TU WIEN Informatics

Frontend-only browser-based modeling tools

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

David Jäger, BSc
Matrikelnummer 11723775

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ass. Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork
Mitwirkung: Dipl.-Ing. Dr.techn. Philip Langer
Dr.techn. Martin Fleck

Wien, 3. Mai 2024

David Jäger | Dominik Bork

TU WIEN Informatics

Frontend-only browser-based modeling tools

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

David Jäger, BSc
Registration Number 11723775

to the Faculty of Informatics

at the TU Wien

Advisor: Ass. Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork
Assistance: Dipl.-Ing. Dr.techn. Philip Langer
Dr.techn. Martin Fleck

Vienna, 3rd May, 2024

David Jäger | Dominik Bork

Erklärung zur Verfassung der Arbeit

David Jäger, BSc

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, 3. Mai 2024

David Jäger

Acknowledgements

First and foremost, I would like to thank my supervisor Dominik Bork, for supporting me with advice and patience. His quick and insightful feedback helped me a lot throughout the course of my thesis.

Furthermore, I would like to extend my appreciation to Philip Langer and Martin Fleck, who have been pivotal during the development phase of my thesis by always being helpful in case of issues.

Most importantly, I would like to thank my family, especially my parents, Nicola and Ewald Jäger, who have supported me in every phase of my life. Their constant encouragement has been crucial in enabling me to complete my studies successfully.

To my friends, especially Max, Simon, and Dominik, whose companionship and support during my time at TU Wien have been integral to this journey.

Lastly, I would like to thank Laura, who always helped me whenever I faced difficulties. Without her presence, this achievement would not have been possible.

Kurzfassung

In den letzten Jahren hat sich ein Trend weg von funktionsreichen Integrated Development Environments (IDE) hin zu leichtgewichtigen Web-Clients abgezeichnet. Um diesen Wandel zu ermöglichen, wurde für textbasierte Editoren das Language Server Protocol (LSP) entwickelt, während für grafische Editoren eine Erweiterung des LSP, das Graphical Language Server Platform (GLSP), eingeführt wurde. Die Modellverwaltung für grafische Editoren wurde jedoch weiterhin von Java-Servern übernommen, einschließlich der Erstellung von Metamodellen und der Laufzeitverwaltung von Modellen.

Diese Arbeit verfolgt das Ziel, den nächsten Schritt für webbasierte grafische Modell-Editoren zu gehen und die Modellverwaltung auf einen reinen TypeScript-Technologiestapel umzustellen. Dazu werden die Funktionalitäten des Next-Generation Language Frameworks Langium erforscht und um eine Modellserver-API erweitert, die es modellorientierten Clients ermöglicht, auf den Abstract Syntax Tree (AST) zuzugreifen, der von Langium erstellt wird und den aktuellen Zustand eines Modells darstellt. Zudem wird eine neue, in TypeScript native Grammatiksprache konzipiert, um eine TypeScript-native Lösung für die Definition von Metamodellen zu bieten.

Um die Modellserver-API mit der TypeScript-basierten Grammatiksprache zu verbinden, wird ein Generator erstellt, der die gesamte Modellverwaltung aufbaut. In Java-basierten Umgebungen wird das Metamodell für die Komponente zur Modellverwaltung üblicherweise mit dem EMF (Eclipse Modeling Framework) Ecore Metamodell erstellt. Um den Übergang von Ecore zu der auf TypeScript basierenden Grammatiksprache zu erleichtern, wird in die Implementierung des Generators ein Mechanismus integriert, der die Erstellung der auf TypeScript basierenden Grammatikdefinition aus dem Ecore Metamodell ermöglicht.

Die Evaluierung dieser Arbeit wird in zwei Teilen durchgeführt: Zuerst wird die TypeScript-basierte Grammatiksprache durch einen Vergleich mit dem weit verbreiteten Ecore-Metamodell bewertet. Weiters, werden zwei State-of-the-Art Modell-Editoren, die GLSP nutzen, mithilfe des Generators nachgebaut. Dadurch kann bewertet werden, ob die Erstellung des Metamodells sowie der Modellverwaltung durch den Generator korrekt funktioniert.

Abstract

In recent years, a shift from feature-rich Integrated Development Environments (IDE) to lightweight web clients could be observed. In order to be able to make that shift, for textual editors, the Language Server Protocol (LSP) has been created, while for graphical editors, an enhancement of the LSP has been introduced in the Graphical Language Server Platform (GLSP). However, the model management for graphical editors has still been handled by heavy-weight Java model servers. This includes both the creation of metamodels and the runtime handling of models.

This thesis aims to make the next step toward web-based graphical model editors and shift the model management to a TypeScript-only technology stack. For this, the functionalities of the next-generation language framework Langium are explored and extended by a model server API. This enables model-oriented clients to access the Abstract Syntax Tree (AST), which is created by Langium and holds the current state of a model. Furthermore, a new TypeScript native grammar language is conceptualized to provide a TypeScript native solution to define metamodels.

To combine the model server API with the TypeScript-based grammar language, a generator is created that sets up the entire model management component. Typically, in Java-based environments, the metamodel for the model management component is created using the EMF Ecore metamodel. Therefore, to ease the transition from Ecore to the TypeScript-based grammar language, a mechanism to create the TypeScript-based grammar definition from the Ecore metamodel is added to the implementation of the generator.

The work in this thesis is evaluated in two parts: First, the TypeScript-based grammar language is evaluated by comparing it with the widely used Ecore metamodel. Second, two state-of-the-art modeling tools, which utilize GLSP, are rebuilt to evaluate the generator by creating the metamodel and model management using the generator's capabilities.

Contents

CHAPTER 1

Introduction

This chapter serves to introduce the problem that will be addressed in the thesis and explain the reasons behind its significance. The chapter will establish the specific research questions and the methodology employed. Finally, a brief overview of the remaining thesis sections will be provided.

1.1 Problem Statement and Motivation

In recent years much effort has been made to move from rich client applications to web clients. One of the reasons for this trend is the tight coupling of domain-specific language features into the feature-rich Integrated Development Environment (IDE). To decouple this tight integration, the Language Server Protocol (LSP), which enables language-agnostic clients to communicate with language-specific servers [REIWC18], has been implemented. By using this protocol the client can be a very lightweight editor, while syntax highlighting, code completion, and cross-references are handled in the language server. While this protocol works very well for textual editors, graphical editors were still very tightly coupled to the IDE. As a result, the Graphical Language Server Platform (GLSP) was introduced to extend the LSP with a graphical notation, allowing graphical editors to follow this trend.

Recently, another framework, named Langium [Lan], which is a textual language framework written in TypeScript, has been released. This framework offers built-in support for a lot of different complex problems when dealing with model management including language parsing, semantic models, and cross-references. While this framework provides a lot of important features, it is still only a textual language framework. Hence, this framework only offers solutions for textual representations of a model. Therefore, the question arises, whether the features of Langium can be made available for model-oriented clients, which will be one of the main foci of this thesis.

Furthermore, an important topic in language engineering is the definition of Domain-Specific Modeling Languages (DSL), which traditionally is done by the creation of a grammar [JBF11]. In recent years, the approach to creating DSLs shifted to first create the metamodel of a DSL. Nowadays, the creation of a metamodel and the grammar can be done in parallel, as the metamodel can be created from the grammar and vice versa. An example of a newer approach is Xtext [Foud] which generates the metamodel from the grammar. While this eases the development of DSLs, it is Java-based, and therefore unable to run inside a browser runtime. Hence, it would be very convenient to have a web-based solution that has native support for the creation of metamodels, that include more advanced features, such as cross-references. In this work, a TypeScript-based approach for the creation of metamodels shall be investigated.

