TU WIEN Informatics

Multi-Notation Support for a Hybrid VS Code Modeling Tool

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software- and Information Engineering

by

Georg Hammerschmied
Registration Number 01633663

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork
Assistance: BSc. Philipp-Lorenz Glaser

Vienna, 9th September, 2022

Georg Hammerschmied	Dominik Bork

Erklärung zur Verfassung der Arbeit

Georg Hammerschmied

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, 9. September 2022

Georg Hammerschmied

Kurzfassung

Die Datenmodellierung und damit der Prozess der Strukturierung von Daten ist eine sehr wichtige Kernaufgabe der Wirtschaftsinformatik. Besonders das Entity-Relationship (ER)-Modell erfreut sich großer Beliebtheit und wird häufig für die Konzeption von Datenbankanwendungen verwendet. Aufgrund dieser Popularität sind viele Tools für die ER Modellierung entstanden. Diese Tools sind jedoch oft proprietär, unflexibel oder auf eine bestimmte Plattform beschränkt. In vielen Fällen bieten sie auch nur die Möglichkeit für eine textuelle oder eine grafische Modellierung. Hybride Lösungen bilden hier eher die Ausnahme. Aufgrund der aktuellen Entwicklung, dass Anwendungen auf Web-Plattformen verlagert werden, sind einige neue vielversprechende Technologien wie das Language Server Protocol (LSP) oder das Sprotty-Framework entstanden. Diese Technologien eignen sich hervorragend, die Einschränkungen der aktuellen Modellierungswerkzeuge zu überwinden. Die bigER-Erweiterung für Visual Studio Code baut auf diese Technologien auf und bietet verschiedene Funktionen zur flexiblen Spezifikation und Visualisierung konzeptioneller ER-Datenmodelle. Im Zuge dieser Arbeit wird ein Betrag zu der Release-Version von bigER geleistet, indem mehrere populäre ER-Notationen in den hybride Editor integrieren werden. Durch die Erweiterung der Grammatik auf der Serverseite und des grafischen Editors des Clients sind User in der Lage ER-Diagramme für die Notationen Bachman, Chen, Crow's Foot, Min-Max und UML zu erstellen. Diese zusätzlichen Notationen erweitern den Anwendungsbereich von bigER und es wird dadurch möglich eine größere Anzahl von BenutzerInnen anzusprechen. Die Unterstürzung von mehreren Notationen sollte bigER auch einen Vorteil gegenüber seinen direkten Konkurrenten aus dem Visual Studio Ökosystem bieten, da diese lediglich die Crow's Foot Notation unterstützen.

Abstract

Data modeling and therefore the process of structuring data is a very important core task for business informatics. Especially the entity-relationship (ER) model became very popular and is often used for the conceptual design of database applications. Because of this popularity, many tools employ its concept. Nonetheless, current modeling tools are often proprietary, inflexible, or restricted to a specific platform and in many cases limited to textual or graphical modeling. Resulting of the recent trend that applications are moving to web-platforms new promising technologies like the Language Server Protocol (LSP) and the Sprotty framework have emerged which provide aid to overcome the limitations of the current modeling tools. The bigER extension for Visual Studio Code makes use of those technologies and offers various features for flexibly specifying and visualizing conceptual ER data models. In this thesis, we contribute to the release version of bigER by integrating multi-notation support of popular ER notations for the hybrid modeling tool. By extending the textual language on the server side and the graphical view on the client, user can create ER Diagrams for the notations Bachman, Chen, Crow's Foot, Min-Max and UML. Because of the increased field of application by the multi-notation support a wider range of user can be addressed. The multi-notation support should also bring bigER an advantage over its main competitors in the Visual Studio Code ecosystem as they only rely on the Crow's Foot notation.

Contents

CHAPTER 1

Introduction

Data modeling and therefore the process of structuring data is a very important core task for business informatics. It creates a blueprint of the most valuable resources of a system and represents the requirements of the data in a conceptual way [1]. Data modeling also provides many advantages. Because of the increased transparency the maintenance effort is reduced and the models support software developers with their work [1, 2]. The entity-relationship model which is a popular high-level data model became the de-facto standard for the conceptual design of database applications. This popularity leads to the fact that many tools employ its concept. Nonetheless, current modeling tools are often proprietary, inflexible, or restricted to a specific platform and in many cases limited to textual or graphical modeling [3]. Additional to these shortcomings such tools often only support one ER notation which leads to limitations that restrict them to certain use cases.

Because of the recent trend that applications are moving to web-platforms new promising technologies have emerged which provide aid to overcome the limitations of the current modeling tools. One of these technologies is the Language Server Protocol (LSP) which provides an important contribution to web modeling of textual languages. It supports rich editing features like source code auto-completions or Go to Definition for a programming language. A single implementation of a language server can be reused in multiple Integrated Development Environments (IDE) and therefore the tools can support languages with minimal effort [4]. The downside of the LSP is that it is restricted to textual languages. By enhancing the LSP with the Sprotty framework textual and graphical modeling can be combined to create a hybrid modeling tool.

This thesis focuses on extending the hybrid modeling tool bigER to support the popular notations Bachman, Chen, Crow's, Min-Max, and UML. These notations share an underlying commonality but differ in the possibilities to define constraints for relationships [1]. This extension involves the underlying grammar of the textual editor and the notation-

specific rendering in the graphical editor. The multi-notation support increases the field of application of bigER and therefore should attract a wider range of users.

CHAPTER 2

Backgground

In this chapter, we cover background information and foundational concepts which are required for the implementation of multi-notation support for a hybrid modeling tool.

2.1 Data Modeling

The process of structuring data is called data modeling and it is a very important core task for business informatics. This is because data from a viewed section of reality has to be structured before it can be processed. The task of data modeling consists of a detailed description of information objects and the relation between them and the modeling process has to be very precise because it forms the basis of the later software development [2].

Data models o□er various advantages. They reduce the maintenance e□ort because of the increased transparency of the program code and they support the software developer with their work. The e□ort for the integration of a standard software solution into a existing environment or to exchange data form an old system to a new one can also be reduced significantly. Data models also result in a better communication for the requirements analysis between software developer and coworkers of other departments with a di□erent knowledge background [2].

2.2 Entity-Relationship Modeling

The entity-relationship (ER) model is a popular high-level conceptual data model. This model is often used for the conceptual design of database applications, and many tools employ its concepts [5].

Peter Chen introduced the Entity-Relationship model [6] in 1976 and the core components cover three fundamental elements [2].

Entity - An entity represents aspects of the real world in an abstract way. It will be identified by a name and it has certain attributes which describe the entity. An example for an entity is a customer or an article of a store. In ER diagram entities will be visualized as rectangles [2].

Relationship - Relationships describe the connection between entities. They are referred to by a name and can include attributes similarly to entities. An example of a relationship is a customer of a shop that buys multiple articles. In ER diagrams they have a diamond shape and are connected to the entities via edges [2].

Attribute - Attributes are used to describe details of entities or relationships. They can be used for example to describe the address of a customer or the price of an article [2].

Another important aspect of ER modeling are key attributes. Key attributes are a minimal set of attributes that are used to identify an entity. A key of only one value is referred to as a simple key and otherwise when the key contains more than one value it is called a composite key. Simple keys are easily created as consecutive numbering. An example of a simple key is a customer number or article number [2].

The connection between entities can be simple but also quite complex. For example, a customer can buy multiple articles in a shop but an article is most of the time bought by just one customer. For this problem, a relationship additionally defines constraints on the occurrences of participating entities. This connectivity is called cardinality and the value for the cardinality depends on the chosen ER notation. Each notation o□ers di□erent options to define the occurrence of the entities. This is a very important aspect of this thesis because cardinality is the main di□erence between the notations which have to be expressed for the integration of di□erent notations in a textual language. In the following the di□erences between the notations Bachman, Chen, Crow's Foot, Min-Max, and UML on how to define the cardinality are described.

2.2.1 Bachman

The Bachman notation is used to model binary relationships. The relation is represented as a line where both ends are connected to the involved entities. A filled-in circle at the end of a relationship indicates that the relationship is mandatory for a pair of entities. On the other hand, an open circle shows that an entity is optional. With the arrow, the cardinality many can be expressed. The arrows in combination with the two kinds of circles allows to define the cardinality one or many and zero or many [7].

Figure 2.1: Bachman optional entity

Figure 2.2: Bachman mandatory entity

Figure 2.3: Bachman one or more

Figure 2.4: Bachman zero or more

2.2.2 Chen

Peter Chen introduced the Chen notation [6] in 1976 and it is one of the most popular notations for ER models. When modeling after the Chen notation there exist three fundamental types of relationships.

