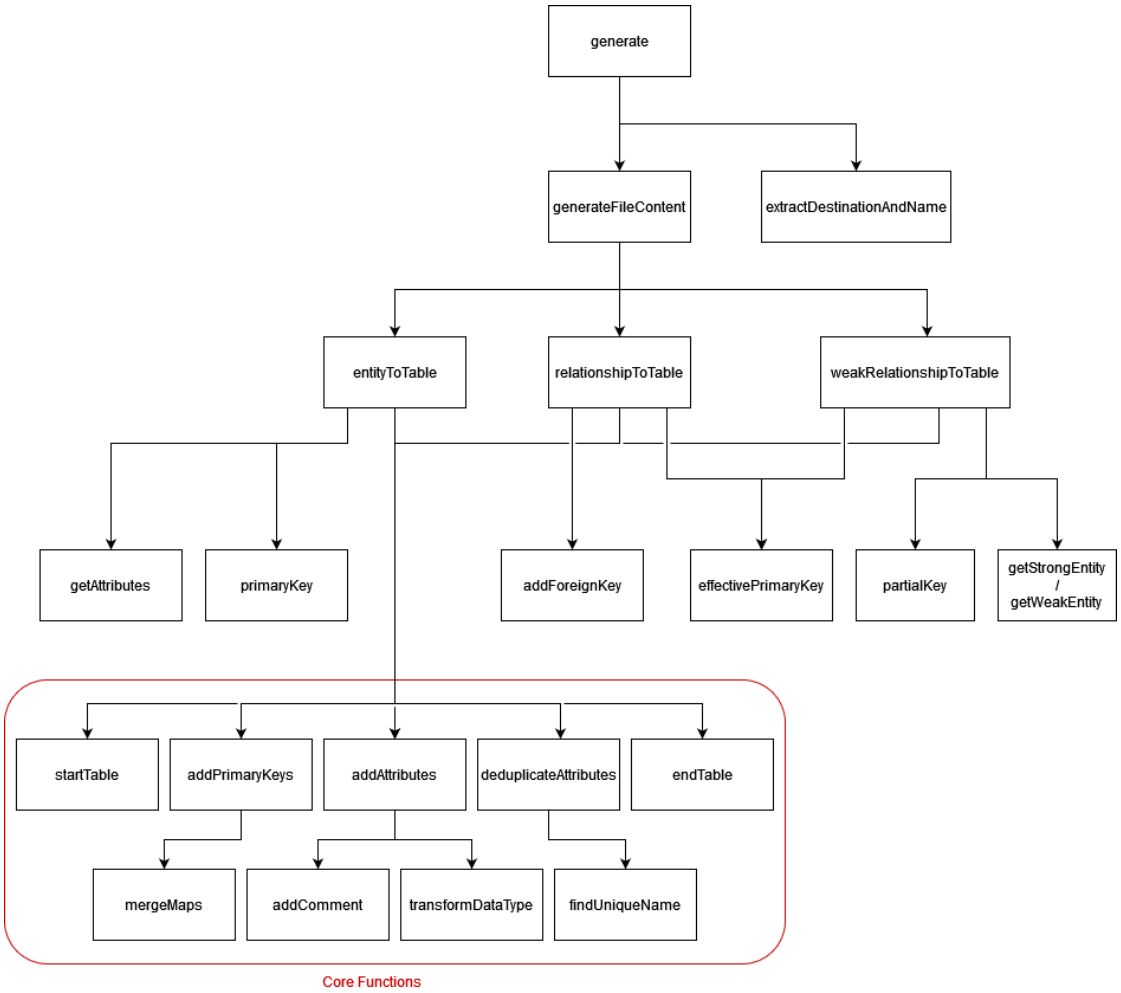


Figure 4.8: Call Hierarchy of the Code Generator

For the mentioned functions, there is a common core of sub-functions they all call in the same order and for similar reasons. This core begins with the startTable() function assembling a statement for creating a new table including the necessary keywords, its name and opening a bracket for the following content.

```
CREATE TABLE place_order (
```

Listing 1: Table Generation: Create Statement

Attributes of entities and relationships have to be given a name that is unique among their

siblings before they can be persisted, which is done in deduplicateAttributes() by looping through map and array types and generating unique names with findUniqueName(). This function maintains sets of taken attribute names for each model element, specifically one for entities and up to four for relationships, where one is for the attributes themselves and one for the primary keys of each entity in the relationship. Upon reading a name that is already taken for such a set, it renames the current duplicate by assigning it the first not-taken number starting from 2 and entering this new name into the set. The final names resulting from findUniqueName() are finally collected as keys in a map referring to their respective attributes.

This map is needed for the next step after deduplication, which is generating the code defining the attributes. The function addAttributes() uses it by iterating through every key-value-pair and assembling a line of code containing the final name and datatype of the attribute. If no datatype was provided, the function will default to the 'VARCHAR' type and assign a size of 255 characters. In this case, a comment documenting this action is added at the end of the line. The same is also done if the attribute was renamed in the previous step. This is done through the helper function addComment(), which ensures that if several comments are added to one line, they are separated by a semicolon.

```
CREATE TABLE place_order (
        id INT,
        order_nr INT,
        id2 VARCHAR(255),        -- renamed from: id; added default type
```

Listing 2: Table Generation: Attributes

Next, the as of yet unprocessed information about key attributes is handled in addPrimaryKeys(). Here, an array of all the maps containing keys of the current model element - such as those created in the case of relationships during deduplication - is converted to one statement listing all their entries. In the example code, each primary key originated from a separate map. To facilitate this conversion, the array of maps is first merged into one map with the helper function mergeMaps(). Since there is no pre-built way of merging maps in TypeScript, this helper copies every entry of every map into a new map and returns it. The names of each key are then added to the statement separated by commas, while the map as a whole is in turn added to a global map effectivePrimaryKeys, where it is referenced through the model element it was created for. The map effectivePrimaryKeys keeps track of all elements that have already been processed during the code generation process and will become relevant in combination with the function effectivePrimaryKey() mentioned later in this section.

The last of the core functions, endTable(), simply terminates the table by closing the bracket opened in start table and appending a semicolon. In addition, there are a few more functions specific to the kind of model element being processed. Starting with entities, not only their own attributes have to be processed in the core, but also inherited

```
CREATE TABLE place_order (
        id INT,
        order_nr INT,
        id2 VARCHAR(255),        -- renamed from: id; added default type
        PRIMARY KEY (id, order_nr, id2),
```

Listing 3: Table Generation: Primary Keys

ones. To this end, getAttributes() recursively goes through chain of ancestor entities and compiles an array of all the attributes in it to be passed on to deduplicateAttributes(). For the resulting map, the primary keys have to be determined before they can be added with addPrimaryKeys(). This is done in primaryKey() by searching the map for the subset whose type corresponds to 'key' and returning them as a new map.