1.2 Aim of the Thesis and Expected Results

This thesis aims to expand the functionalities of Langium to provide model-oriented clients access to its Abstract Syntax Tree (AST). This will be done by implementing a model management API, which extends Langium via a service. With this enhancement, the model-oriented clients will be able to manage and manipulate the model state (by editing the state of the AST). As Langium is written in TypeScript, it can be browser-packaged, which means the complete model management can be handled inside the browser. Access to the local file system can be implemented to load model files by adding another service to Langium.

After implementing the model management API, a process will be introduced to define metamodels and generate Langium-based model management. This shall be done by creating a type definition based on TypeScript interfaces or classes, which add meta-information to model elements by the usage of custom annotations (e.g., `@crossReference`, which is used to signal a cross-reference) or custom container types (e.g., `CrossReference<T>`, which can also be used to signal a cross-reference). After that, a generator shall be implemented, which creates a generic JSON grammar from these type definitions.

According to the goals of this work, the following research questions will be answered in the course of this thesis:

(**RQ1**) : How can the Abstract Syntax Tree, which is created by Langium, be made available to model-oriented clients so that the editing of the model can be handled in the browser?

- The answer to this question provides a solution to how Langium can be used for the model management of browser-only model editors.

(**RQ2**) : How should a type definition be conceptualized to create an accurate metamodel that includes all needed language concepts and cross-references?

- By answering this question, it is ensured that the type definition concept is complete regarding the language concepts that are included within this thesis.

(**RQ3**) : How can the previously defined type definition be used to generate a language specification in a generic JSON grammar?

- With the answer to this question, a concept for a generator is created, which enables generic language specification.

1.3 Methodology

The methodological approach in this thesis follows Design Science Research [HMPR04] and consists of the following steps:

1. **Requirements Analysis**:
   In the first step, the requirements for the model service API and the type definition to define metamodels have to be gathered. The model service API will be analyzed using the implementation of state-of-the-art model editors (e.g., GLSP). Furthermore, for the type definition, the requirements will be specified by examining the capabilities of the Eclipse Modeling Framework (EMF) [Foub] Ecore metamodel.

2. **Conceptualization**:
   In this step, a concept for the model service API shall be created, which defines a generic way to create Langium-based model management (**RQ1**). Furthermore, a concept for the type definition has to be created, including all needed language constructs, to generate a metamodel. (**RQ2**)

3. **Build & Evaluate Artifacts**:
   In this step, the model service API and the generator (**RQ3**) that creates the language specification from the previously defined type definition, shall be implemented.

4. **Evaluation**:
   The evaluation of the artifacts that are created within the work of this thesis is threefold:

   - **Functional Testing**: To ensure the accuracy and dependability of the implemented artifacts, it is essential to create a comprehensive test suite comprising unit tests. This suite will help detect any potential defects or errors in the implementation of the artifacts and provide a reliable measure of their correctness.
   - **Descriptive Evaluation - Informed Argument**: During this evaluation, the newly implemented artifacts and the existing Ecore metamodel solution will be tested against predefined criteria. These criteria will assess the functional requirements and usability of both solutions and reveal any limitations of the newly implemented artifact. The evaluation will also highlight the advantages and disadvantages of the new and the existing solutions.

- **Descriptive Evaluation - Scenarios**: This assessment aims to reconstruct pre-existing tools using the recently implemented artifacts. The effectiveness of these artifacts will only be considered valid if the tools - such as the workflow-diagram example [Eclc] and the bigUML [BIGa] tool - can be rebuilt without any limitations.

1.4 Summary and structure of the work

This chapter gave an introduction to the thesis by providing the problem statement, as well as the aim and the methodological approach.

In chapter 2, some important terminology, followed by background information on the frameworks discussed in this thesis, is presented.

Chapter 3 presents some state-of-the-art graphical model editors and gives some insight into language engineering in general.

The requirements for the proposed artifacts are discussed in chapter 4. Following that, in chapter 5, the solution concepts for the requirements are presented. Chapter 6 discusses the actual implementation of the artifacts using the presented concepts.

Chapter 7 discusses all evaluation steps and results with respect to the research questions presented in this chapter.

Finally, chapter 8 concludes this thesis by presenting the findings of this thesis and discussing some possible future work.

CHAPTER 2

Background

In this chapter, the terminology around model engineering is introduced first. Next, the Eclipse EMF Ecore meta-model, which is one of the most popular frameworks for creating meta-models in the Java ecosystem, will be described. After that, the Language Server Protocol (LSP), which is a standard interface for connecting code editors and language servers, will be discussed. Following that, Graphical Language Server Platform (GLSP), a protocol extension that allows for creating rich graphical editors for Domain-Specific Modeling Languages, will be introduced. Subsequently, the focus will turn to Langium, a powerful language development framework based on the Xtext grammar development toolkit. Finally, this chapter will be wrapped up with a discussion of the Yeoman generator, which is a powerful tool that can be used to automate the creation of new projects based on predefined templates.

2.1 Terminology

In this section, some terminologies are presented, which should give an overview in regards to modeling.

2.1.1 Model

For the term model, there do exist a lot of different notions and definitions. In the following, a few of those definitions shall be mentioned.

"We can informally define a model as a simplified or partial representation of reality, defined in order to accomplish a task or to reach an agreement on a topic. Therefore, by definition, a model will never describe reality in its entirety."[BCW17]

Bézivin and Gerbé define a model as a simplification of a system built with an intended goal. The model should be capable of answering all questions in place of the actual system [BG01].

Thalheim states that a model is a well-formed, adequate, and dependable instrument that represents something and functions in scenarios of use [Tha22].

According to Selic, a model has to conform to the following five characteristics [Sel03]:

- **Abstraction**: A model is a simplified representation of a real system that includes only the relevant parts.
- **Accuracy**: A model must accurately represent the important aspects of the real system.
- **Cost-effectiveness**: The creation of the model should be more cost-effective than building the actual system.
- **Predictiveness**: By analyzing the model it should be possible to predict the properties of the modelled system.
- **Understandability**: It is not enough to exclude insignificant elements from the system. It is also required that the remaining elements inside a model be represented in an intuitive fashion (e.g. graphical notation).

Another definition of a model is provided by Seidewitz [Sei03]: *"A model is a set of statements about some system under study."*

Models can be used for various purposes and can have different applications. They can describe an already existing system, like a model of the subway system of a city that only includes stations and subway lines. Moreover, models can provide instructions on how a system should be developed. These purposes are known as descriptive and prescriptive, respectively [SHK12].

2.1.2 Modeling Language

Modeling languages are tools to create a textual or graphical model (see Section 2.1.1) of a system under study. These languages consist of abstract syntax, concrete syntax, and semantics [Kle08].

Abstract Syntax

In its original meaning, the term abstract syntax is the hidden, underlying, unifying structure of a number of sentences. This definition comes from natural-language research and was introduced by Chomsky [Cho65].

In the field of computer science, the following properties can also define the abstract syntax:

- **hidden**: The model's abstract syntax is hidden, and interaction is done through its concrete representation.

- **underlying**: The abstract syntax representation is the underlying structure of a model.
- **unifying**: The abstract syntax of a model is its unifying structure. This means that the unifying structure lies within the abstract syntax, regardless of the various concrete representations.

The abstract syntax is often referred to as the metamodel of a modeling language, and typically a model's abstract syntax is saved in the Abstract Syntax Tree (AST).