1:1 relationship The 1:1-relation describes the connection between entities that have a definite allocation to each other. For example, a student can only have one student ID and the other way around a student ID can only be assigned to one student [2].

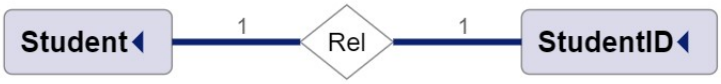

Figure 2.5: Chen 1:1 relationship

1:N relationship The 1:N relationship describes the relation of one entity that is connected with an arbitrary number of another entity. For example a student studies only on one university but a university has many students [2].

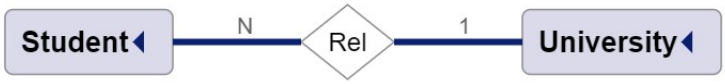

Figure 2.6: Chen 1:N relationship

M:N relationship In an M:N relationship an arbitrary number of one entity is connected with an arbitrary number of another entity. For example, a student can attend multiple lectures, and lectures are typically attended by multiple students [2].

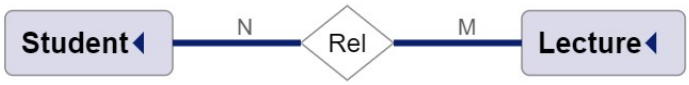

Figure 2.7: Chen M:N relationship

2.2.3 Crow's Foot

The Crow's Foot Notation is used for binary relations and the cardinality is expressed by four different types of edges. All of these edges are composed of three symbols. The circle is used for the cardinality zero. One is visualized with a straight line and for arbitrary many, the so-called crowfoot symbol is used [8]. These three symbols can be combined to express constraints of the cardinality.

Two straight lines form the cardinality one and only one.

Figure 2.8: Crow's Foot one and only one

With a straight line and a crowfoot, the cardinality one or more can be formed.

Figure 2.9: Crow's Foot one or more

A circle and a crowfoot form the cardinality zero or more.

Figure 2.10: Crow's Foot one or more

The cardinality zero or one is visualized by a straight line and a circle.

Figure 2.11: Crow's Foot one or more

2.2.4 Min-Max

The Min-Max notation is another way to define constraints on the relationship between entities. It was introduced because with the Chen notation certain cardinality constraints cannot be expressed. With the Min-Max notation it is possible to define a lower and a upper bound for the cardinality. For every entity in a relationship a minimum and a maximum value will be specified. The min and max values define how many entities of a certain kind are at least and at most involved in a relationship [2].

There are two possibilities on how to express the upper bound for the Min-Max notation which is shown in figure 2.12. The first way is to define a number as the upper bound which has to be equal to or greater than the lower bound and the second method is to use a '*'-symbol which stands for arbitrary many.

Figure 2.12: Cardinality for Min-Max

2.2.5 UML

The Unified Modeling Language (UML) is a general-purpose modeling language and it represents a collection of best engineering practices. It plays an important part in developing object-orientated software and it offers two different views of a system model. The static view uses objects, attributes, and relationships to visualize the structure of a system. This view includes the class diagram. On the other hand, the dynamical view focuses on the dynamic behavior of a system by showing collaborations between objects and changes to their internal state. This view includes activity and sequence diagrams [9]. For this thesis only certain aspects of the class diagram and therefore the static view are important. Those aspects consists of the cardinality, the role and the aggregation of an object. UML offers different possibilities to define a connectivity constraint on how many objects participate in an association.

1 - Exactly one entity

* - Zero or more entities

0..* - Zero or more entities

1..2 - At least one and at most two entities

The definition of a role for an entity is only used to clarify the nature of a relationship and it is not necessarily required to define roles for entities. Another important aspect of UML is the aggregation because it allows the creation of a part-whole relationship between two entities. There are two kinds of aggregation. The weak or shared aggregation which

is shown in figure 2.13 allows to model a relationship where one entity owns another entity, but other entities can own that entity as well. The second kind is the composition which is shown in figure 2.14 and it determines that one entity exclusively owns the other entity [8].

Figure 2.13: UML aggregation

Figure 2.14: UML composition

2.3 LSP

Supporting rich editing features like source code auto-completions or Go to Definition for a programming language in a developing editor traditionally implies a lot of work and it is very time-consuming. This is because the work for adding such features has to be repeated for each development tool, as each provides di□erent Application Programming Interfaces (API). A di□erent approach is provided by the Language Server Protocol (LSP) which allows creating a language server back end and therefore decoupling the implementation of language-specific servers from language-agnostic clients. A single implementation of a Language Server can be re-used in multiple IDEs, and therefore the tools can support languages with minimal e□ort [4].

The LSP standardizes the messages exchanged between an IDE and a language server to simplify these sorts of integrations and it provides a useful framework for exposing language features to a variety of tools. Tools like Visual Studio communicate with the server by using the language protocol over JSON-Remote Procedure Calls (RPC) which are considered lightweight, stateless, and communicate at the level of document references and document positions. Those types of data are neutral for programming languages and apply to all of them. Not every server can support all features of the protocol and therefore the client and the server both announce their supported feature set through capabilities during commuication [4].

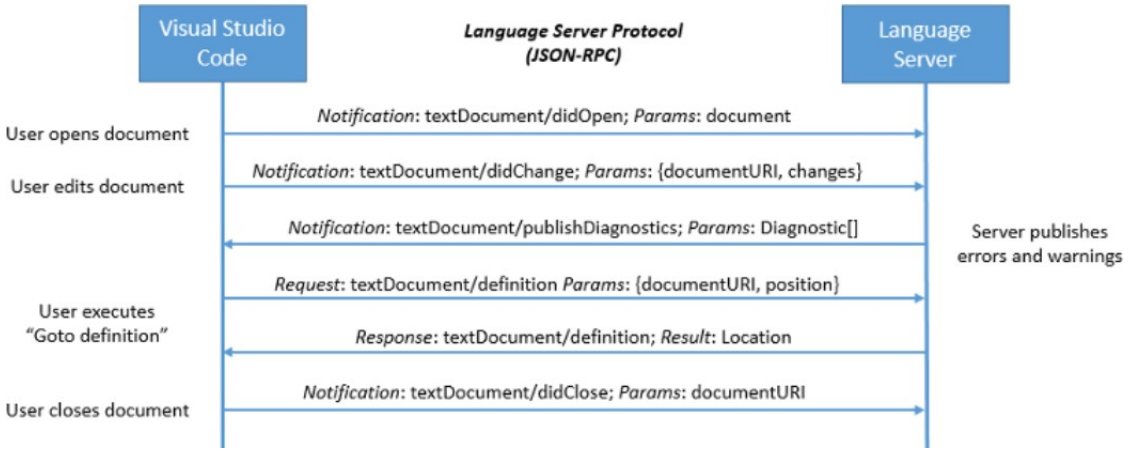

Figure 2.15: LSP communication during editing session [4]

For illustration, an example request- and response message for finding a definition are shown in listing 2.1 and listing 2.2.

```
{
    "jsonrpc": "2.0",
    "id" : 1,
    "method": "textDocument/definition",
    "params": {
        "textDocument": {
            "uri": "file:///p%3A/mseng/VSCode/Playgrounds/cpp/use.cpp"
        },
        "position": {
            "line": 3,
            "character": 12
        }
    }
}
```

Listing 2.1: JSON-RPC example request [4]

```
{
    "jsonrpc": "2.0",
    "id": "1",
    "result": {
        "uri": "file:///p%3A/mseng/VSCode/Playgrounds/cpp/provide.cpp",
        "range": {
            "start": {
                "line": 0,
                "character": 4
            },
            "end": {
                "line": 0,
                "character": 11
            }
        }
    }
}
```

Listing 2.2: JSON-RPC example response [4]

One major drawback is that the LSP primarily targets textual languages and therefore lacks in supporting prominent IDE features like graphical languages. This often leads to a mixture of different protocols to create hybrid modeling tools for graphical and textual languages. In this thesis, the Sprotty framework is used to extend the LSP to synchronize the textual- and graphical model of the editor.

2.4 Sprotty

Sprotty is an open source web-based framework which allows adding modern diagrams to web applications with little effort. Its architecture allows the distribution of the diagram execution between a client and a server. Diagrams often only visualize a small part of a big data set which means that the client only needs the required information to render the diagram which matches the LSP. Sprotty provides a powerful Java library that was designed to enhance language servers and therefore to take over the graphical part in IDEs. It works particularly well in combination with language servers generated by the Xtext framework [10].