For strong relationships, addForeignKey() specifies which of the primary keys reference a key of an involved entity by adding a 'FOREIGN KEY' statement for each such case. This statement contains the key as it is named in the relationship, the entity it was taken from, the key as it is named in the entity and an instruction to cascade in case the original key is deleted.

```
CREATE TABLE place_order (
        id INT,
        order_nr INT,
        id2 VARCHAR(255),        -- renamed from: id; added default type
        PRIMARY KEY (id, order_nr, id2),
        FOREIGN KEY (id) REFERENCES Customer (id) ON DELETE CASCADE,
        FOREIGN KEY (order_nr) REFERENCES Order (order_nr) ON DELETE CASCADE,
        FOREIGN KEY (id2) REFERENCES Seller (id) ON DELETE CASCADE
);
```

Listing 4: Table Generation: Foreign Keys

For weak relationships, there are the functions getStrongEntity() and getWeakEntity(), which determine the strong and weak entity of the relationship respectively. The strong entity is then used in weakRelationshipToTable() for determining the foreign keys of the relationship, while the weak entity is used for naming the relationship table. Since the weak entity has no keys of its own, another function partialKey() is called instead of primaryKey() to collect those of its attributes marked as partial keys instead of keys, thereon using them as keys of a strong entity would be used.

Shared between both kinds of relationships is effectivePrimaryKey(). It is used for retrieving the primary keys of each strong entity - exactly one for weak relationships and up to three for strong relationships - from the global map effectivePrimaryKeys. This retrieval doubles as an internal check whether the involved elements exist and were

processed in the correct order, as a relationship referring to an entity that is not contained in the map will throw an InvalidArgumentError. The resulting code creating relationship tables will therefore never refer to nonexistent tables.

```
function effectivePrimaryKey(entity: Entity): Map<String, Attribute> {
    const name = entity.name;
    if (!effectivePrimaryKeys.has(name)) {
        throw new InvalidArgumentError("Entity " + name + " not yet processed.")
    } else {
        return effectivePrimaryKeys.get(name)!;
    }
}
```

Listing 5: effectivePrimaryKey() keeping a list of processed entities

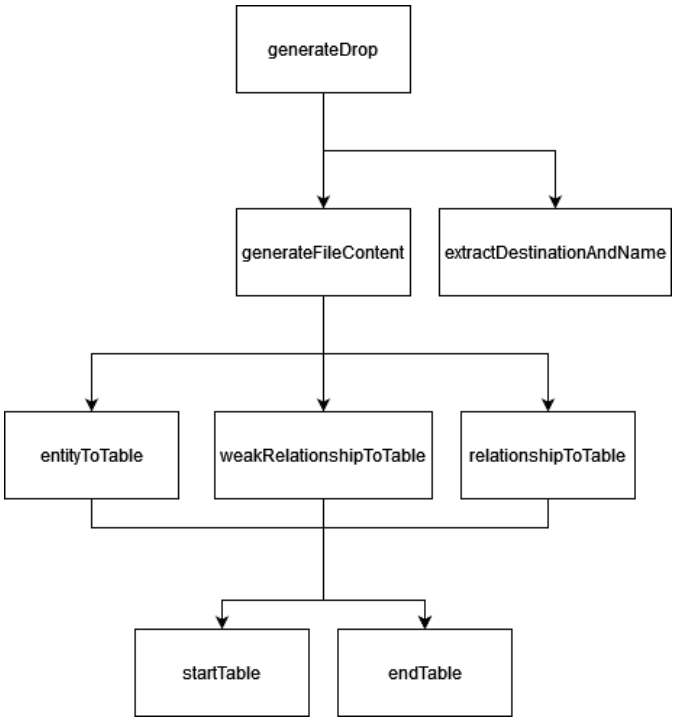


Figure 4.9: Call Hierarchy when generating a drop

If a drop is to be generated instead of creating the tables, each of the "toTable" functions will simply start and immediately end the table, resulting in only a drop statement followed by the name of the entity or relationship. Since the entries to be dropped are presumed to be valid (as they would have to have been entered into a database successfully for this feature to be useful in the first place), all functions determining keys or checking correct order of entries are skipped in favor of quicker code generation. In the

same vein functions generating table content such as adding and deduplicating attributes are skipped as well. This is also reflected by the simpler call hierarchy in Figure 4.9.

```
DROP TABLE place_order;
```

Listing 6: A generated drop statement

The main differences to the original generator are the function calls and data types used. Since TypeScript's default types are far more limited in number and consequently specialization, LinkedHashMaps became Maps, Lists became Arrays and StringConcatenations became CompositeGeneratorNodes, the latter of which being a replacement class added by Langium. Furthermore, since TypeScript does not support function overloading in the same way Java does, any occurrences of such overloading were disambiguated into functions whose names start with the corresponding input. An example of this would be the toTable() functions for entities and relationships being renamed to entityToTable() and relationshipToTable(), with weakToTable() being renamed to weakRelationshipToTable() for naming consistency.

In addition, we added the function getAttributes() for getting attributes of ancestor entities, as the original code generator did not consider inheritance when compiling attributes. In this new version, attributes of ancestors will show up as direct attributes would. This change also works with the other features mentioned in this section, such as keeping names unique and adding default types. The following code snippet gives an example of this by assigning both entities an 'id' attribute with no type.

```
entity Seller {
  id key
}

entity LocalSeller extends Seller {
  id key
}
```

Listing 7: Inheritance: ER example with identical attributes

When invoking the 'generateSQL' command, this ER input is converted to the following SQL statements.

We also achieved slight performance improvements by combining loops, such as the ones for weak and strong relationships inside generateFileContent(), and moving all calculations irrelevant for dropping tables inside an if-branch reflecting this, such as placing the determining of keys via effectivePrimaryKey() in relationshipToTable inside an `if(!drop)` check.

```
CREATE TABLE Seller (
        id VARCHAR(255),        -- added default type
        PRIMARY KEY (id)
);

CREATE TABLE LocalSeller (
        id VARCHAR(255),        -- added default type
        id2 VARCHAR(255),        -- renamed from: id; added default type
        PRIMARY KEY (id, id2)
);
```

Listing 8: Inheritance: generated Code

## 4.3 Validation

Implementing the validator again mainly consisted of translating the xtend code of the original BIGER to TypeScript to achieve the same functionalities with Langium. The current version is divided into naming checks inside the NamingValidator and conformity checks inside the EntityRelationshipValidator, both of which are located in src\language-server\validation.