Concrete Syntax

The concrete syntax of the model enables humans to interact with the model. It can take on many different forms including a textual notation (see Figure 2.1) or a graphical notation (see Figure 3.2), which are the most common cases [BKP18]. The concrete syntax is typically provided via an editor.

```
{
    "nodes": [
        {
            "__type": "ActivityNode",
            "__id": "an__1",
            "name": "ActivityNode1"
        },
        {
            "__type": "ActivityNode",
            "__id": "an__2",
            "name": "ActivityNode2"
        }
    ],
    "edges": [],
    "metaInfos": [
        {
            "__type": "Position",
            "__id": "pos_an__1",
            "x": 100,
            "y": 100,
            "node": {
                "__type": "Reference",
                "__refType": "Node",
                "__value": "an__1"
            }
        }
    ]
}
```

Figure 2.1: Textual representation of the Workflow Diagram Model

Semantics

The semantics of a model are used to give meaning to the various symbols and variables contained in its representation. This meaning should be a part of a domain that is

well-understood and well-defined [HR04]. An example of semantics can be illustrated by considering the class diagram from UML [Gro17]. If a specialization edge connects two classes, the semantics of this specialization edge would inform us that one of the classes is the subclass of the other class.

Types of Modeling Languages

There are two main classes of modeling languages: General Purpose Modeling Languages (GPL) and Domain-Specific Modeling Languages (DSL) [BCW17].

- **General Purpose (Modeling) Languages (GPL)** : GPLs are not implemented for a certain domain but are rather usable in a lot of different contexts. An example of a GPL is the Unified Modeling Language (UML).
- **Domain Specific (Modeling) Languages (DSL)**: In contrast to GPLs, DSLs are modeling languages that are designed for a certain domain. An example of a widely used DSL is the Hypertext Markup Language (HTML). For DSLs, another distinction between *Internal* and *External* DSLs exist [VBD+13]. An Internal DSL is a DSL built on top of a GPL. This means that Internal DSLs reuse the syntax and tooling from the GPL to create a custom DSL that is specific to a particular domain. On the other hand, External DSLs are designed from scratch, which means that the syntax and tooling has to be newly defined. This gives External DSLs a lot of flexibility since they can be tailored specifically to meet the needs of a particular domain [Fow10].

2.1.3 Metamodeling

Metamodeling is a very important part of model-driven engineering. To understand metamodeling, first, a definition for a metamodel has to be defined. In simple terms, a metamodel can be described as a model of a model.

In [Kle08] a metamodel is defined as *"a model used to specify a language"*.

According to Seidewitz [Sei03], a metamodel defines the structure and constraints of a group of systems under study (SUS), where each SUS in the group is a model that conforms to a particular modeling language.

In [BCW17], a metamodel is an abstraction of an already existing model (that describes a phenomenon in the real world). As with this definition, the metamodel provides a way of describing the whole class of models that can be represented within the model; it can be concluded that the metamodel defines a modeling language.

Metamodeling involves specifying a model's elements, constraints, and interactions in a particular domain. There are different methods of metamodeling [BKP20], such as using a grammar to define a metamodel in Xtext [Foud] and Langium [Lan] (see Section 2.4) or using the metamodeling language provided by EMF Ecore [Ecla] (see Section 2.2) to define a specific language's metamodel.

In Figure 2.2, the 4-layer metamodeling stack [Gro16] of the Object Management Group is illustrated. From top- to bottom-layer, an *instance-of* relationship exists between the layers, while the reverse relationship can be characterized as a *conforms-to* relationship. In the following, each layer of the stack is described:

- **M0-layer**: This is the layer containing the actual user data of the model.
- **M1-layer**: This layer maintains a model that describes the structure and behavior of the M0-layer-instances.
- **M2-layer**: This layer holds a model of the M1-layer, making it a metamodel as it is a model of a model.
- **M3-layer**: This layer holds a model of the M2-layer, which is a metamodel, making the model at the M3-layer a meta-metamodel.

As shown in the figure, no additional layer is above the M3-layer. The reason for this is because the meta-metamodel can function as a model of itself, meaning that it can describe itself [BCW17].

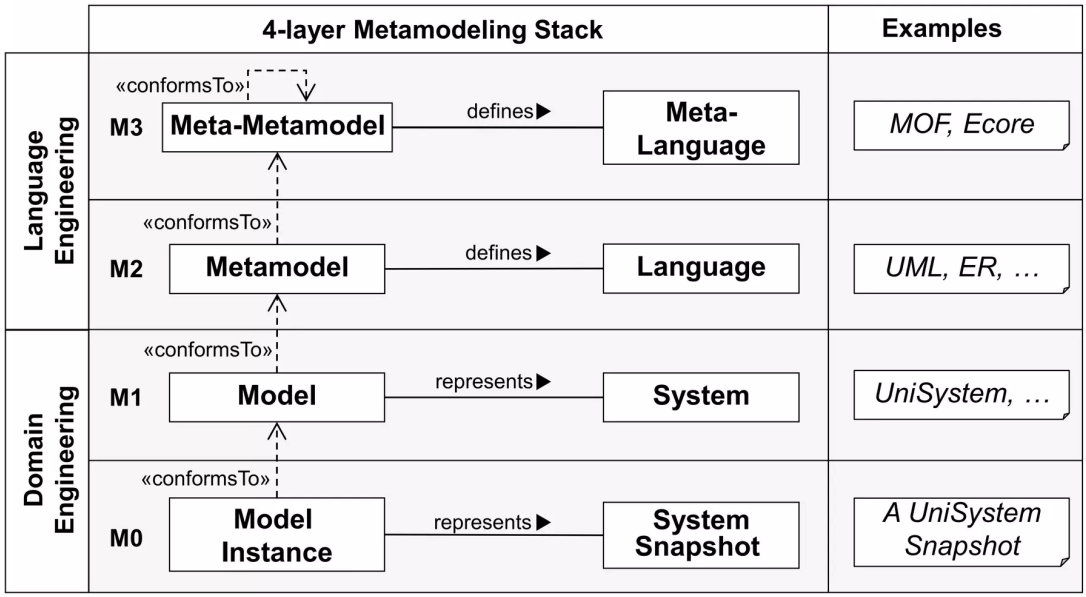


Figure 2.2: 4-layer metamodeling stack as presented in [BCW17]

2.2 Eclipse Modeling Framework (EMF)

The Eclipse Modeling Framework, commonly known as EMF, is a modeling framework that serves as a fundamental component of the Eclipse ecosystem. With the help of

EMF, it is possible to define a model for a system that is under study. EMF enables the generation of multiple representations of this model in different formats, such as Java interfaces, UML, or XMI. The ability to unify different representations of the model, which simplifies the model creation process, is one of the primary functions of EMF. In EMF, it is enough to create a model using one representation, as the other representations are automatically generated. The diagram in Figure 2.3 illustrates how an EMF model unifies these different representations.

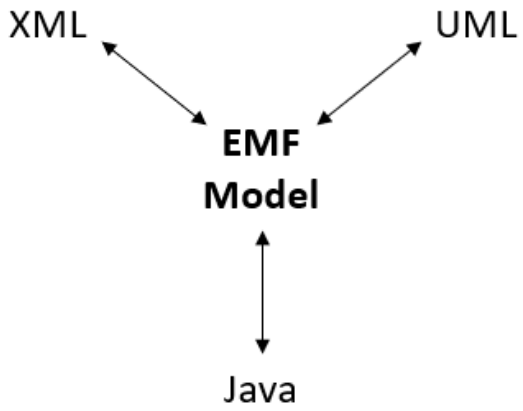


Figure 2.3: EMF model unifying multiple different representations based on [Ste09]

For the definition of a model, EMF provides a metamodeling language in Ecore [Ecla]. Ecore allows developers to specify the structure of their models, including their classes, attributes, and relationships. In [Ste09] the Ecore kernel, which is a simplified representation of the Ecore metamodel, is presented. Its representation is illustrated in Figure 2.4. Some of its most important components are the following:

- **EClass**: These are the main building blocks of a model that represent a class and can contain *EAttributes* and *EReferences*. As can be seen in the figure, an *EClass* can have an arbitrary number of *eSuperTypes*, which are used to represent inheritance.
- **EAttribute**: These elements define the properties of a class in an Ecore model. They provide information about the type of a property and its multiplicity and are identified by their name.
- **EDataType**: These elements specify the data type that an *EAttribute* can hold. Note that the type of elements an *EDataType* can represent cannot be modeled as classes.
- **EReference**: These elements establish connections between *EClasses* and specify the number of elements involved in a relationship. Furthermore, they determine

whether the referenced element is a part of the source element or can exist independently, indicating a cross-reference.