The basis of an application created with Sprotty are two major components. The client that interacts with the user holds the current diagram of the model and renders it. By using Scalable Vector Graphics (SVG) the framework provides stable and scalable rendering on a wide range of browsers. The optional server knows about the underlying semantic model and how to map it to diagram elements. The client and the server communicate with each other by using a JSON protocol. The client passes messages to the server or processes them locally so Sprotty can also be used as a client-only app without a backend [11]. The code of the client is written in Typescript to avoid problems of cross-compilation. Both the server and the client are using dependency injection (InversifyJS/Guice) which offers great flexibility for production-ready applications as almost every aspect can be tweaked [10].

The Viewer, which is a main component of the architecture, uses the model to create a virtual Document Object Model (DOM) and it also adds event listeners for actions. These actions are used for operations on the graph model. The ActionDispatcher receives such actions from the Viewer and converts them to commands which describe the behavior of the operation by using the ActionHandler and sends them to the CommandStack. The CommandStack then executes the actions [12].

Sprotty stores the diagram in a model called SModel and all elements in the model inherit from SModelElement. Every SModelElement has an ID and a certain type to look up its corresponding view. Inside the model, the elements are structured as a tree and the root of the tree is always an instance of SModelRoot. The framework also provides a library to convert the model into a graph with nodes and edges as it is a common case for visualization but it is not mandatory. An SGraph consists of SNodes which are connected with SEdges. SLabels are used to get some text into the diagram [12]. The SGraph and its elements play an important role in later chapters for realizing the multi-notation support.

Figure 2.16 shows the architectural overview of Sprotty. The architecture where the key feature is a unidirectional cyclic event flow is inspired by FLUX and other reactive web frameworks [12].

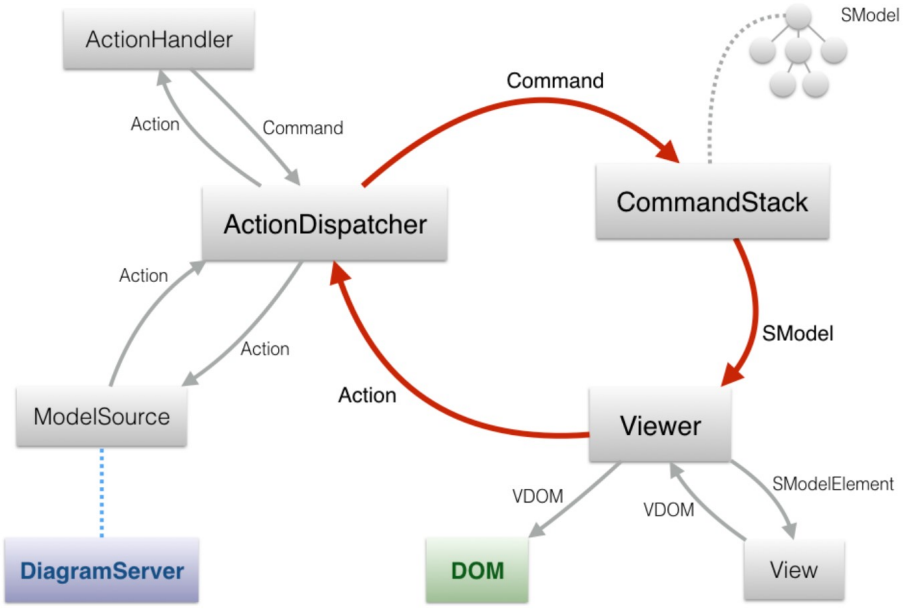

Figure 2.16: Sprotty Architectural-Overview [12]

2.5 Xtext

Xtext is an open-source framework for implementing domain-specific languages (DSLs) and it is part of the Eclipse-Modeling-Framework-Projects (EMF). It is easy to learn and therefore lets you implement languages quickly [13]. This framework is important for this thesis because of its integration with LSP and Sprotty. In the following, we focus only on aspects of the Xtext framework which are relevant for later chapters.
The Xtext framework only needs a grammar specification to start a DSL implementation. The grammar is similar to the Extended Backus–Naur Form (EBNF) but was enhanced with additional features for type inheritance and information for attributes and references. With the specified grammar file, the framework automatically generates a complete language infrastructure, including the parser, code generator, or interpreter and other components that can be customized through Dependency Injection (DI) [13].

For Xtext the grammar file plays an important role because it is a DSL designed to describe textual languages and the grammar of Xtext itself is also implemented with Xtext. The grammar file describes the syntax and how it is mapped to the semantic model which equals an in-memory object graph. This is done on the fly by the Xtext parser when it consumes the input file. Such object graphs are instances of EMF Ecore

models. These models consist of an EPackage that contains EClasses, EDataTypes, and EEnums, and these elements are used to describe the structure of the instantiated objects [14].

2.6 Visual Studio Code

Visual Studio Code is a web-based, lightweight but powerful source code editor made by Microsoft [15]. While the differences in tool choices depend on the type and role of a developer Visual Studio Code is one of the most popular editors across the board. In the Stack Overflow 2021 Developer Survey VS Code was ranked the most popular developer environment tool [16]. The editor can be installed on every common operating system like Windows, macOS, and Linux and its features include support for debugging, syntax highlighting, code completion, code refactoring, and version control. It also offers support for many different programming languages like Java, JavaScript, Go, Node.js, Python, C++, C, Rust, and Fortran [15]. The editor is based on the Electron framework which is used to develop Node.js web applications. One big benefit that VS Code offers is that it can be extended with additional plugins that increase the abilities and language support of the editor. The marketplace which is integrated into the user interface of the application allows selecting a favored plugin from all kinds of different extensions. So almost every part of the editor can be customized. This is possible through the Extension API which is used to enhance the editor. Even many core features are built as extensions and use the same API [17]. A language extension is the main aspect of this thesis and because VS Code offers a great ecosystem to deploy a hybrid modeling tool later chapters will explore how to extend a VS CODE extension.

CHAPTER 3

Multi-Notation Support for a Hybrid VS Code Modeling Tool

This chapter deals with the research objective on how to extend the bigER modeling tool which is based on LSP and Sprotty to support various ER notations. The modeling tool for creating ER diagrams is implemented as a VS Code extension and offers a textual and a graphical editor. The extension with a simple default ER notation will illustrate the approach and will be extended throughout the chapter. Because the whole extension was already built as a full-fledged hybrid modeling tool, this approach focuses on enabling the multi-notation support on the language server and on the rendering of the web view for the graphical editor.

First, we will look into the architectural overview of the VS Code extension followed by the extension of the language server. For this we will enhance the grammar with the characteristics of the notations Bachman, Chen, Crow's Foot, Min-Max, and UML and adapt the validation of the language model with additional constraints which are not possible to express through the grammar. After the validator, the generator for transforming the model to an SGraph will be extended and a NotationHandler will be created as preparation to allow a notation change on the client side as well. When the language server supports the five notations we will create an individual rendering for every notation in the webview of the client. To be able to switch the notation in the graphical editor as well we will add a drop-down to the toolbar of the webview and we will create a custom action to propagate a notation change from the webview to the language server.

3.1 VS Code Extension

Figure 3.1 shows the architecture overview of the VS Code extension for hybrid ER-Diagramm modeling. The bigER modeling tool consists of three core components. The

language server was enhanced with Sprotty to extend the LSP with Sprotty actions, the webview which renders the Sprotty diagrams of the graphical model, and the extension. The extension handles the communication by receiving and sending extended Sprotty LSP messages for textual- and graphical language features. The architecture is realized as a client-server application where the extension and the webview belong to the client side and the language server to the server side [18].