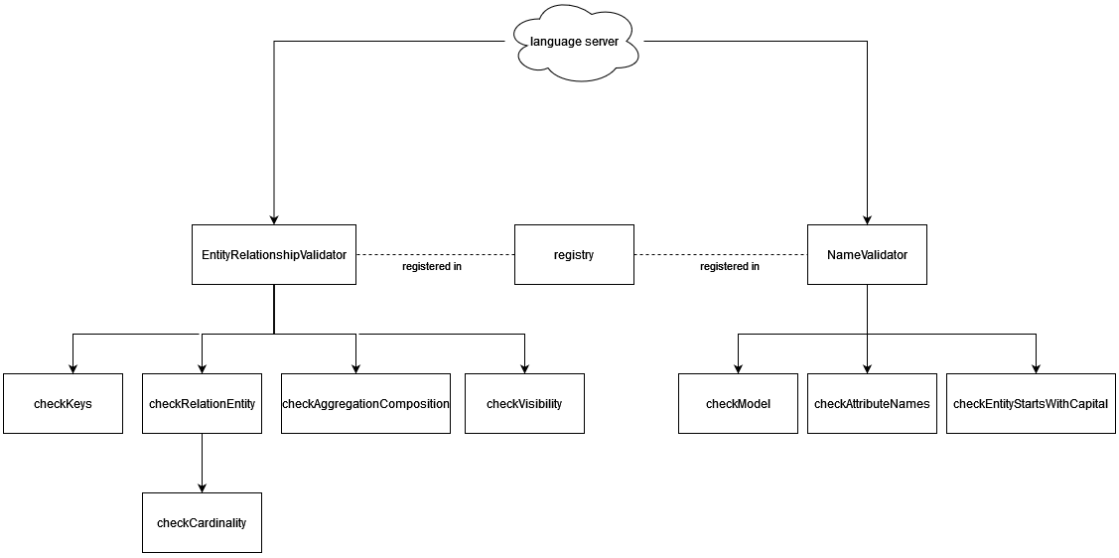


Figure 4.10: Call Hierarchy of Validation

Naming checks, as their name suggests, concern the naming of elements. Of these, checkModel() is the most extensive. It checks whether the model is named (ignoring whitespaces) and whether entities and relationships have unique names amongst themselves. An entity and a relationship having the same name is allowed, though, as they are still identifiable due to their different classes. For attributes within these elements,

there is checkAttributeNames() doing the same check for unique names. However, the scope for the check is now limited to one entity or relationship, meaning names belonging to different elements may have the same name. If the rules of either checkModel() or checkAttributeNames() are violated, an error is displayed because this will cause code generation and graphical rendering to malfunction or crash. This is not the case for the last function, checkEntityStartsWithCapital, which simply verifies entity names start with an upper-case letter. Since it only informs the user that lower case entity names are not formally correct, a warning instead of an error is sufficient.

Conformity checks concern the adherence to rules for entity-relationship-diagrams or specific forms of notation. The function checkKeys() checks if an entity has at least one key or if it has at least one partial key and is a weak entity. This is done by searching all of the entities owned and inherited attributes for one with a type matching a key or partial key. If one is found, the validation passes. Moving on, some notations allow only a reduced number of cardinalities. In the current state, these are bachman, chen and crowsfoot notation. The function checkRelationEntity() handles these cases by filtering for when one of them is used and then calling checkCardinality() with the RelationEntity in question and the ValidationAcceptor object used for displaying validation messages as parameters. checkCardinality() then passes the validation if the cardinality used is either '0..1', '0..N', '1' or 'N', since these are the possible combinations of the allowed cardinalities 'one' and 'many' and optional participation as described in [SEP95]

Of the current notations, only UML supports Aggregation and Composition, where Composition first appeared in Version 1.3 [BJR00]. checkAggregationComposition() therefore verifies that these relation types are used only in combination with UML notation and, if that is the case, that they are only used in binary relationships. The visibility operators offered by the BIGER tool are also only meant to be used if the notation is set to UML, which is validated by checkVisibility(). This does not include the visibility operator 'none' as it is just an explicit way of not providing one. Unlike the naming violations before, conformity checks are altogether presented as warnings instead of errors, as ignoring them is unlikely to produce fatal errors, although the resulting diagrams will at best ignore the input and at worst become invalid.

Structurally, this implementation of the validator differs from the original BIGER in the way validation messages are sent to the client. As mentioned in the description of checkRelationEntity(), instead of the separate warning, error and info methods in Xtend, there is one ValidationAcceptor object which is supplied with the details of the message. These details include the severity, which is the functional equivalent to the separate methods, the content to be displayed to the user and the node to display the message at.

In this section, we find the first widespread use of references to types from a list, such as attribute types or notation types. In the current state, these references are mere string comparisons, representing an instance of the recurring hard-coded 'references' necessitated by the lack of proper enumerations in Langium. This will be further elaborated on in Chapter 5

```
//get a list of all names that are equal
if (entityNames.filter(n => n === name).length > 1) {
accept('error', `Multiple entities named '${name}'.`,
    { node: entity, property: 'name'});
}
```

Listing 9: Usage of the ValidationAcceptor accept()

```
if (notation === 'uml') {
// TODO: replace hardcoded references to notation
// and relationship type with AST reference
    if (relationship.secondType) {
        if (relationship.firstType !== '->' ) {
        ...
        }
    }
}
```

Listing 10: Example of hardcoded references in the Validator

In terms of improvements, we were able to implement some interesting additional features. First, the original checkKeys() function would consider only the entity's own attributes, displaying a warning for missing keys or partial keys even if one was inherited from another entity. This has been rectified in tandem with the updated code generator in Chapter 4.2, meaning the extension as a whole properly handles inheritance now. A smaller improvement was made to checkAttributeNames(), which used to allow duplicates in relationships because it was only registered to entities. Harnessing the perks of TypeScript's flexible typing, this could be done by simply adding 'Relationship' as an alternative class for the 'element' parameter. Since the structural position of attributes in entities is the same as in relationships, accessing them as 'element.attributes' works in either case.

Lastly, we extended checkAggregation() to checkAggregationComposition by including an extra check and corresponding warning if a composition was used without UML notation. The existing warning for using aggregation in a ternary relationship was also improved by including composition, making the warning show even if only one part of the relationship was an aggregation/composition and writing a custom warning message for this case.