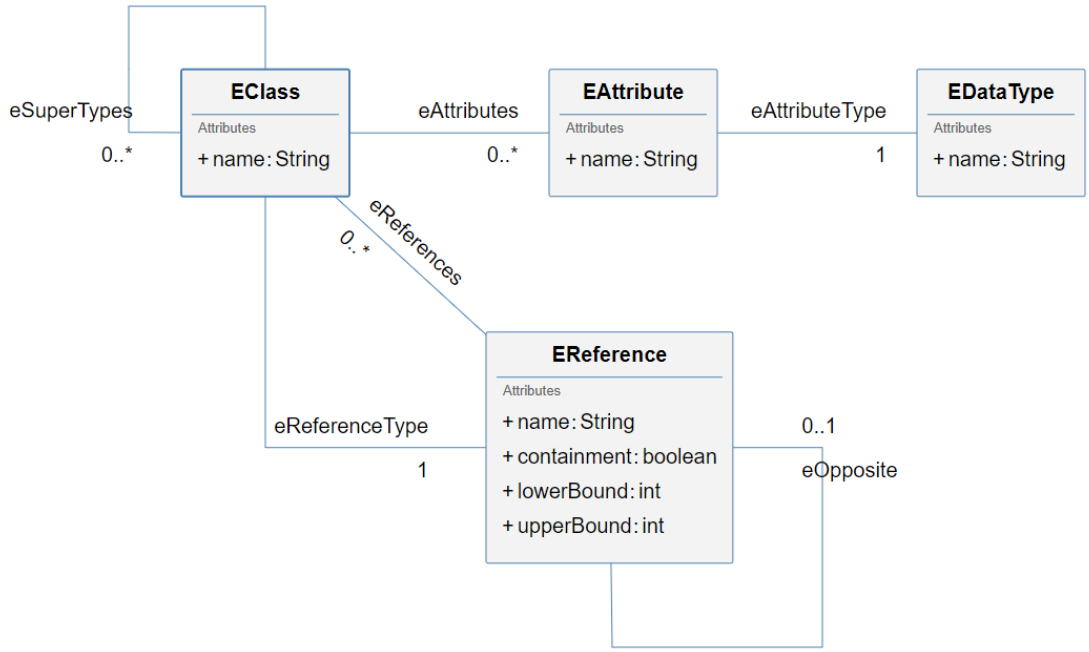


Figure 2.4: Ecore kernel based on [Ste09]

Next to its tools to implement models, EMF provides first-class support for code generation, including different types: Model code generation, adapter code generation, and editor code generation [Foub].

Another important part of EMF is that it can be easily integrated within other Eclipse Modeling projects such as Xtext [Foud]. Xtext is a language development framework that can be used for the creation of Domain-Specific Modeling Languages. Using EMF in conjunction with Xtext allows developers to define the structure of their Domain-Specific Modeling Languages using EMF's modeling capabilities.

2.3 GLSP - Graphical Language Server Platform

The Language Server Protocol (see Section 2.3.1) has been gaining popularity due to the shift from feature-rich Integrated Development Environments (IDE) to lightweight web clients. The problem with IDEs is the tight coupling of language features to the IDE, which means that for every DSL, there needs to be a plugin for the respective IDE. For example, to get syntax highlighting and code completion for Java within a code editor like Emacs or Vim, a plugin needs to be implemented specifically for the respective editor.

2.3.1 LSP - Language Server Protocol

The Language Server Protocol [Micb] [BL23] has been introduced to decouple the language features from the IDEs. With LSP, the web client is just a text editor, while the language smarts are implemented within a language server. This means, that syntax highlighting, autocomplete support, and cross-references are all handled within the language server, while the client is responsible for the representation of the current state and sending requests to the language server. Figure 2.5 illustrates how the implementation effort can be reduced using the Language Server Protocol in case one wants to implement language features for a certain language.

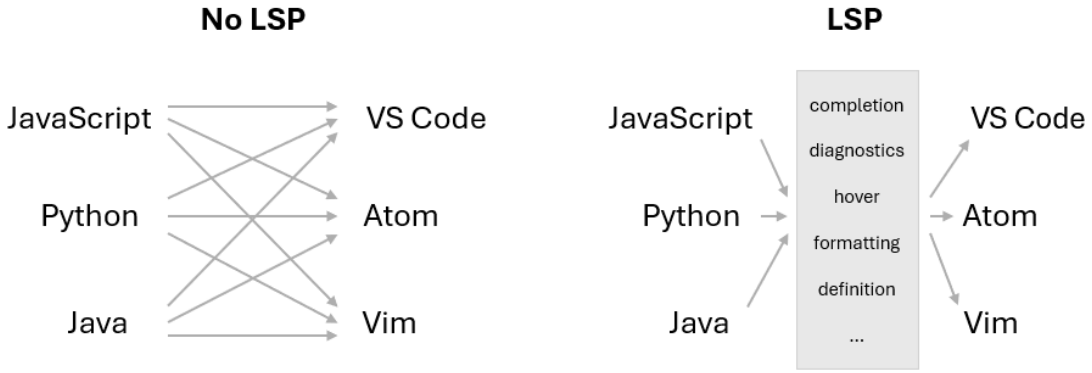


Figure 2.5: Visualization of the coupling of language servers and IDEs with and without using LSP

The architecture of the Language Server Protocol consists of a client process and a server process, which communicate using a well-defined protocol. The communication is done via JSON-RPC requests.

Figure 2.6 represents an example communication procedure between a server and client. The procedure begins with the user opening a text document on the client, which causes the client to send a request to notify the language server, that a document has been opened. Following that, the user edits the document and the client sends a request to notify the server of the changes. The server then processes this request, creates diagnostics for the current text document (i.e., checks for syntax errors), and sends the result back to the client. Subsequently, the user looks up the definition of an element. Therefore, the client sends a request to the server to request the definition for the element on the selected position. The server then processes the request and returns the location of the definition of the element. Finally, the user closes the text document, which causes the client to notify the server, that the text document has been closed.