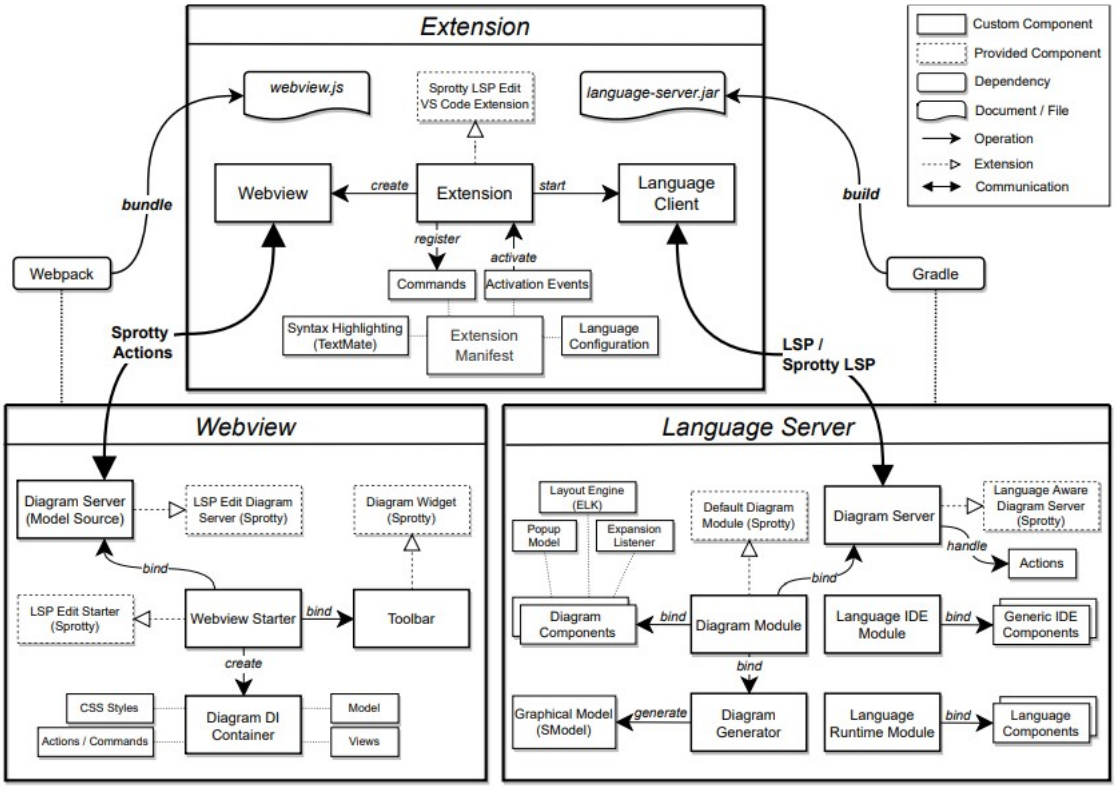

Figure 3.1: Architecture of the bigER modeling tool [18]

The language server provides language-specific functionality to the client and the textual language was implemented with the Xtext language workbench that generates a whole language infrastructure. To enable graphical modeling the language server was enhanced with graphical language features through the Sprotty framework. A diagram generator is required on the server side for Sprotty which converts the underlying textual model to a graph-like model called SGraph. To allow custom actions for additional diagram operations the communication with the client is realized in the Diagram Server that extends the Language Aware Diagram Server implementation provided by Sprotty [18].

The webview which is part of the client is separated from the extension in a different npm package and the extension makes use of the webview by loading a single JavaScript file

which is bundled through webpack. The diagrams of the graphical model are rendered in a webview panel inside VS Code which can display diverse web content within an iframe HTML element. Through dependency injection in the Diagram DI Container, it is possible to configure certain aspects of the diagram like custom actions or the elements of the model and their corresponding view and CSS styles. The Sprotty Webview Starter which is used to create the diagram with additional VS Code bindings also binds a toolbar to the diagram to allow the execution of operations on the diagram [18].

The extension combines the webview with the language server and acts as a central communication point. The language client which is part of the extension executes available binary from the language server and handles the communication by receiving and sending extended Sprotty LSP messages for textual- and graphical language features. For every VS Code extension, the base consists of an Extension Manifest in a package.json file which defines that the extension should be activated when a .erd file is opened. Upon activation commands are registered, the language client will be started and the webview will be initialized [18].

3.2 Language Server

To achieve multi-notation support inside the bigER extension, the first step is to adapt the underlying grammar of the language server because the modeling tool uses its textual representation as its main model and therefore the graphical representation always follows the elements specified textually [3]. In the following the process of extending the textual modeling language of the VS Code extension will be described and it will be shown how to extend the Xtext grammar to support various notations.

3.2.1 Grammar

The goal is to extend the existing grammar of bigER which offers a basic default ER notation with the popular notations Bachman, Chen, Crow's Foot, Min-Max, and UML. Therefore the grammar must support the characteristics of every notation. Extending an existing language is always tricky because the extension should be minimal to keep the resulting grammar as simple as possible but still sufficient enough to achieve the desired expressiveness. In the following, we will extend the grammar file EntityRelationship.xtext which is located in the language package of the project folder. Because the official Xtext documentation already offers a great overview of the specific language used for Xtext grammar [1] we assume the reader to be familiar with the basic concepts.

Listing 3.1 shows the grammar of bigER. The language offers different modeling features. First, the name of the diagram has to be defined followed by the generateOption. The generationOption allows determining if a Structured Query Language (SQL) Script should be generated out of the underlying model of the language. The notationOption was already implemented to select a favored notation but the only NotationOptionType

[1] https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.htm

available is the default value. After these options, the language allows the creation of entities and relationships. An entity has a name as ID and two optional parameters to define if it is weak or if it extends another entity. It also provides an arbitrary number of attributes. The relationship has similar values as the entity except for the difference that it cannot extend other relationships and it contains up to three RelationEntities. A RelationEntity contains the information about the involved entity followed by the cardinality. For the cardinality, it is possible to choose from two offered values from the CardinalityType or to define a custom string.

```
grammar org.big.erd.EntityRelationship with org.eclipse.xtext.common.Terminals

generate entityRelationship "http://www.big.org/erd/EntityRelationship"

Model:
        'erdiagram' name=ID ('{'
                ('generate' '=' generateOption=GenerateOption)?
                ('notation' '=' notationOption=NotationOption)?
        '}')?

        // keep old generate option until dependent code is fixed
        (generateSql?='generateSql')?

        (entities+=Entity | relationships+=Relationship)*;

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

Relationship:
        (weak?='weak')? 'relationship' name=ID ('{'
                (first=RelationEntity (('->' second=RelationEntity)
                ('->' third=RelationEntity)?)?)
                (attributes += Attribute)*
        '}')?;

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

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

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

enum GenerateOption:
    OFF='off' | SQL='sql';

enum NotationOption:
    DEFAULT='default';

enum AttributeType:
        NONE = 'none' | KEY = 'key' | FOREIGN_KEY = 'foreign-key' |
        PARTIAL_KEY = 'partial-key' | OPTIONAL = 'optional' |
        DERIVED = 'derived' | MULTIVALUED = 'multivalued';

enum CardinalityType:
        ONE = '1' | MANY = 'N';
```

Listing 3.1: Xtext Grammar of bigER

To allow different notations the first step is to define new enums for the notationOptionType. This value can be entered after the notation keyword in the textual language

and is important because it carries the information about the selected notation to the language server and the webview of the client.

```
enum NotationType:
    DEFAULT='default' | CHEN='chen' | MINMAX='minmax' | BACHMAN='bachman' |
    CROWSFOOT='crowsfoot'| UML='uml';
```

Listing 3.2: NotationType

The greatest difference in the textual representation of the notations belongs to the cardinality because every notation has certain options to define the multiplicities of a relation. In the grammar, the RelationEntity already contains the information about the cardinality but the CardinalityType only offers two values which is not enough to express one of the desired notations. The string customMultiplicity can also not be used to define the multiplicity because it has to match a certain pattern. For the notations Chen, Crow's Foot, and Bachman it is sufficient to add additional enums for the CardinalityType to describe all possible multiplicities. Chen uses the enums ONE, MANY, and MANY_CHEN to describe the cardinality of an entity. Bachman uses the enums ONE, ZERO, ONE_OR_MORE, and ZERO_OR_MORE and Crow's Foot requires the enums ONE, ZERO_OR_MORE, ONE_OR_MORE, or ZERO_OR_ONE to express all possible multiplicities. For the language of the textual editor, it is self-explanatory that for every constant the respective values will be used.

In the grammar, there are no restrictions on which enums of the CardinalityType are allowed for a certain notation. For this, a validator comes into play which we will explain later. This validator ensures that every notation only uses the correct types.

```
enum CardinalityType:
        NONE = 'NONE' | ONE = '1' | MANY = 'N' | MANY_CHEN = 'M' |
        ZERO = '0' | ONE_OR_MORE = '1+' | ZERO_OR_MORE = '0+' | ZERO_OR_ONE = '?';
```

Listing 3.3: CardinalityType

The cardinality of the notations Min-Max and UML cannot just be expressed through a simple value represented by the CardinalityType and UML also allows additional information about the type of aggregation and role of an entity. To be able to express both notations inside a RelationEntity two additional terminals are required one for each notation.

```
RelationEntity:
        target=[Entity] ('['
                (cardinality=CardinalityType | customMultiplicity=STRING |
                minMax=MinMax | uml=Uml)
        ']')?(role=STRING)?;
```

Listing 3.4: RelationEntity

The cardinality of the Min-Max notation consists of a nonoptional number to define the lower bound followed by a comma. After the comma, there are two possibilities to

define the upper bound. Again, it is either possible to define a simple number or to use the asterisk symbol to describe an arbitrary upper bound for the entity. When entering the upper bound, it is required that it is as great or greater as the lower bound. This requirement cannot be expressed through the grammar itself and therefore the validator checks if this condition is not violated.

```
terminal MinMax:
         ('0'..'9')+','(('0'..'9')+|'□');
```