## 4.4 Sprotty Model-Generation

This section concerns the server-sided generation of the graphical representation of the model. This is done through the set of functions located in diagram\diagram-generator.ts upon receiving a diagram-related request, the most common of which are the 'Open in

Diagram' command in the textual editor and the 'Refresh Diagram' command in the graphical editor. Everything visible in the Sprotty diagram is an instance of a Sprotty interface. These are typically named starting with 'S' followed by a name corresponding to their function and will come up repeatedly in this section. While they are not covered in the Sprotty documentation directly, a description of them can be found in the Sprotty GitHub repository [Foue]. Due to the lax typing rules of TypeScript, results of functions are usually returned or pushed to an array as JSON-objects. For reasons of readability and type-safety, however, they are labelled when their type is not obvious.

```
return <SCompartment>{
    type: 'compartment:attribute-row',
    id: attributeId,
    layout: 'hbox',
    layoutOptions: {
        vAlign: 'center',
        hGap: 5
    },
    children: children
}
```

Listing 11: Example of a Sprotty class in JSON

Each model has one root element of type SModelRoot that is constructed in the 'generateRoot' function and contains all other elements. It is also what is returned to the client at the end of the generation process. The root element directly contains the four distinct elements that may be used when creating a diagram and are depicted in the example in Chapter 4.4. These are entities, relationships, relationship edges, and inheritance edges and are each generated in their own function, called generateNode(), generateRelNode(), generateRelEdges(), and inheritanceEdges() respectively.

The distinction between relationship and inheritance edges is due to the former connecting one entity with one relationship, while the latter connects one entity with another entity. Additionally, it is important to note that relationship edges are generated per relationship, therefore one call of generateRelEdges() usually results in two or three edges. The order of these function calls corresponds to the order they were mentioned before. This is significant for the generation of edges, as they attach to other elements as their source and target, which naturally need to exist at this point.

The function generateNode() creates the graphical representation of an entity. It consists of an attribute SCompartment and a name SCompartment, which is made up of a name SLabel for the entity and a button for expanding the attribute compartment. SLabel elements contain the text displayed in the graph. The attribute SCompartment is populated by a number of sub-compartments, one for each attribute, generated in createAttributeLabels(). Here, the sub-compartments are filled with SLabels for each property of the attribute, which are its name, its datatype and, if UML notation is used,

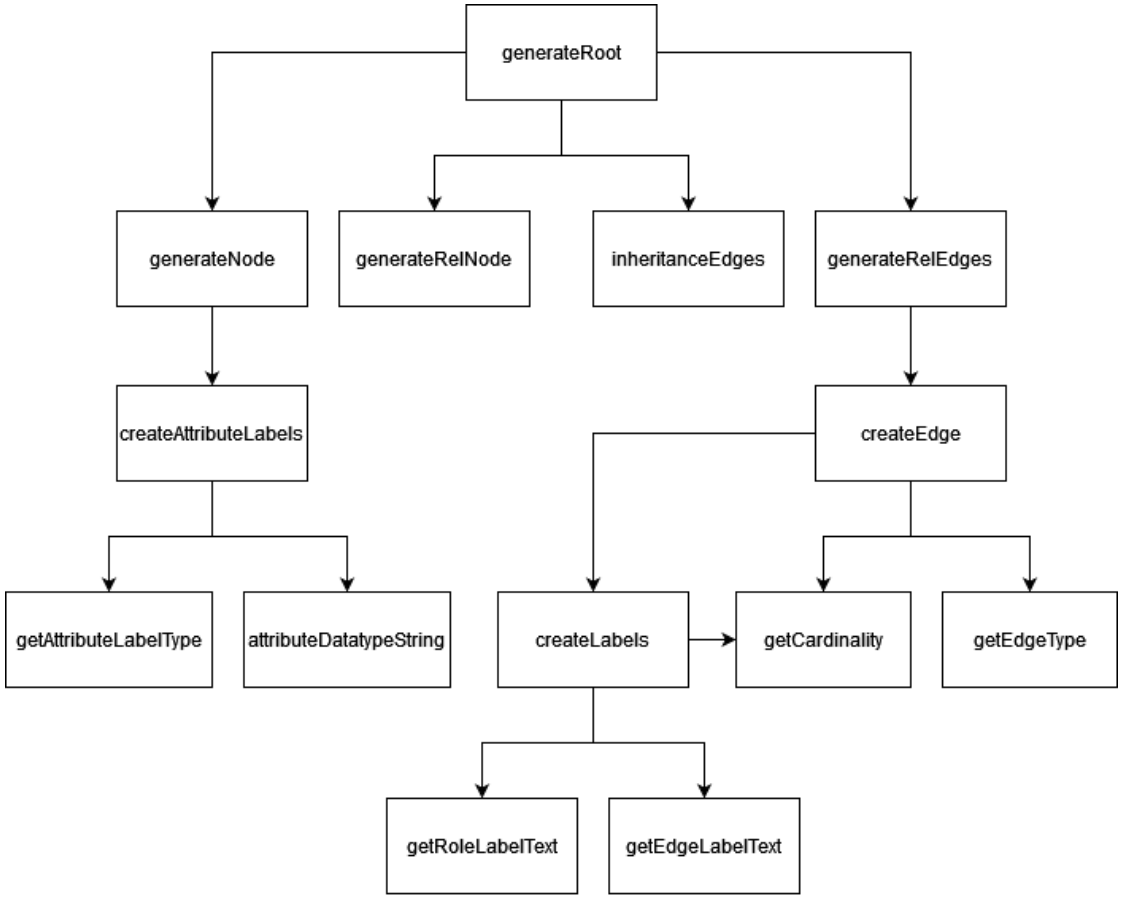


Figure 4.11: Call Hierarchy of the Diagram Generator

its visibility operator. Since the styling of an attribute is dependent on its logical type, each attribute receives a visual type. The visual type is derived from the logical one inside the helper function getAttributeLabelType(). The datatype is similarly transformed to a string by the helper function attributeDatatypeString(). It parses the datatype field of the attribute and returns a representation that depends on whether the field exists and if a size is attached.

The functions generateRelNode() and inheritanceEdges() are some of the simpler parts of the generator. The former adds just the relationship node itself, without any lines connecting it to entities, and then attaches an SLabel giving the name of the relationship. The latter creates an unlabeled edge between two entities connected through the 'extends' property. This edge has its source at the child entity (the one extending) and its target at the parent entity (the one being extended).