2.3.2 GLSP

Although the Language Server Protocol eases the implementation of language support for multiple different editors, it focuses only on textual languages. Because of this, the

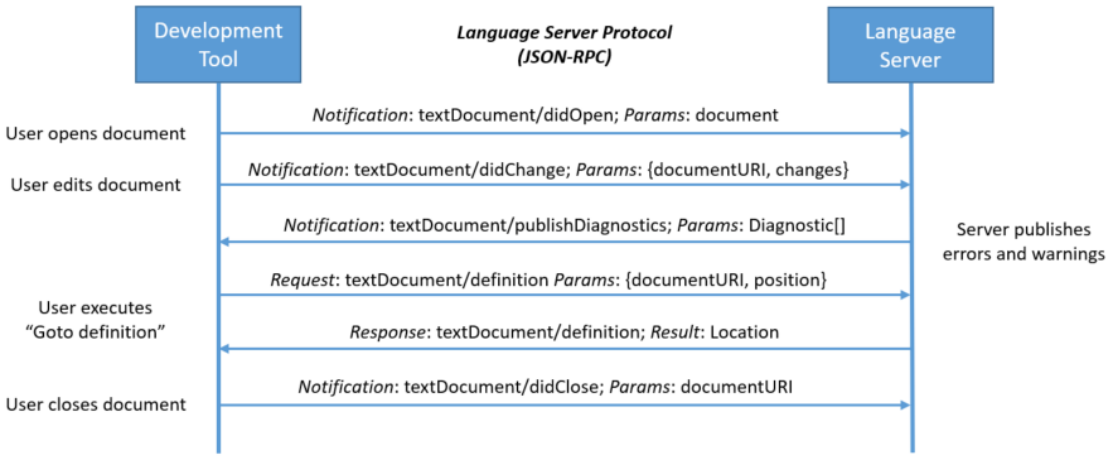


Figure 2.6: Sample communication between LSP client and server [Micc]

Graphical Language Server Platform (GLSP) [1] [Eclb], which is built on top of the LSP to facilitate the development of diagram editors using this decoupled architecture, extending its capabilities for graphical modeling [BLO23] [MB23], has been introduced.

Figure 2.7 illustrates the structure of GLSP. As can be seen, similar to LSP, in GLSP, there exists a model client, and a GLSP server, which is responsible for handling the computation-heavy work like loading, saving, and transforming the model, while also providing the language smarts. The communication between the server and client is done using the graphical language server protocol, an extension of the LSP.

Either the GLSP server holds the source model, as depicted in the figure, or the source model can be integrated within a separate server, which is then called the model server. In the course of this thesis, such a model server will be created using Langium (see Section 2.4).

GLSP provides ready-to-use components for each part of its architecture. For the client, a Visual Studio Code [Micd] extension and a Theia [Foua] extension exist. However, a diagram editor can also be built inside any web-based application to be used with the graphical language server protocol.

While the GLSP server can be written in any language, ready-to-use components in TypeScript and Java exist.

Furthermore, the source model can be saved in any format, like JSON or text. Also, once again, it is possible to use components that have already been integrated into GLSP, like EMF [Foub], EMF.cloud [Fouc], or GModel-JSON (which holds the graphical model directly).

The process of using the graphical language server platform can be described as the following: After the startup of both the GLSP client and server, the server starts by

[1] `https://eclipse.dev/glsp/`

Figure 2.7: Graphical Language Server Platform structure as presented in [Eclb]

loading the requested model and transforming it into the graphical model. The graphical model is a serializable representation of the source model and is the central communication artifact to the client. Subsequently, the server sends the graphical model to the client, which is then responsible for rendering the model. Additionally, the client requests information from the server regarding the possible operations on each distinct element of the model. Using the information provided by the server, the client can then provide the editing tools for the different kinds of model elements.

In case of an edit on the client side, the client sends a request to the server with the respective operation. The server then applies the operation on the source model, regenerates the graphical model from the updated source model, and sends it back to the client, which renders the graphical model.

On a more technical side, GLSP focuses on customizability and extensibility, as diagram editors are particular to their respective diagram languages. Both client-side and server-side are implemented using an inversion of control pattern based on Dependency Injection (DI). For the node-based implementation, *inversify.js*[2] is used to put every service and component inside a global DI container. Additionally, these services and components can easily be extended or replaced by custom implementations.

Having control over the visual representation of a model is crucial when working with diagram editors. GLSP provides complete command over the technologies used for the user interface, including Eclipse Sprotty [3], CSS, and SVG. With GLSP, developers have the freedom to customize their editing tools and user interface controls in HTML, without any abstraction layers that could limit their power. As a result, they can maintain complete control over the rendering of the user interface, cf. [CLB22].

[2] https://github.com/inversify/InversifyJS
[3] https://sprotty.org/

2.3.3 GLSP - Components

As has already been discussed, GLSP offers a lot of extensibility and customizability. The following will give an introduction to some of the most essential components and services of GLSP.

Source Model

As we have previously discussed, the source model contains the actual data of the diagram and can be saved in any format. This is possible because GLSP allows the developer to define how a source model should be loaded.
For the implementation of the source model, two main components have to be implemented which are:

- **Source Model Storage**: This interface consists of two functions, `loadSourceModel` and `saveSourceModel`. As their name states, the implementations of these functions are responsible for the loading and saving of the source model.
- **Model State**: This is an important class that stores the current state of the original source model within a client session. Other services and handlers can access the model state to get the necessary information about the model to perform their diagram editing tasks.

Graphical Model

The graphical model is a serializable representation of the source model and is the main communication artifact from the server to the client. The graphical model is created using the `GModelFactory` after the source model storage has loaded the source model. The `GModelFactory` is very specific to each diagram language. Therefore, it has to be customized for most GLSP servers.

The graphical model consists of exactly one root element called `GModelRoot`. All other elements inside the graphical model can be one of the following:

- `GShapeElement`: This is the base element of graphical elements and consists of a shape with visual bounds (position and size). The following concrete subtypes exist:
  - `GNode`: This elements represents a graphical node.
  - `GPort`: This element is used as a connection point for `GEdges`.
  - `GLabel`: This element represents a text label.
  - `GCompartment`: This element represents a container for grouping multiple graphical elements.
- `GEdge` This element represents an edge between two `GShapeElements`.

Model Operations

In order to make changes to the model, model operations need to be provided. These operations are managed by `OperationHandlers`. An `OperationHandler` specifies which type of operation needs to be managed and how it should be processed. When an operation is executed, the `OperationHandler` processes it and produces a `Command` that captures the changes made to the source model. This `Command` is then placed on the command stack and sent to the server, which updates the source model according to the `Command`.

DiagramModule

The `DiagramModule` plays a crucial role in configuring a GLSP server. It serves as a central artifact where all custom components and services can be bound to the Dependency Injection container, seamlessly integrating various functionalities. The importance of the `DiagramModule` lies in its ability to simplify the development process, by providing a single point of reference for all the necessary configurations.

2.4 Langium

Langium[4] is a framework designed to develop Domain-Specific Modeling Languages (DSL). It provides many tools for language engineers to create a custom language tailored to the specific needs of a certain system. Additionally, Langium is built with TypeScript, which means that it can be easily integrated within web-based applications.

In the following, the key features of Langium will be described.

2.4.1 Language Server Protocol (LSP) Support

Langium includes comprehensive Language Server Protocol (LSP) support through its core framework and specific LSP implementations. With its pre-built implementations, Langium simplifies language tasks such as parsing, Abstract Syntax Tree (AST) generation, validation, scoping, cross-referencing, and more. Langium also includes a ready-to-use language server implementation that can be used to build and maintain a language server.

2.4.2 Grammar Definition

Langium provides a concise syntax that can be used to define a DSL. This syntax includes constructs to define tokens, rules, and the overall structure of the language. The structure of this grammar language will be described in detail in the following subsections.

Language Declaration The language declaration is done by a grammar file with an entry rule. This rule declares the name of the language. Listing 2.1 defines a grammar language named `ExampleGrammar`.

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

```
grammar ExampleGrammar
```

Listing 2.1: Declaration of the grammar language name

Terminal Rules The initial step in language parsing is lexing, which converts a character stream into tokens. A token is a sequence of one or more characters that matches a terminal rule, creating an atomic symbol. Terminal rules are typically written using Regular Expressions, but it is also possible to write them using Extended Backus-Naur Form (EBNF) expressions. An example of a terminal rule that parses a stream of alphabetic characters into a token with the name `STRING` can be seen in Listing 2.2.

```
terminal STRING: /[A-Za-z]+/;
```

Listing 2.2: Terminal rule for a string

Terminal rules return string elements per default if not otherwise specified. If it should return an element of another type (like a number), it is possible to adapt the rule, as can be seen in Listing 2.3.