Listing 3.5: Min-Max terminal

The integration of the UML notation is the most complex one out of the five notations. At first, it is possible to define the aggregation type of an entity as compositing or aggregation. After the aggregation type, a required number must be entered to define the lower bound of the cardinality. In contrast to the Min-Max notation, it is not necessarily required in UML to define an upper bound but again there are two similar possibilities to Min-Max to do so. After entering two periods either a number or the asterisk symbol can be used as the upper bound. Additional validations of the model again check that the upper bound is either as great or greater as the lower bound because such conditions are impossible to define by the grammar itself. Besides these options, it is also possible to only use the asterisk symbol to define that zero or more entities of a certain kind are involved in the relationship.

```
terminal Uml:
         (('comp ')?|('agg ')?)((('0'..'9')+(('..'('0'..'9')+)?|('..''□')?))|'□');
```

Listing 3.6: UML terminal

3.2.2 Validation

After the enhancement, the grammar can express the characteristics of five di□erent notations at once and so the expressiveness has to be restricted to match only the selected notation. As already mentioned earlier, the grammar cannot express certain conditions for example that one particular value is greater than another value. For this, additional validations are required. The grammar of a language has an impact on what is required for a document or semantic model to be valid. Xtext takes care of this and o□ers automated and custom validation. The syntactical correctness of any textual input is validated automatically by the parser but it is as well possible to specify additional constraints specific to the model. For a custom validation, the Xtext language generator provides two Java classes. The first one is a generated abstract class that extends the AbstractDeclarativeValidator and it is used to register the EPackages for which this validator introduces constraints. The second Java class is a generated subclass of the abstract class and this class is the right place for additional validations [14]. In the case

of bigER the abstract validator class is called AbstractEntityRelationshipValidator and the subclass has the name EntityRelationshipValidator and is placed inside the source folder of the language server.

For the notations Bachman, Chen, and Crow's Foot the validation is quite simple because these three notations only use the CardinalityType of the grammar to describe the multiplicity. For those notations, the validator ensures that only values are used which are suitable for the selected notation.

```
def checkBachmanCardinality(RelationEntity relationEntity, Relationship relationship,
EStructuralFeature feature) {

        if (relationEntity !== null && (relationEntity.cardinality === null ||
        relationEntity.customMultiplicity !== null ||
        relationEntity.minMax !== null || relationEntity.uml !== null ||
        relationEntity.cardinality === CardinalityType.MANY ||
        relationEntity.cardinality === CardinalityType.MANY_CHEN ||
        relationEntity.cardinality === CardinalityType.ZERO_OR_ONE)) {
                info('''Wrong cardinality. Usage: [0],[0+],[1] or [1+]''',
                relationship, feature)
        }
}
```

Listing 3.7: Bachman validation

```
def checkChenCardinality(RelationEntity relationEntity, Relationship relationship,
EStructuralFeature feature){
        if(relationEntity !== null && (relationEntity.cardinality === null ||
        relationEntity.customMultiplicity !== null ||
        relationEntity.minMax !== null ||
        relationEntity.uml !== null ||
        relationEntity.cardinality === CardinalityType.ZERO ||
        relationEntity.cardinality === CardinalityType.ONE_OR_MORE ||
        relationEntity.cardinality === CardinalityType.ZERO_OR_MORE ||
        relationEntity.cardinality === CardinalityType.ZERO_OR_ONE)){
                        info('''Wrong cardinality. Usage: [1],[N] or [M]''',
                        relationship, feature)
        }
}
```

Listing 3.8: Chen validation

```
def checkCrowsFootCardinality(RelationEntity relationEntity, Relationship relationship,
EStructuralFeature feature){
        if(relationEntity !== null && (relationEntity.cardinality === null ||
        relationEntity.customMultiplicity !== null ||
        relationEntity.minMax !== null ||
        relationEntity.uml !== null ||
        relationEntity.cardinality === CardinalityType.MANY_CHEN ||
        relationEntity.cardinality === CardinalityType.MANY ||
        relationEntity.cardinality === CardinalityType.ZERO)){
                        info('''Wrong cardinality. Usage: [1],[0+],[1+] or [?]''',
                        relationship, feature)
        }
}
```

Listing 3.9: Crow's Foot validation

For the Min-Max notation, the validator first checks if the textual input uses the right element in the grammar and not just a simple CardinalityType or a custom string for example. If the input is valid the validation verifies that the entered upper bound is

as great or greater than the lower bound. This check is not required when the asterisk symbol is used to define an arbitrary upper bound.

```
def checkMinMaxCardinality(RelationEntity relationEntity, Relationship relationship,
EStructuralFeature feature) {
        if (relationEntity === null) {
                return
        }
        if (relationEntity.minMax === null) {
                info('''Wrong cardinality.Usage: [min,max] or [min,□]''',
                relationship, feature)
        }else{
                if(relationEntity.minMax.toString.contains(",")){
                        if(relationEntity.minMax.toString.split(",").length == 2){
                                var fistNumber =
                                relationEntity.minMax.toString.split(",").get(0)
                                var secondNumber =
                                relationEntity.minMax.toString.split(",").get(1)

                                if (fistNumber.matches("\\d+") &&
                                secondNumber.matches("\\d+") &&
                                Integer.parseInt(fistNumber) >
                                Integer.parseInt(secondNumber)) {
                                        info('''Wrong cardinality. Usage: [min,max]
                                        min <= max''', relationship, feature)
                                }
                        }else{
                                info('''Wrong cardinality.Usage: [min,max] or
                                [min,□]''',relationship, feature)
                        }
                }else{
                        info('''Wrong cardinality.Usage: [min,max] or [min,□]''',
                        relationship, feature)
                }
        }
}
```

Listing 3.10: Min-Max validation

For UML, the same checks for the boundaries are made. The UML notation also o□ers the opportunity to declare an entity of a relationship as composition or aggregation and for this, the validator checks that only one entity of a relationship has the optional information about the aggregation. For the RelationEntity of the grammar, it is also possible to define a role independent of the selected notation. A role is only allowed for UML and so the validator verifies that for every other notation no role is used. Whenever an error within the textual input is discovered, the user gets informative feedback in form of a usability message as they type.

```
def checkUmlCardinality(RelationEntity relationEntity, Relationship relationship,
EStructuralFeature feature) {
        if (relationEntity === null) {
                return
        }
        if (relationEntity.customMultiplicity !== null ||
        relationEntity.minMax !== null ||
        (relationEntity.uml === null && relationEntity.cardinality !==
        CardinalityType.ZERO && relationEntity.cardinality !==
        CardinalityType.ONE)) {
                info('''Wrong cardinality.Usage: [num],[min..max] or [min..□]''',
                relationship, feature)
        }
        if (relationEntity.uml.contains("..")) {
                var cardinality = relationEntity.uml

                // remove type (agg|comp)
                if (relationEntity.uml.contains(" ")) {
                        cardinality = relationEntity.uml.split(" ").get(1)
                }
                var numbers = cardinality.split("\\.\\.")
                if (numbers.length === 2) {
                        if (numbers.get(0).isEmpty || numbers.get(1).isEmpty) {
                                info('''Wrong cardinality.Usage: [num],[min..max] or
                                [min..□]''', relationship, feature)
                        }
                        var n1 = numbers.get(0)
                        var n2 = numbers.get(1)
                        if (n1.matches("\\d+") && n2.matches("\\d+") &&
                        Integer.parseInt(n1) > Integer.parseInt(n2)) {
                                info('''Wrong cardinality. Usage: [min..max]
                                min <= max''', relationship, feature)
                        }
                } else {
                        info('''Wrong cardinality.Usage: [num],[min..max] or [min..□]''',
                        relationship, feature)
                }
        }
}
```

Listing 3.11: Min-Max validation

3.2.3 Graph generation

When Xtext has parsed the textual input into an in-memory representation a generator inside the language server transforms the EMF model of the modified document to a Sprotty diagram model (SGraph) that describes the associated diagram. This is a very important step because it provides the opportunity to have an impact on the transformation. The bigER tool contains already the ERDiagramGenerator which implements the IDiagramGenerator interface provided by Sprotty and this generator adds entities and relationships to the graph.