The last in the list, generateRelEdges(), is the most complex in both lines of code and number of subroutines. It is run for every relationship, creating an edge for every entity involved, which may be up to three in the case of a ternary relationship. For each edge,

it first determines a type, which corresponds to the types of the relationship, which may be either normal, aggregation, or composition. With this type, it then calls createEdge() to handle generating the actual edge. To achieve this, createEdge() determines many different properties from different sources. The source and target node are taken from the model context, the type of the edge is returned by getEdgeType(), the relationship type is given as a parameter by generateRelEdges() and the cardinality of the edge is determined by getCardinality(). The two helper functions that were mentioned, getEdgeType() and getCardinality(), are used for dealing with edge cases while reading the according fields, like a model in chen notation using a cardinality of '0..1' or '0..N'.

```
protected getEdgeType(relationEntity: RelationEntity,
    notationType: NotationType): string {
    if (notationType == 'chen') {
        const cardinality = relationEntity.cardinality ?
            relationEntity.cardinality : 'NONE'
        if (cardinality == '0..1' || cardinality == '0..N') {
            return 'edge:partial'
        }
    }
    return 'edge'
}
```

Listing 12: Edge cases handled by getEdgeType()

In addition to generating the edges, createEdge() also assigns them with labels, which it gets from createLabels(). These labels include the cardinality, as it was previously only assigned as a property but not visualized, but also a textual description of the role of the edge, which can be added with a vertical bar followed by the desired string next to the cardinality. For handling edge cases for the text, we have two new helper functions in addition to getCardinality() from before. getEdgeLabelText() omits the cardinality label for crowsfoot and chen notation, where they are communicated through the edges directly, while getRoleLabelText() replaces missing role text with a whitespace to avoid null values. The labels can be distinguished by the id assigned to them. '.label' is for the cardinality and '.roleLabel' is for assigning the role. In addition, there are three extra labels that are currently unused, '.relationName', '.additionalLabel' and '.additionalRoleLabel'.

With the structure in place, the last thing left to configure is the styling. This is done through the assigned types mentioned throughout the previous paragraphs. In the DiagramModule in di.config.ts, they are resolved to a view representing containing the details how each element is to be rendered in Sprotty. This view, in turn, can either be chosen from amongst the number of default representations offered by Sprotty or defined as an injectable inside the webview as was done for many elements in this project with the file 'views.tsx'.

Structurally, this implementation of the diagram generator is pretty close to the original,

although there are a few notable differences. One of them is the fact that model elements used to require manual tracing in Xtend, which is no longer the case in Langium. This leads to functions and function calls related to this process, such as traceAndMark(), being omitted in the current version.

```
def <T extends SModelElement> T traceAndMark(T sElement,
    EObject element, Context context) {
        return sElement.trace(element).addIssueMarkers(element, context)
}
```

Listing 13: Tracing method in the original BIGER

Furthermore, due to incompatible versions of the same class in Sprotty for VSCode and the Sprotty protocol on one hand and automatic type inference, which quickly becomes a nightmare to correct manually, on the other, some functions like those responsible for relationship edges and labels are defined with return type 'any'. Using 'any' in this way circumvents the aforementioned problems through TypeScript's structural typing, which allows transferring untyped objects. However, this still means the object has to be assigned a type eventually and is therefore only possible as long as the object does not contain properties that are not defined in (or inherited by) said type. If any other approach is taken, compiling the extension becomes impossible due to errors as shown in Figure 4.12.

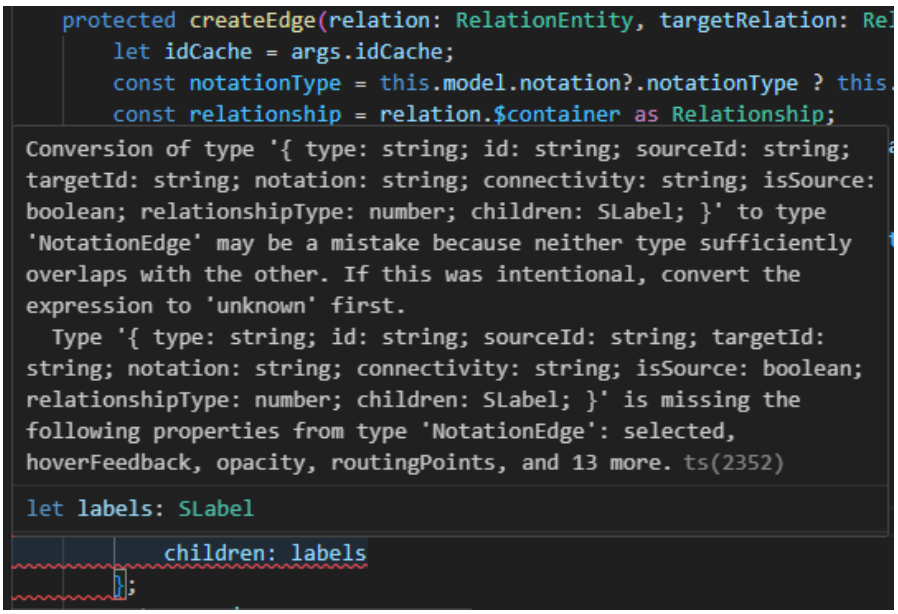


Figure 4.12: Error Message when explicitly typing SLabels

Functionally, there was one significant setback. The expand button in the original BIGER, which would extend and retract attribute lists of entities when clicked, worked through

an instance of IDiagramState. This object keeps track of the state of each element throug a list containing all currently expanded elements. This class, and by extension the list, is not present in Langium-Sprotty, meaning states can be neither adjusted nor tracked. Therefore, the expand button in the current version is non-functional, but was still included as a point of further research.

## 4.5 Toolbar

The toolbar contains functionality relevant to the traversal and functionality of the visual representation. It is like part of the Sprotty implementation and functions as an interface between the client visual representation, the language server, and the client visual features. Like the graph components explained in Chapter 4.4, the toolbar is implemented as a combination of Sprotty modules. However, it does not depend on the .er file that is being edited but rather persists as part of the graphical representation in the form of an overlay.

Through the toolbar various features are accessible. These features, however, do not all communicate with the language server. The "fit to screen" button for instance does not send an LSP request. Instead, it communicates with the web view. While no elements of the visual model are selected, it will zoom in or out to exactly fit the entire model. When elements are selected, it will instead try to fit all selected elements. Other features, like the "reload" button, do require LSP requests to function. The "reload" button requests a new visual model based on the contents of the textual model. These LSP requests are not sent by the webview directly. They are first sent to the extension, which translates them into LSP and relays the request to the language server via LSP.