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

Listing 2.3: Terminal rule for a number

A special case of a terminal rule is the hidden terminal rule, which is used to specify which characters inside a text file need to be ignored by the lexer. Typically, the lexer should ignore white spaces and comments. Additionally, it should be noted that hidden terminal rules are applied globally. An exemplary terminal rule, which is used to ignore white spaces can be seen in Listing 2.4.

```
hidden terminal WS: /\s+/;
```

Listing 2.4: Hidden terminal rule to ignore white spaces

Parser Rules *"While terminal rules indicate to the lexer what sequence of characters are valid tokens, parser rules indicate to the parser what sequence of tokens are valid." [Lan]* Parsers lay the groundwork for creating the Abstract Syntax Tree as they are used to specify the structure of objects that need to be created.
Parser rules are defined using EBNF expressions; a rule consists of the name of the rule followed by the definition of the elements that the rule consists of.

For example, Listing 2.5 consists of two rules: The first rule is creating an element of type `Node` with the property name, which is a `STRING`. The other defines an element of type `WeightedNode`, which consists of the properties name, which is a `STRING`, and weight, which is a `number`.

```
Node:
    'node' name=STRING;
WeightedNode:
```

```
    'weightedNode' name=STRING weight=NUMBER
```

Listing 2.5: Declaration of two kinds of parser rules

Additionally, the entry rule is a special parser rule that serves as the entry point for the parser. It begins with the keyword `entry`. This can be seen in the exemplary entry rule in Listing 2.6, which defines the `ExampleGrammar` that consists of an arbitrary length of either `Node` or `WeightedNode` elements.

```
entry ExampleGrammar:
    (nodes+=Node | weightedNodes+=WeightedNode)*;
```

Listing 2.6: Declaration of the grammar language's entry rule

In Listings 2.1 to 2.6, a lot of options for the definition of elements have already been used. The following list explains most of the options that can be used to define the elements of a parser rule:

- **Cardinality**: With cardinality, the number of elements in a given set is defined; there exist four options:
  - exactly one: no operator
  - zero or one: question mark (?) operator
  - zero or more: star (*) operator
  - one or more: plus (+) operator
- **Alternatives**: Using the pipe operator (|), it is possible to define multiple different valid alternatives of how an object is defined.
- **Assignments**: There do exist three different kinds of assignments:
  - Single value assignment: With the assignment operator (=), it is possible to assign exactly one value to a property.
  - Multi-value assignment: With the plus-equals operator (+=), it is possible to assign multiple values to one property in an array.
  - Optional value assignment: With the question-mark assignment operator (?=), it is possible to assign a boolean value to a property in case a certain element exists during parsing.
- **Cross-References**: The definition of cross-references is a very important aspect, as they can be used to create a reference to other elements inside the model. To assign a property to a cross-reference, the target type has to be put inside square brackets. Listing 2.7 contains an example that uses a cross-reference. In this example, two objects are defined: a `Node`, which includes a name-property that is parsed by the terminal rule `ID`, and an `Edge`, which includes two references to `Node` elements within its source and target property.

```
Node: 'node' name=ID;
Edge: 'edge' source=[Node:ID] target=[Node:ID];
```

Listing 2.7: Declaration of cross-reference

- **Unassigned Rule Calls**: It is not required for a rule call to return a new object. It is also possible for a rule call to assign sub-rule calls to return the object, as can be seen in Listing 2.8. In this example, the rule `TaskNode` is an abstract rule or unassigned rule call that returns no concrete object; it rather assigns the object creation to either `AutomatedTask` or `ManualTask`, which will return an object of the associated type with a name property.

```
TaskNode: AutomatedTask | ManualTask;
AutomatedTask: 'task' 'automated' name=STRING;
ManualTask: 'task' 'manual' name=STRING;
```

Listing 2.8: Declaration of unassigned rule calls

- **Unordered Groups**: If the body of a rule consists of multiple properties, the parser expects the elements in a certain order. An example of this can be seen in Listing 2.9, where the parser would detect a `WeightedNode` only if the text appears in exactly the order as defined (e.g., "node NodeName 69").

```
WeightedNode: 'node' name=STRING weight=NUMBER;
```

Listing 2.9: Declaration of parser rule with ordered group of attributes

With the and-operator (&), it is possible to ignore the order of elements as can be seen in Listing 2.10. With this definition, the parser would recognize both "node NodeName 69" and "node 69 NodeName" as an object of type `WeightedNode`.

```
WeightedNode: 'node' name=STRING & weight=NUMBER;
```

Listing 2.10: Declaration of parser rule with unordered group of attributes

A limitation for these unordered groups is the assignment of properties with a cardinality of either * (zero or more) or + (one or more). For these properties, the elements of these properties must appear continuously, without being interrupted by another assignment. An example of this can be seen in Listing 2.11, where the parser would not be able to parse the text "node NodeName1 69 NodeName2" as an object of type `WeightedNode`, as the `names` elements do not appear in continuous order.

```
WeightedNode: 'node' (names+=STRING)+ & weight=
   NUMBER;
```

Listing 2.11: Declaration of parser rule with unordered group of attributes, with one attribute being an array type

2.4.3 Code Generation and Tranformations

Langium provides a flexible and extensible architecture that makes creating custom code generators that meet specific needs easy. The Langium CLI can be extended to include a generate command, which can be used to start the generation process. This generate command can be customized to include various options that control the output of the generated code.

To enable code generation, a simple JavaScript function can traverse the Abstract Syntax Tree (AST) and modify the elements as required for the desired output. The JavaScript function can be designed to perform various operations on the AST, such as adding, removing, or modifying nodes.

2.4.4 Abstract Syntax Tree (AST) Generation and Document Lifecycle

The Abstract Syntax Tree generation is one of the main features of the Langium framework. To understand how a source text is transformed into its AST representation, the document lifecycle of Langium needs to be explained first.

On startup, Langium initializes the opened workspace and loads all files inside the workspace, which should be processed. This is done by the `WorkspaceManager` service. To determine which files need to be loaded, the `WorkspaceManager` checks the file extension of all files inside the workspace and loads those whose file extension equals the one in the Langium configuration file.

All files in the workspace are then represented as instances of `TextDocument`, which hold the contents of the respective files as simple strings. After determining which files should be loaded for each selected document, an instance of `LangiumDocument` is created using the `LangiumDocumentFactory` service. Unlike a `TextDocument` that represents the content of a file as a string, a `LangiumDocument` uses an AST to represent the content. Therefore, the `LangiumDocumentFactory` has to parse the source-`TextDocuments` and transform the parsed content into an AST. Once the `LangiumDocument` instances are created, they are stored inside the `LangiumDocuments` service, and the documents possess the state `Parsed`.

Subsequently, the `DocumentBuilder` service takes over and starts with the indexing of symbols, which are AST nodes, that can be used for cross-references. The indexing starts with the root element of the parsed document and creates an `AstNodeDescription` for every descendent. This description allows other documents inside the workspace to access the symbol. All exported symbols are then saved in the `IndexManager`. Here, the `ScopeComputation` is responsible to select which elements inside a document should be exported symbols and therefore be available for cross-references. In the default implementation, all elements that contain a `name` property are selected for indexing. After this step is done, the current `LangiumDocument` is in the state `IndexedContent`.