For the webview of the client, additional values inside the SGraph are required to transport the information about the selected notation from the language server to the webview. For this, a new ERModel was introduced which extends the SGraph from the Sprotty framework. The ERModel has an extra value for the notation but this value is only used for the toolbar of the webview and so the extension of the SGraph is not enough to enable a notation-specific rendering in the webview. On the client part of Sprotty, every element in the graph will be rendered by a corresponding view. Such a view only receives the specific element which has to be rendered. The only elements of the diagram

which will be visualized differently depending on the chosen notation are the SEdges. The NotationEdge extends the SEdges and adds further information about the selected notation, the cardinality of the relationship, or an optional role to the Edge. When creating the NotationEdge in the Generator all these values will be set and therefore enables the possibility to render every edge depending on the selected notation.

```
@Accessors
class ERModel extends SGraph {
        String name
        String generateType
        String notation

        new() {
        }
        new((ERModel)=>void initializer){
                initializer.apply(this)
        }
}.
```

Listing 3.12: ERModel

```
@Accessors
class NotationEdge extends SEdge {
        Boolean isSource
        String notation
        Boolean showRelationship
        String relationshipCardinality
        String umlRole

        new() {
        }

        new((NotationEdge)=>void initializer) {
                initializer.apply(this)
        }
}
```

Listing 3.13: NotationEdge

Not every notation requires to render nodes for relationships like Crow's Foot. For those cases, only one edge between two entities will be added to the graph instead of two. An edge always carries the cardinality information for one entity involved in a relationship and therefore it is problematic when only one edge is added to the graph. To overcome this problem the cardinality information from two entities will be combined into one and the webview will split these values again to render each end of the edge corresponding to the given cardinality.

3.2.4 Notation Handler

The toolbar of the diagram view of bigER, which will be created for the webview in the following part, should allow choosing the favored notation by a dropdown. For this, a custom action has to be sent from the client to the language server. In Sprotty, all kinds of actions are received in the ActionDispatcher and delegated to a respective

ActionHandler where they are converted to commands. Such commands are passed to the CommandStack by the dispatcher to update the underlying model [12]. bigER uses its textual representation as its main model and therefore the graphical representation always follows the elements specified textually [3]. For customized actions, bigER uses the ERDiagramServer which extends the LanguageAwareDiagramServer provided by Sprotty Xtext. The ERDiagramServer determines which ActionHandler hast to be used for a certain kind of action. Whenever the server receives a ChangeNotationAction, the NotationHandler changes the value for the selected notation inside the textual language, and therefore an SGraph with the new notation will be generated. This newly created SGraph will be sent to the webview where the rendering changes depending on the chosen notation.

3.3 Webview

After the enhancement of the language server, the SGraph contains additional information by which the webview can render the edges of the graph in the diagram view differently depending on the selected notation. This chapter describes the necessary extension in the webview to enable the rendering of five different notations in the diagram view.

3.3.1 Custom View for Edges

Sprotty can be customized by using dependency injection and diagrams are implemented by creating a DI container. With this container, it is possible to define the bindings to services and the mapping of SModel elements to their corresponding view classes [19].

Before creating a custom view to render edges depending on the selected notation a counterpart to the NotationEdge on the server side has to be implemented in the webview as well to be able to receive the additional information inside the edges of the graph. In the model.ts file of the webview package, the NotationEdge is defined as an export class which offers the same values as the NotationEdge on the server side. After doing so it is possible to create a custom view for the NotationEdge which handles the rendering. The already mentioned DI container with the mapping of the elements to their corresponding view is located in the di.config.ts file. Here it is required to add a further mapping of the NotaitonEdge to the NotationEdgeView which will be created shortly.

In the bigER extension custom views were already created for entities and relationships inside the views.tsx file. For the creation of the NotationEdgeView two methods are required to be defined. The first one is the render method. This method is required to make certain decisions about the elements involved in the rendering. The first check determines if both ends of an edge need to be rendered. This is only the case for Crow's Foot and UML because for those two notations the entities in the diagram are directly connected without a relationship node in between. Next, it has to be decided if the label of an edge should be rendered. The label contains the name of the relationship specified in the textual editor. This decision is required for the Bachman and Crow's Foot notation

as they do not show the name of the relationship on the edge. For this, the rendering of the child elements of the SEdge is left out. The only child of a SEdge in the bigER extension is a SLabel which again has its own view. The second method required for the NotationEdgeView is the renderEndOfEdge-method. As the name already suggests, this method is used for the notation-specific rendering of the end of an edge which means for example for Bachman, a circle will be rendered if the cardinality equals one in the textual input. If both ends have to be rendered in a specific way the method has to be called twice because it only deals with one end at a time. The notations Chen and Min-Max are not covered in this method because they do not need an additional rendering of their ends because a normal line for an edge with a cardinality labels is sufficient.

For the notations Bachman, Crow's Foot and UML an SVG will be rendered on the end of an edge. SVG views are suited very well for the rendering as they allow a high scalability which is often one of the shortcomings of many other existing modeling tools [3]. Creating those SVGs is tricky because to realise a certain design basic shapes like circles or lines have to be combined to form the desired shape. The transformation is also very important when creating an SVG for bigER because when an entity or a relationship is moved around the SVGs on the edges have to move and rotate as well to always maintain the same relative position. Listing 3.14 shows an example for the Crow's Foot notation with the cardianlity ONE_OR_MORE to demonstrate the content of an SVG.

```
<svg>
    <line x1={point.x + 24} y1={point.y + 11} x2={point.x + 24} y2={point.y - 11}
          transform={`rotate(${this.angle(point, next)} ${point.x} ${point.y})`}/>
    <line x1={point.x + 17} y1={point.y} x2={point.x} y2={point.y + 11}
          transform={`rotate(${this.angle(point, next)} ${point.x} ${point.y})`}/>
    <line x1={point.x + 17} y1={point.y} x2={point.x} y2={point.y - 11}
          transform={`rotate(${this.angle(point, next)} ${point.x} ${point.y})`}/>
</svg>
```

Listing 3.14: Min-Max terminal

3.3.2 Toolbar

bigER is a hybrid modeling tool which means it is either possible to change the underlying model through the textual editor or the graphical editor. Changing the favored notation is currently only possible by defining the notation in the textual language. To be able to change the notation in the graphical view as well a drop-down in the toolbar will be integrated. In the extension, the webview starter already binds a toolbar to execute actions on the diagram. Whenever an action is sent or received from the extension the diagram server which acts as a model source is responsible for handling those actions.

In bigER custom actions were already created for example to add entities via the diagram view and those actions are defined in the actions.ts file in the package of the webview. On the server side, the NotationHandler was already created which receives ChangeNotationActions. The ChangeNotationAction has to be added on the client side in the actions.ts file as well. It contains only a notation value to transport the information about the selected notation from the client to the server.

```
export interface ChangeNotationAction {
    kind: typeof ChangeNotationAction.KIND
    notation: string
}
```

Listing 3.15: ChangeNotationAction

The integration of the drop-down for the notations is a good opportunity to change the appearance so that it better blends in into VS Code. With the Webview UI Toolkit[2] for VS Code, extensions can easily be created that appear like the editor itself. The toolkit provides HTML-elements like for example a vscode-button that already o□ers the design of VS Code. The toolkit also o□ers the right element for a drop-down. Via vscode-dropdown, the drop-down can be created and via vscode-option a selectable element can be defined within the drop-down. An EventHandler listens to a change event of the drop-down. When changing the notation the handler extracts the value for the notation out of the DOM and passes the value to the ActionDispatcher as a ChangeNotationAction. This action will be sent through the extension to the language server where the NotationHandler changes the notation value in the textual language with the values provided through the ChangeNotationAction.

[2] https://github.com/microsoft/vscode-webview-ui-toolkit

CHAPTER 4

Showcase

After the implementation of various ER notations in the previous chapter the bigER modeling tool offers the opportunity to create ER diagrams for Bachman, Chen, Crow's Foot, Min-Max, and UML. In this chapter, the result of the integration of the multi-notation support will be shown.

4.1 Bachman

The ER concept for the Bachman notation is shown in table 4.1. Four options are available to define the cardinality. After the name of the entity either 0 for zero, 0+ for zero or more, 1 for one, and 1+ for one or more follows.

Table 4.1: Bachman Syntax

ER Concept	Bachman
A[0]	zero
A[0+]	zero or more
A[1]	one
A[1+]	one or more

If the user has selected the Bachman notation and enters a valid input that lies within the grammar of the textual language but does not match the notation a usability message will be created from the validator. The usability message informs the user about the available options for the Bachman notation.

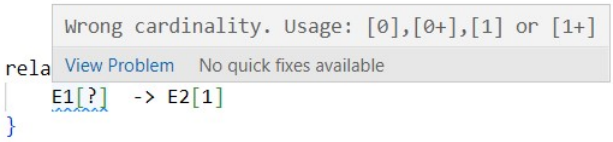

Figure 4.1: Bachman usability message

The rendering in the diagram view is different for every cardinality option. Figure 4.2 shows the cardinalities zero for E1 and zero or more for E2 and figure 4.3 shows one for E1 and one or more for E2.