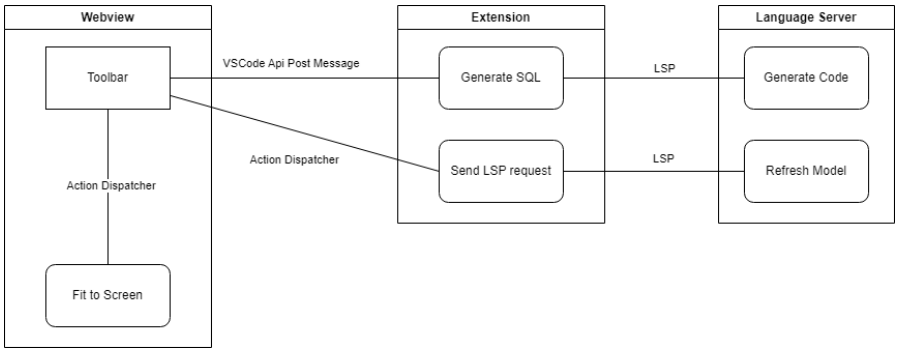


Figure 4.13: Communication of the toolbar with other components

Figure 4.13 shows how the toolbar communicates with the other components of the BigER Langium tool. Compared to the toolbar of the original BigER VSCode extension, this toolbar lacks some features. Most prominently, the creation of attributes and relations is not yet implemented. In the future, this feature would be used to create both entities and relations before being able to modify them in the diagram.

```
export class RefreshButton implements ToolButton {
    constructor(
        public readonly id = "btn_refresh",
        public readonly label = "Refresh Diagram",
        public readonly icon = "refresh",
        public readonly action = RefreshAction.create()
    ) {}
}
```

Listing 14: ReloadButton definition

The interactivity for the toolbar is achieved by using Sprotty buttons. These buttons are defined in the buttons.ts file. Listing 14 shows how the refresh button was defined. To implement it in the toolbar, it has to be created as a model. Listing 15 displays the way the reload button is implemented as part of the toolbar. Finally, the toolbar as a whole has to be registered in the diagram config file (di.config.ts) as seen in Listing 16. This is done in order to bind the toolbar to the diagram as a whole.

```
protected createRightSide(): HTMLElement {
        const rightSide = createElement("div", ["toolbar-right"]);
        rightSide.appendChild(this.createSeparator());
        rightSide.appendChild(this.createToolButton(new RefreshButton()));
        rightSide.appendChild(this.createToolButton(new FitToScreenButton()));
        rightSide.appendChild(this.createToolButton(new CollapseAllButton()));
        rightSide.appendChild(this.createToolButton(new ExpandAllButton()));
        rightSide.appendChild(this.createSeparator());
        rightSide.appendChild(this.createHelpButton());
        return rightSide;
}
```

Listing 15: Reload Button implementation

```
import { ContainerModule } from "inversify";
import { TYPES } from "sprotty";
import { ToolBar } from "./toolbar";

const toolbarModule = new ContainerModule((bind) => {
    bind(ToolBar).toSelf().inSingletonScope();
    bind(TYPES.IUIExtension).toService(ToolBar);
});

export default toolbarModule;
```

Listing 16: di.config.ts file for the toolbar

CHAPTER 5

# Conclusion

Overall it can be said that the project managed to achieve many of its original goals, although unexpected difficulties in various parts necessitated a narrowing of the scope. This culminated in the presented prototype, which is fully functional, but lacks many quality-of-life features, such as graphical editing and database generation.

## 5.1 Findings

The unfinished state of Langium at time of development gave rise to a number of difficulties in implementing this project. First of all, while the documentation provides a good introduction to how projects are structured and useful pointers on how to get started, a lot of other important details are missing. For example, the way the command line is used in the documentation, "./bin/cli <command> <options>" only works for Linux, which is not acknowledged and there is no alternative provided, leaving users to figure out the correct command in windows, which is "node bin/cli <command> <options>", for themselves.

The design of Langium sometimes leads to problems with passing information from the grammar over the abstract syntax tree to services further downstream like validation or the diagram generator. The most immediate issue is the lack of enumerations. Inside the grammar, this issue can be solved by listing a number of rules which each return a particular string. However, this workaround is not translated into the AST, making concrete values of the makeshift enum impossible to refer to outside the grammar. As there is no other solution provided in the documentation, this necessitates a large amount of hardcoding enum values inside the services, increasing the amount of work needed for future refactoring.

Sprotty suffers from some of the same issues as Langium, especially in that its documentation focuses on frequently used parts of the code, while leaving many other features

with only vague descriptions, which quickly become insufficient when working with them and trying to understand their inner workings. One feature where this becomes apparent is the "Expandable" interface, which is described as "Model elements that implement this interface can be expanded/collapsed" with no further elaboration as to how this works or where further information on it may be found, which became a problem when the feature did not work as intended.

Despite these issues, we judge the result of our investigation to be in favor of switching to the Langium framework, due to one important fact. Almost all of the problems we faced were caused by our usage of rather new technologies, the Langium framework more so than Sprotty. Accordingly, there are new features being added every few months [Gmba], which stands in stark contrast to Xtext, where as of writing of this thesis, no large changes have been made in over a year. This means that in order to take advantage of new developments in the field of language engineering, the bigER project should not rely on Xtext going forward.

Besides this, there are other advantages of using the Langium framework. Langium requires little proprietary languages, only utilizing the Langium grammar language for the creation of its grammar. Its syntax is very close to that of EBNF and by extension Xtext. This makes getting started with Langium relatively easy. Building modules with Langium is done in TypeScript, which is more welcoming and well known than Xtend, required for LSP features in Xtext. Since both VSCode and Langium are based on TypeScript it also greatly reduced the code complexity for this project. However language servers should be able to work with different code editors by design and this was only an advantage in the case of the development for exactly this tool.

Langium is well suited for web-based application due to its codebase being almost entirely TypeScript. From what was gathered during research for this thesis, it seems that the direction language servers are going indicates that future code editors will be more and more web-based. Langium will be a great toolkit for working with these applications.

## 5.2 Further Research

The bigER Langium version as it is still requiring a few features to be usable independently as a textual and graphical modeling tool for entity relationship diagrams.

The most important aspect is the implementation of graphical editing capabilities. It should be possible to create a model entirely in the graphical editor and generate a textual model from there. However, due to scope limitations for this thesis it was not possible to implement this feature as it would require a lot of additional code since the integration between Sprotty and Langium does not support this out of the box.

Another feature that should be added to the tool is a way to translate code written in bigER to the bigER Langium version. This is especially important since bigER Langium should be backwards compatible to the Xtext based version. Users of the

VSCode bigER extension should have a way to translate their work to the new bigER Langium extension.

To continue using bigER Langium as a replacement for bigER it will also be of essence to add the additional features currently present in bigER but not in bigER Langium.

Like mentioned in Chapter 3.2, one of the biggest disadvantages of LSP right now is the performance issue due to the language server needing to run on the same machine as the client to access the same file system. In the future it would be beneficial to make the bigER Langium tool accessible for strictly web-based editing tools such as Eclipse Theia. By doing so the tool would become even more accessible not only eliminating the Java dependency of bigER but also removing the need for any software installs (VSCode).

As it stands, many of the future features are limited by the current development status of Langium and Sprotty. As both frameworks are still in active development, many of the core features are prone to change, as they have done during the development of bigER Langium. They are both extremely useful tools for the development of language servers however due to the lack of documentation working with them is limited and very time inefficient.