Next, the local scope computation is done. This computation collects all the symbols present in the AST. Once all the symbols are collected, their metadata is stored as

`AstNodeDescription` in a multimap, which assigns all symbols in the document to their respective container. After this process, the document is in the `ComputedScopes` state.

Now, the `DocumentBuilder` starts the linking phase, in which cross-references are resolved using the `Linker` service. Here, the `ScopeProvider` service is used to obtain a scope that describes all reachable elements from the AST node holding a cross-reference. Using this scope, the symbol matching the cross-reference identifier is searched, and if a match is found, the respective AstNode is loaded. Following the linking phase, the state of the document is `Linked`.

After this phase, the `IndexManager` service is used again to index all cross-references. A `ReferenceDescription` between source and target documents for the cross-references is created in this phase. After this, the document is in the state `IndexedReferences`.

The final phase before the document is ready to process requests from the editor is the validation. Here, the `DocumentBuilder` validates the document and creates diagnostics containing errors during lexing, parsing, and linking, as well as possible errors identified using custom validation functions. Now, the document is in the state `Validated` and ready for requests.

Once the client modifies the document, the `LangiumDocument` switches its state to `Changed`. The modified document is invalidated and removed from the previously mentioned `LangiumDocuments` service. All documents connected to the modified document that could be affected by the change get their references unlinked and run through the linking phase again. A new `LangiumDocument` instance is created for the modified document, and it runs again through all the steps described above.

In Figure 2.8, an overview of the different states of a `LangiumDocument` is illustrated.

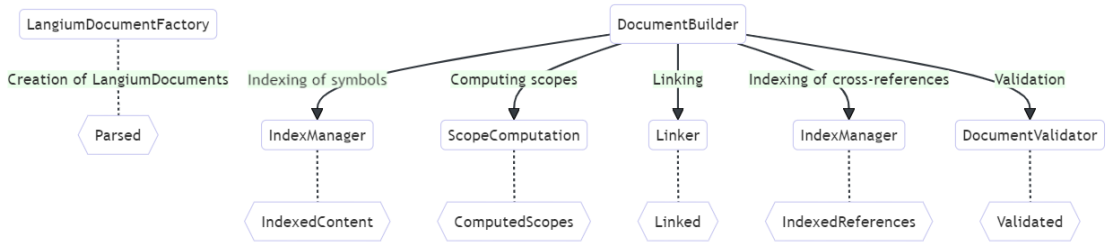


Figure 2.8: Document Lifecycle of `LangiumDocuments` as presented in [Lan]

2.5 Generator

Generators play an important part in modern software engineering. They can be used in different contexts, like code generation, which has already been mentioned a few times, or project generation.

A very popular tool regarding project generation for web-based tools is the Yeoman[5] package. This package allows developers to create an initial structure of an application through the use of templates, which are called "generators" in the context of Yeoman.

The usage of the yeoman package is very simple. To create an initial project for some application, all that needs to be done is to install the package and the generator for the respective application. If, for example, a Langium project is set up, the following commands need to be executed on the command line:

1. `npm install -g yeoman generator-langium`
2. `yo langium`

During the execution of the second command, the generator will prompt the user to enter some parameters to determine predefined elements inside the application that shall be generated. In the case of Langium, this will include the name of the language, its language ID, and its file extension.

After this command is executed successfully, the initial project setup for a Langium project is created.

Yeoman generators do not only offer an option to set up an initial project structure, but it is also possible to include these generators throughout the development process. They can offer a set of subcommands, which can be called through the CLI. In the Langium generator example, a command can be called to regenerate the AST definition after editing the grammar definition.

2.6 Summary

In this chapter, some terminology that is relevant to this thesis has been presented. Additionally, some important technologies have been discussed, such as the Graphical Language Server Platform, Langium, and Yeoman, whose capabilities will be utilized in the development of the artifacts that this thesis aims to create. In the following, some state of the art model editors and how their model management is handled will be presented.

[5] `https://yeoman.io/`

CHAPTER 3

State of the art

This chapter will present selected state of the art model editors. The focus will be on whether their abilities overlap with the aim of this thesis, which is to develop model management that is entirely browser-based. Finally, also insight into the current capabilities of language engineering is given.

3.1 Model Editors

Some approaches already exist for developing web-based modeling tools. In the following, the functionality of some of these editors will be discussed, and the differences in the solution that will be created within this thesis will be mentioned.

3.1.1 BIGER modeling tool [GB21]

The BIGER modeling tool is a hybrid textual-graphical editor solution that can be used to design entity relationship diagrams. Figure 3.1 illustrates the different views of the BIGER tool.

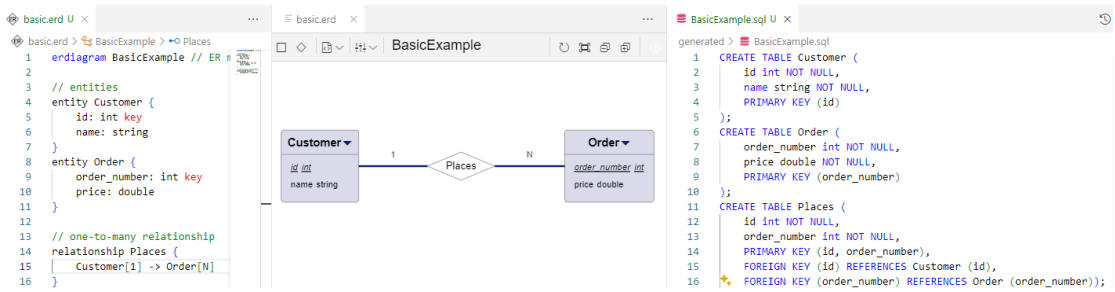


Figure 3.1: Views of the BIGER modeling tool

Similar to the artifacts in this thesis, bigER is distributed as a VSCode extension. However, it utilizes the Language Server Protocol (see Section 2.3.1) and its language server is written using Xtext, while the client is implemented with Sprotty. A drawback of the usage of Xtext, is that it relies on Java in the backend; therefore, it is not possible to run this tool entirely inside a browser runtime, which would be the goal for this thesis.

3.1.2 GLSP based editors

The Graphical Language Server Platform (see Section 2.3) [Eclb] [MB23] is an extension of LSP, which is also used within the artifacts of this thesis. Additionally, it uses either Java or TypeScript in its implementation.

In Figure 3.2, an example model of the workflow diagram example [Eclc] can be seen, which is the default example of GLSP.