Figure 4.2: Bachman zero and zero or more

Figure 4.3: Bachman one and one or more

4.2 Chen

Table 4.2 shows how to define the cardinality for the Chen notation and there are three options available to choose from. After entering the name of the notation the CardinaltyType 1 for one and N or M for many follows.

Table 4.2: Chen Syntax

ER Concept	Chen
A[1]	one
A[M]	many
A[N]	many

For an input that lies within the grammar but does not match the Chen notation, a usability message with the available cardinality options will be created.

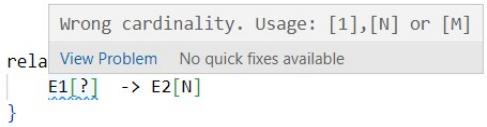

Figure 4.4: Chen usability message

The rendering in the diagram view is quite simple because no additional rendering for the end of the edges is needed. It is sufficient to only show the cardinality label on the edges.

Figure 4.5: Chen cardinality

4.3 Crow's Foot

The Crow's Foot notation offers four options to define the cardinality which are shown in table 4.3. After the name of the entity, the multiplicity can either be defined as 1 for one, 0+ for zero or more, 1+ for one or more, and ? for one or zero.

Table 4.3: Crow's Foot Syntax

ER Concept	Crow's Foot
A[1]	one
A[1+]	one or more
A[0+]	zero or more
A[?]	zero or one

Whenever the user inputs something that lies within the grammar but does not match the Crow's Foot notation a usability message with the available cardinality options will be created.

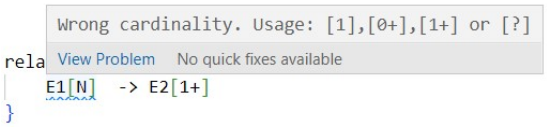

Figure 4.6: Crow's Foot usability message

Every of the four cardinality options offers a different rendering of the edges and in contrast to other notations a relationship node will not be shown. Figure 4.7 shows the cardinalities one for E1 and one or more for E2 and figure 4.8 shows zero or more for E1 and one or zero for E2.

Figure 4.7: Crow's Foot one and one or more

Figure 4.8: Crow's Foot zero or one and zero or more

4.4 Min-Max

For the Min-Max notation it is possible to define a lower and an upper bound for the cardinality. The min and the max values are separated by a comma and both values are restricted in a way that the max value has to be equal to or greater than the min value. Table 4.4 shows the concrete syntax of the Min-Max cardinality.

Table 4.4: Min-Max Syntax

ER Concept	Min-Max
A[min,max]	min <= max
A[min,*]	* for arbitrary upper bound

Two different usability messages inform the user about an incorrect input. Whenever the user inputs something that lies within the grammar but does not match the Min-Max notation a usability message informs the user about the available options. The second usability message will be displayed when the user has chosen a lower bound which is greater than the upper bound.

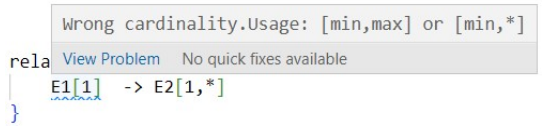

Figure 4.9: Min-Max wrong input usability message

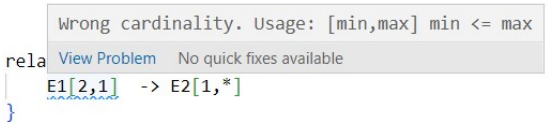

Figure 4.10: Min-Max boundaries usability message

As for Chen the rendering of the Min-Max notation is quite simple. Only the cardinality enclosed in brackets will be rendered on the edges.

Figure 4.11: Min-Max cardinality

4.5 UML

UML is the most complex notation supported by the bigER extension. As shown in table 4.5 the cardinality can either be defined with a simple number or a asterisk symbol but it is also possible to define a lower and an upper bound just like for MIN-MAX. The only difference between MIN-MAX and UML is the double period between the two boundaries instead of a comma. Despite the cardinality, the user is also able to define an entity as aggregation with the keyword agg or as composition with the keyword comp. In UML it is also possible to enter a role for an entity which is done with a string in single quotes after the information about the cardinality.

Table 4.5: UML Syntax

ER Concept	Min-Max
A[num]	arbitrary number
A[*]	zero or more
A[min..max]	min <= max
A[min..*]	* for arbitrary upper bound
A[agg min..max]	agg for aggregation
A[comp min..max]	comp for composition
A[min..max]'Role'	Definition of a role

In UML three different usability messages inform the user about invalid input. Figure 4.12 shows the message when a wrong cardinality is entered which is not allowed in UML. A message like on figure 4.13 will be displayed when the lower bound is bigger than the upper bound and figure 4.14 shows a usability message that will be created when for more than one entity a aggregation information was entered.

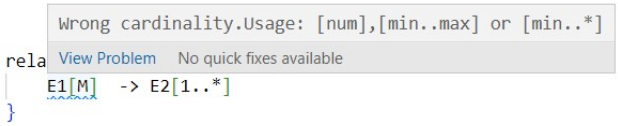

Figure 4.12: UML wrong input usability message

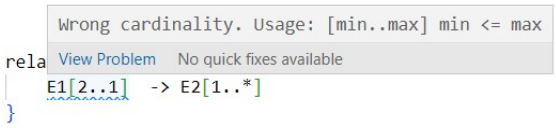

Figure 4.13: UML boundaries usability message

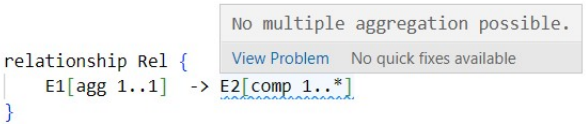

Figure 4.14: UML multiple aggregation usability message

In the bigER extension only for the UML notation, it is allowed to define a role so for very other supported notation an error message will be displayed when a role is entered.

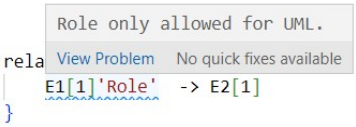

Figure 4.15: Role not allowed usability message

For a relationship with two entities involved the rendering for the UML notation consists mostly of rendered labels. The role will be displayed underneath the cardinality and the name of the relationship will be shown on the edge. Additionally, when an aggregation information is entered a diamond-shaped end of an edge will be rendered.

Figure 4.16: UML cardinality

Figure 4.17: UML aggregation

Figure 4.18: UML composition

For ternary relationships there will be a relation node displayed and the label on the edges with the name of the relation will not be shown.

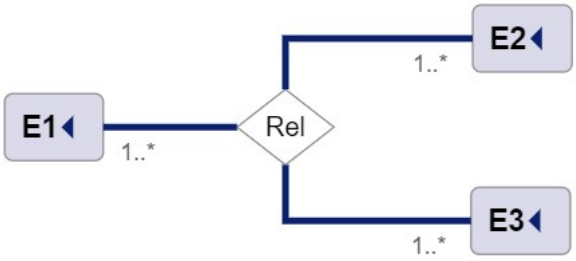

Figure 4.19: UML ternary relationship

4.6 Toolbar

The toolbar of the graphical editor was redesigned to blend in more into the appearance of VS Code and to give an alternative to the textual editor to switch between notation styles. The notation can be switched by a context menu which appears when the mouse cursor hovers over the selected notation.

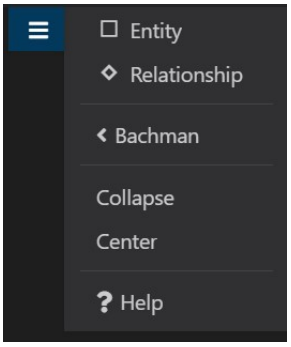

Figure 4.20: bigER toolbar

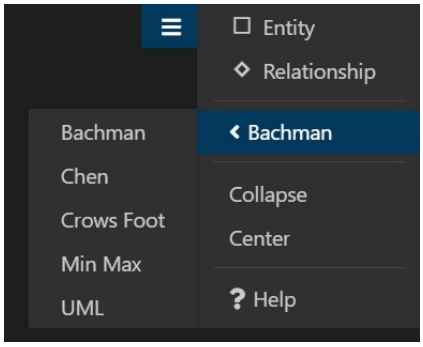

Figure 4.21: bigER toolbar notations

To increase the usability and to help the user to understand which cardinality options are available for a selected notation it is possible to hover over the help button which will show a separate panel with a usage information. Figure 4.22 shows as an example the usage information for the Bachman notation.