# List of Figures

# List of Tables

# List of Algorithms

# Acronyms

**ALL** Adaptive Left-to-right, Leftmost derivation. 8

**ANTLR** ANother Tool for Language Recognition. 8

**AST** Abstract Syntax Tree. 8, 17, 18, 35, 39

**CSS** Cascading Style Sheets. 5

**CST** Concrete Syntax Tree. 17, 18, 39

**DSL** Domain-Specific Language. 4, 5, 9–11, 19

**EBNF** Extended Backus-Naur form. 8, 11, 13, 36

**ELK** Eclipse Layouting Kernel. 10

**ER** Entity-Relationship. 10, 11, 14, 15, 24, 43

**HTML** HyperText Markup Language. 5, 12

**IDE** Integrated Development Environment. 3, 9, 16

**IT** Information Technology. 9, 11

**JSON** JavaScript Object Notation. 3, 7, 28, 43

**LL** Left-to-right, Leftmost derivation. 8

**LSIF** Language Server Index Format. 9

**LSP** Language Server Protocol. vii, 2–5, 8, 9, 13, 17, 18, 32, 36, 37

**RPC** Remote Procedure Call. 3

**SQL** Structured Query Language. 7, 13, 19, 24

**SVG** Scalable Vector Graphics. 5

**UML** Unified Modeling Language. 9, 11, 16, 26–28

**VSCode** Visual Studio Code. 3–5, 8, 11, 13, 16, 31, 32, 36, 37

# Bibliography


[Ada] Adaptagrams. `https://www.adaptagrams.org/documentation/libavoid.html`. Accessed: 2024-1-07.

[ARB+22] Sander Albers, Nando Reij, Wouter Brinksma, Lex Bijlsma, and Harrie Passier. Towards a tool to support students through procedural programming guidance. In *Proceedings of the 11th Computer Science Education Research Conference*, pages 13–23, 2022.

[BDC23] Dominik Bork and Giuliano De Carlo. An extended taxonomy of advanced information visualization and interaction in conceptual modeling. *Data & Knowledge Engineering*, 147:102209, 2023.

[BJR00] Grady Booch, Ivar Jacobson, and Jim Rumbaugh. *OMG Unified Modeling Language Specification Version 1.3*. Rational Software Corporation, 2000.

[BK20] Hendrik Bünder and Herbert Kuchen. Towards multi-editor support for domain-specific languages utilizing the language server protocol. In *Model-Driven Engineering and Software Development: 7th International Conference, MODELSWARD 2019, Prague, Czech Republic, February 20–22, 2019, Revised Selected Papers 7*, pages 225–245. Springer, 2020.

[BL23] Dominik Bork and Philip Langer. Language server protocol: An introduction to the protocol, its use, and adoption for web modeling tools. *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, 18:9–1, 2023.

[BOJC16] Gema Bello-Orgaz, Jason J Jung, and David Camacho. Social big data: Recent achievements and new challenges. *Information Fusion*, 28:45–59, 2016.

[Bün19] Hendrik Bünder. Decoupling language and editor-the impact of the language server protocol on textual domain-specific languages. In *MODELSWARD*, pages 129–140, 2019.

[Com] PugJS Community. `https://github.com/pugjs/pug`. Accessed: 2024-1-16.

[Cora] Microsoft Corp. https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide. Accessed: 2024-1-16.

[Corb] Microsoft Corp. https://microsoft.github.io/language-server-protocol/. Accessed: 2023-12-26.

[Corc] Microsoft Corp. https://microsoft.github.io/language-server-protocol/implementors/servers/. Accessed: 2024-1-06.

[Cord] Microsoft Corp. https://microsoft.github.io/language-server-protocol/implementors/tools/. Accessed: 2024-1-06.

[Core] Microsoft Corp. https://microsoft.github.io/language-server-protocol/specifications/lsif/0.6.0/specification/. Accessed: 2024-1-06.

[CSM18] Danton S Char, Nigam H Shah, and David Magnus. Implementing machine learning in health care—addressing ethical challenges. *The New England journal of medicine*, 378(11):981, 2018.

[EV06] Sven Efftinge and Markus Völter. oaw xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, 2006.

[Foua] Eclipse Foundation. https://eclipse.dev/Xtext/releasenotes.html. Accessed: 2023-12-10.

[Foub] Eclipse Foundation. https://eclipse.dev/Xtext/xtend/. Accessed: 2024-01-07.

[Fouc] Eclipse Foundation. https://github.com/eclipse-sprotty/sprotty. Accessed: 2023-12-18.

[Foud] Eclipse Foundation. https://github.com/eclipse-sprotty/sprotty-vscode. Accessed: 2024-1-07.

[Foue] Eclipse Foundation. https://github.com/eclipse-sprotty/sprotty/blob/master/packages/sprotty-protocol/src/model.ts. Accessed: 2024-01-05.

[Fouf] Eclipse Foundation. https://sprotty.org/. Accessed: 2023-12-18.

[Foug] Eclipse Foundation. https://sprotty.org/docs/svg-rendering/#server-layout. Accessed: 2024-1-07.

[Fouh] Eclipse Foundation. https://theia-ide.org/. Accessed: 2024-1-06.

[Fuh11] Hauke A. L. Fuhrmann. *On the Pragmatics of Graphical Modeling*. Number 2011-1 in Kiel Computer Science Series. Department of Computer Science, May 2011. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel.

[Gau] bd82 Gaurav, Shahar Soel. `https://chevrotain.io/docs/guide/resolving_grammar_errors.html`. Accessed: 2024-1-06.

[GB21] Philipp-Lorenz Glaser and Dominik Bork. The biger tool-hybrid textual and graphical modeling of entity relationships in vs code. In *2021 IEEE 25th International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 337–340. IEEE, 2021.

[GL21] Ilya Gornev and Tatiana Liakh. Ride: Theia-based web ide for the reflex language. In *2021 IEEE 22nd International Conference of Young Professionals in Electron Devices and Materials (EDM)*, pages 503–506. IEEE, 2021.

[Gmba] TypeFox GmbH. `https://github.com/eclipse-langium/langium/blob/main/packages/langium/CHANGELOG.md#v210-nov-2023`. Accessed: 2024-01-06.

[Gmbb] TypeFox GmbH. `https://github.com/eclipse-langium/langium/releases?page=2`. Accessed: 2023-12-11.

[Gmbc] TypeFox GmbH. `https://langium.org/`. Accessed: 2023-12-11.

[Gmbd] TypeFox GmbH. `https://langium.org/docs/grammar-language/`. Accessed: 2023-12-11.

[GMGC22] Joan Giner-Miguelez, Abel Gómez, and Jordi Cabot. Describeml: a tool for describing machine learning datasets. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 22–26, 2022.