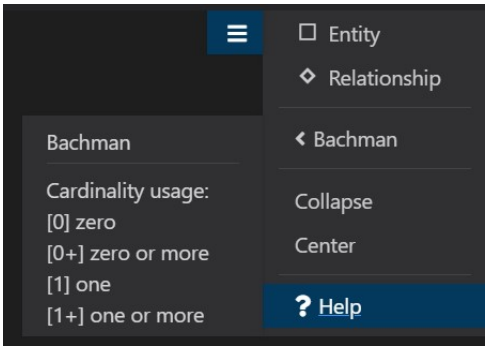

Figure 4.22: bigER toolbar notations

CHAPTER 5

Discussion

In this chapter, observations during the development phase of this thesis will be discussed and afterwards the bigER tool will be compared with its main competitors.

5.1 Observations

LSP, Sprotty, and Xtext provided a great infrastructure for the integration of the multi-notation support for the textual and the graphical editor as well but on the client side, there are still view shortcomings in the rendering of the edges for the UML notation. For UML additional labels for the cardinality and an optional role will be rendered. These labels have a fixed position relative to the relation edge which can lead to view problems. One problem is that it is hard to determine the position of the cardinality label independent of the user input for the multiplicity. A very large number can lead to an overlapping of elements in the diagram which is shown in figure 5.1. The positioning is especially hard for the entities rendered on the right end of an edge. The same problem with the positioning consists for the UML role. It is hard to find an algorithm for the calculation of the positioning that creates a satisfying solution for short and long inputs simultaneously.

Figure 5.1: UML overlapping labels

Another issue with overlapping elements can happen when the label for the name of a relationship or the cardinality is rendered overlapping with the edges in the graph which

can be seen in figure 5.2. The user can manually solve the problem of the overlapping by moving the entities but this is not a sufficient solution.

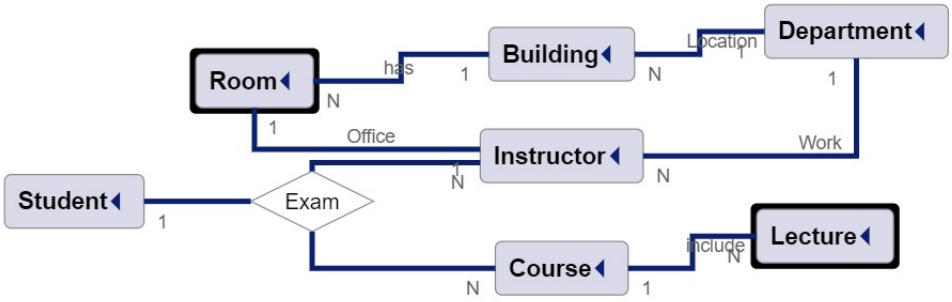

Figure 5.2: Issue with overlapping elements

The last problem in the diagram view which was found during the development is that the rendering of the edges seems to work only in a certain area of the diagram view. In figure 5.3 the entity E1 was dragged to the left side of the diagram view until the circle on the end of the edge started to disappear. It is not clear which component involved in the rendering causes this problem. It appears this phenomenon is in relation to the size of the initially rendered diagram. Only when the user tries to move the elements further apart as the initial rendering this problem occurs.

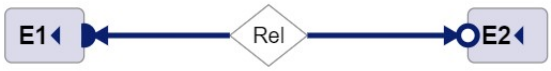

Figure 5.3: Restricted area for rendering

5.2 Comparison with competing modeling tools

Because of the popularity of entity-relationship modeling a wide variety of tools exists. In context of VS Code two relevant extensions beside bigER are available named ERD Editor and ERD Preview. ERD Editor focuses only an a graphical editor and the supported ER notation is Crow's Foot. ERD Preview is like bigER a hybrid modeling tool with a textual and a graphical editor and it also only uses the Crow's Foot notation. This makes bigER the only hybrid modeling tool available in the VS Code ecosystem that provides different ER notation including the Crow's Foot notation. bigER has currently the fewest downloads with approximately 800 and ERD Editor the most with almost 40000 downloads. In comparison, bigER offers a better feature set and great usability. Until now it is not officialy released but once this is done, we can be confident that the number of downloads will increase significantly.

The first major release of bigER will be introduced to the ER community in the course of the ER2022 the 41st International Conference on Conceptual Modeling. The introduction

paper will provide an overview of the centerpieces of the bigER modeling tool and the first two major extensions. These two extensions include the multi-notation support realized in this thesis and an improved edge routing through the libavoid library. bigER is the first freely available hybrid modeling tool for VS Code and therefore we believe that it provides a great benefit to the ER community [20].

CHAPTER 6

Conclusion

6.1 Summary

Summing up, in the background of this thesis we analyzed the importance of entity-relationship modeling and the differences between various popular ER notations as they can express different relationship constraints. Furthermore, we investigated the technologies LSP, Sprotty, and Xtext which allow the creation of a hybrid modeling tool with textual and graphical editors for modern lightweight IDEs like VS Code. We discovered that a Sprotty enhanced language server is very well suited to support multiple notations within one grammar. The Xtext framework offers a validator that can be used to restrict the expressiveness of a grammar that supports various notations to only match one selected notation. We also found out that the elements SGraph and SEdge from the backend side of Sportty can be extended with additional values to transport the information about the chosen notation, cardinality, role, and aggregation from the server to language-agnostic clients. On the client side, Sprotty uses views for every element in the semantic model and this created the opportunity for the implementation of customized rendering of notation-specific edges in the graphical view.

6.2 Outlook

With the completion of the implementation of the multi-notation support, we contributed to the release version of bigER. However, the work for further releases are already in development. For example bigER offers the feature for SQL Code generation for its default basic notation but for the newly integrated notations this feature is not supported yet. The work to support the SQL generations for the new notations is part of a recently started bachelor thesis and the functionality will be integrated into bigER in a future release.

List of Figures

List of Tables

Bibliography

[1] Alexander P. Pons, Peter Polak, and Joel Stutz. Evaluating the teaching e□ectiveness of various data modeling notations. *The Journal of Computer Information Systems*, page 78, 2005/2006.

[2] Andreas Gadatsch. *Datenmodelierung für Einsteiger*. Springer Verlag, 2017.

[3] P.-L. Glaser and D. Bork. The biger tool - hybrid textual and graphical modeling of entity relationships in vs code. *in: 25th International Enterprise Distributed Object Computing Workshop, EDOC Workshop 2021*, page 337–340, 2021.

[4] Language server protocol. `https://docs.microsoft.com/en-us/visualstudio/extensibility/language-server-protocol?view=vs-2022`. accessed: 2022-08-19.

[5] Ramez Elmasri and Shamkant B. Navathe. *FUNDAMENTALS OF Database Systems*. ADDISON WESLEY PUB CO INC, 2015.

[6] P. P.-S. Chen. The entity-relationship model-toward a unified view of data. *ACM Trans. Database Syst.*, 1:9–36, 1976.

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

[8] Robert J. Muller. *Database design for smarties*. Morgan Kaufmann, 1999.

[9] John Holt. *Uml For Systems Engineering: Watching the Wheels (Computing and Networks)*. Institution of Engineering and Technology, 2005.

[10] Jan Köhnlein. Eclipse sprotty - diagrams in the web. `https://www.eclipse.org/community/eclipse_newsletter/2018/october/sprotty.php`. accessed: 2022-08-21.

[11] Jan Köhnlein. Sprotty – a web-based diagramming framework. `https://www.typefox.io/blog/sprotty-a-web-based-diagramming-framework`, 2017. accessed: 2022-08-19.

[12] Jan Köhnlein. Architectural overview. https://github.com/eclipse/sprotty/wiki/Architectural-Overviewk, 2019. accessed: 2022-08-21.

[13] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[14] Xtext documentation. https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#validation. accessed: 2022-06-07.

[15] Visual studio code getting started. https://code.visualstudio.com/docs. accessed: 2022-08-19.

[16] Stack overflow developer survey results. https://insights.stackoverflow.com/survey/2019. accessed: 2022-08-19.

[17] Visual studio code extension api. https://code.visualstudio.com/api. accessed: 2022-08-19.

[18] P.-L. Glaser. Developing sprotty-based modeling tools for vs code. *https://model-engineering.info/publications/theses/thesis-glaser.pdf*, 2022.

[19] Jan Köhnlein. Dependency injection. https://github.com/eclipse/sprotty/wiki/Dependency-Injection, 2019. accessed: 2022-08-21.

[20] P.-L. Glaser, G. Hammerschmied, V. Hnatiuk, and D. Bork. The biger modeling tool. *41st International Conference on Conceptual Modeling (ER 2022)*, page in press, 2022.