[Hna23] Vladyslav Hnatiuk. Adaptagrams/libavoid for sprotty. 2023.

[LP20] Samuele Lo Piano. Ethical principles in machine learning and artificial intelligence: cases from the field and possible ways forward. *Humanities and Social Sciences Communications*, 7(1):1–7, 2020.

[Ltd] MacroMates Ltd. `https://macromates.com/`. Accessed: 2024-1-16.

[Pet22] Jette Petzold. A textual domain specific language for system-theoretic process analysis, 2022.

[PHF14] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll (*) parsing: the power of dynamic analysis. *ACM SIGPLAN Notices*, 49(10):579–598, 2014.

[REIWC18] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. Towards a language server protocol infrastructure for graphical modeling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 370–380, 2018.

[RESvH17] Ulf Rüegg, Thorsten Ehlers, Miro Spönemann, and Reinhard von Hanxleden. Generalized layerings for arbitrary and fixed drawing areas. *J. Graph Algorithms Appl.*, 21(5):823–856, 2017.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[SEP95] Il-Yeol Song, Mary Evans, and Eun K Park. A comparative analysis of entity-relationship diagrams. *Journal of Computer and Software Engineering*, 3(4):427–459, 1995.

[SK07] Charilaos Skiadas and Thomas Kjosmoen. Latexing with textmate. *The PracTEX Journal,(3)*, 2007.

[Spo] Miro Spoenemann. `https://www.typefox.io/blog/langium-1.0-a-mature-language-toolkit/`. Accessed: 2024-1-06.

# Source Code

## ER Langium Grammar

```
grammar EntityRelationship

entry Model:
    'erdiagram' name=ID
        (notation=NotationOption?)
    (entities+=Entity | relationships+=Relationship)*;

NotationOption:
    'notation' '=' notationType=NotationType;

Entity:
    (weak?='weak')? 'entity' name=ID ('extends' extends=[Entity])? '{'
        (attributes+=Attribute)*
    '}';

Relationship:
    (weak?='weak')? 'rel' name=ID '{'
        (source=RelationEntity ((firstType=RelationshipType
        target=RelationEntity) (secondType=RelationshipType
        target2=RelationEntity)?)?)?
        (attributes += Attribute)*
    '}';

RelationEntity:
    target=[Entity:ID] ('['
        cardinality=CardinalityType ('|' role=STRING)?
        ']')?;

Attribute:
    (visibility=VisibilityType)? name=ID
        (':' datatype=DataType)? (type=AttributeType)?;

DataType:
    type=ID ('(' size=INT (',' d=INT)? ')')?;


AttributeType returns string:
    ATTR_NONE | KEY | PARTIAL_KEY | OPTIONAL | DERIVED | MULTIVALUED;
```

```
ATTR_NONE returns string: 'none';
KEY returns string: 'key';
PARTIAL_KEY returns string: 'partial-key';
OPTIONAL returns string: 'optional';
DERIVED returns string: 'derived';
MULTIVALUED returns string: 'multivalued';

CardinalityType returns string:
    CARD_NONE | ZERO_OR_ONE | ZERO_OR_MORE | ONE |
        EXACTLY_ONE | MANY | ONE_OR_MORE;
CARD_NONE returns string: 'NONE';
ZERO_OR_ONE returns string: '0..1';
ZERO_OR_MORE returns string: '0..N';
ONE returns string: '1';
EXACTLY_ONE returns string: '1..1';
MANY returns string: 'N';
ONE_OR_MORE returns string: '1..N';

NotationType returns string:
    NOTA_DEFAULT | CHEN | BACHMAN | CROWSFOOT | UML;
NOTA_DEFAULT returns string: 'default';
CHEN returns string: 'chen';
BACHMAN returns string: 'bachman';
CROWSFOOT returns string: 'crowsfoot';
UML returns string: 'uml';

RelationshipType returns string:
    RELA_DEFAULT | AGGREGATION_LEFT | AGGREGATION_RIGHT |
        COMPOSITION_LEFT | COMPOSITION_RIGHT;
RELA_DEFAULT returns string: '->';
AGGREGATION_LEFT returns string: 'o-';
AGGREGATION_RIGHT returns string: '-o';
COMPOSITION_LEFT returns string: '*-';
COMPOSITION_RIGHT returns string: '-*';

VisibilityType returns string:
    VISI_NONE | PUBLIC | PRIVATE | PROTECTED | PACKAGE | PUBLIC_STRING |
        PRIVATE_STRING | PROTECTED_STRING | PACKAGE_STRING;
VISI_NONE returns string: 'none';
PUBLIC returns string: '+';
PRIVATE returns string: '-';
PROTECTED returns string: '#';
PACKAGE returns string: '~';
PUBLIC_STRING returns string: 'public';
PRIVATE_STRING returns string: 'private';
PROTECTED_STRING returns string: 'protected';
PACKAGE_STRING returns string: 'package';


hidden terminal WS: /\s+/;

terminal ID: /[_a-zA-Z][\w_]*/;
terminal INT returns number: /[0-9]+/;
```

```
terminal STRING: /"(\\.|[^"\\])*"|'(\\.|[^'\\])*'/;

hidden terminal ML_COMMENT: /\/\*[\s\S]*?\*\//;
hidden terminal SL_COMMENT: /\/\/[^\n\r]*/;
```

---

## BigER Langium Textual Concrete Syntax Example

---

```
erdiagram example
notation = uml
entity Customer {
  + id: INT key
  + birthdate: DATETIME
  + age: INT derived
  - returningCustomer: BOOL
}
entity BusinessCustomer extends Customer {
  + companyName: VARCHAR
  ~ UID: LONG
}
entity PrivateCustomer extends Customer {
  + firstName: VARCHAR
  + lastName: VARCHAR
}
entity Address {
  streetName: VARCHAR multivalued
  houseNumber: INT multivalued
  doorNumber: INT optional
}
weak entity ShippingInformation extends Address {
}
weak entity BillingInformation extends Address {
  orderId: INT partial-key
  customerId: INT partial-key
}
entity Order {
  orderNumber: INT key
  price: DOUBLE
}
entity Product {
  price: INT key
  name: String
}
entity BrandedProduct extends Product {
  brand: VARCHAR
}
rel placedOrder {
  Customer[1]
  ->
  Order[N]
  ->
  BillingInformation[1]
```

```
}
rel previousOrders {
  Customer[N]
  -o
  Order[N]
}
rel favoriteProduct {
  Customer[1]
  ->
  Product[1]
}
rel shippingAddresses {
  Customer[1]
  ->
  ShippingInformation[N]
}
rel items {
  Order[1]
  ->
  Product[N]
}
```