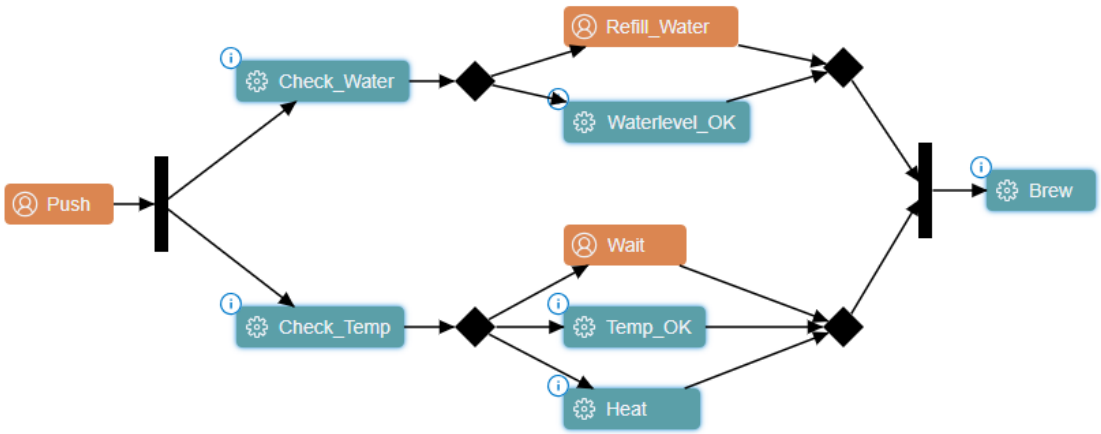
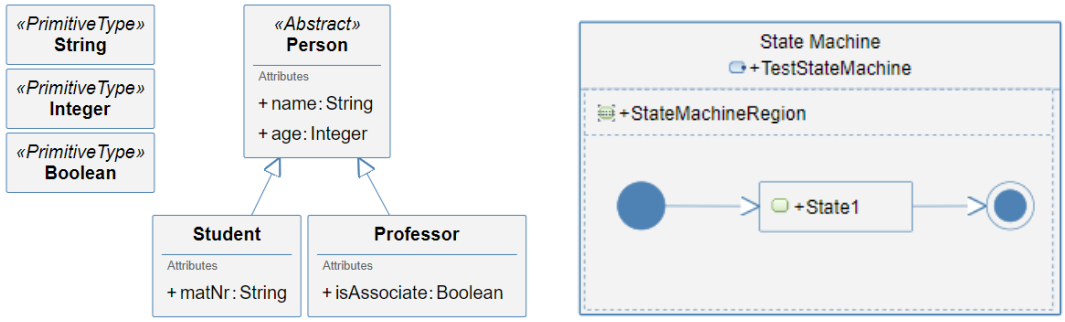


Figure 3.2: Workflow Diagram implemented in GLSP

The bigUML [BIGa] [BIGb] tool is a modeling tool specifically designed for UML diagrams. It supports multiple different types of diagrams, including the class diagram, use case diagram, state machine diagram, and more. In contrast to the bigER tool, the bigUML tool utilizes GLSP. However, bigUMLs implementation of the language server is written in Java; therefore, to be able to start the tool, it is necessary to have a Java runtime environment installed, which again means that this tool can not be used in a browser-only environment. In Figures 3.3a and 3.3b, two example diagrams created with the bigUML tool can be seen. Additionally, bigUML is distributed as a VSCode extension [BIGa]. As an example, Figure 3.4 illustrates the different views of the bigUML tool within the Visual Studio Code editor.



(a) Class Diagram (b) State Machine Diagram

Figure 3.3: Class Diagram and State Machine Diagram created using BIGUML [BIGa]

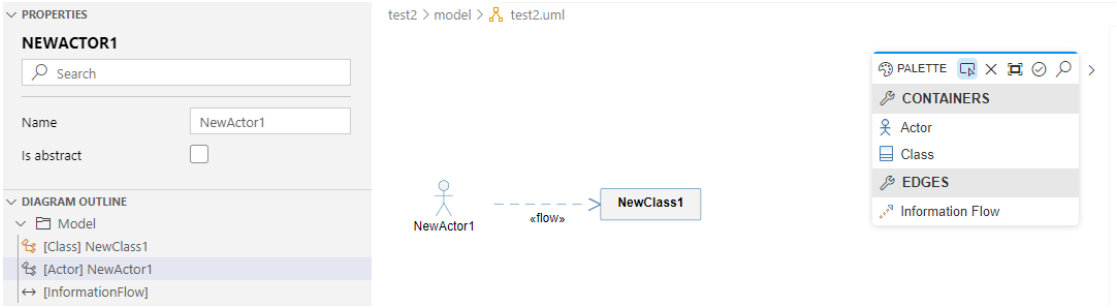


Figure 3.4: Views of the BIGUML modeling tool

Furthermore, in [DCLB22], the capabilities of the modern web-based technology stack used within GLSP are shown, as semantic zoom and off-screen elements are implemented, which are very advanced model visualization and interaction functionalities. Also in this work, the model management, which includes editing and validation, is done within a GLSP server that is written in Java.

3.1.3 Sirius Web

Sirius[1] is a well-known modeling tool in the Eclipse community, which has been extended by Sirius Web[2]. With this extension, users can access graphical editors in the browser and work in a cloud environment. However, it is important to note that Sirius Web can only perform graphical editing in a web browser. Developing the underlying models and defining the appearances for respective graphical editors are still a part of the Sirius Eclipse application, which is also still Java-based.

[1] https://eclipse.dev/sirius/
[2] https://eclipse.dev/sirius/sirius-web.html

3.2 Language Engineering

In the field of language engineering, there are various approaches that have been adopted over time. One approach was to define Domain-Specific Languages (DSLs) by creating a grammar, as stated in [JBF11]. However, there has been a shift towards defining DSLs using metamodels, as demonstrated by the Eclipse Modeling Framework (EMF) [Foub]. This approach uses a modeling language called Ecore to define the metamodels for the DSLs.

Additionally, EMF can be easily integrated with a technology called Xtext [Foud], which enables the creation of domain-specific languages using a Java technology stack. However, this thesis uses a different technology called Langium for model management that is entirely based on TypeScript.

Eclipse provides a solution for managing models within the browser through EMF.cloud [Fouc]. However, this solution only consists of libraries and frameworks that wrap EMF's Java-based solution to enable browser-based model management. As a result, a Java server is still necessary for implementing the model management; therefore, the tool can still not run in a browser-based environment.

In [GMGC22], DescribeML has been introduced, which uses Langium to create a model-driven tool to describe machine learning datasets. In the paper, a DSL is created, and Langium is extended by a few services to extend its capabilities. This paper provides a good overview of what is possible in language engineering when using Langium.

This thesis will discuss a new approach to the metamodel definition using annotated TypeScript classes. This approach allows for the creation of structured definitions that include the specification of properties, methods, and relationships between entities. Utilizing TypeScript classes can ensure that the metamodels are consistent, reliable, and easily maintainable.

3.3 Summary

In this chapter's first section, various graphical editors with different capabilities have been presented. It has been shown that none of them is capable of handling model management in the browser. However, browser-based model management is desired, as it would enable the shift of the entire modeling tool into the browser. This would then allow using the modeling tool without any installation steps. As mentioned in the previous sections, the usage of Java is one of the main obstacles on the path to browser-based modeling tools. Therefore, this thesis will discuss a new model management approach using Langium, which is written in TypeScript. Furthermore, in the second section, some language engineering approaches, including the one used within this thesis, are mentioned to provide an initial overview. The next chapter will focus on the requirements of the model management solution that will be implemented during this thesis.

CHAPTER 4

Requirements

In this chapter, the requirements of the artifacts that will be implemented in the course of this thesis are presented. First, the general idea and approach of the implementation will be explained. Subsequently, the requirements for the different artifacts are presented.

4.1 Main idea and general approach

The main aim of this thesis is to offer an alternative option for graphical modeling that moves away from the Java-based EMF technology and towards a web-friendly TypeScript solution. This will be achieved by using Langium, a next-generation textual language framework that is written in TypeScript and provides out-of-the-box support for the Language Server Protocol. Langium is highly extensible and as such, it will be extended to enable graphical editors to access the Abstract Syntax Tree of a model. This will be done by creating a model server, that provides an interface for graphical clients to access the AST.

As the definition of a metamodel is also typically done using Eclipse EMFs Ecore metamodel, another important aspect in this thesis is the definition of metamodels using a TypeScript-based approach. For this, Langium provides a grammar language that allows to define a grammar for generating a language metamodel. However, this grammar is not TypeScript-native. Therefore, this thesis aims to create a grammar definition language based on TypeScript classes in combination with annotations, interfaces, and types.

Furthermore, to provide out-of-the-box support for the newly defined grammar language, which already includes the model service, and to enable seamless integration with graphical model clients, a project generator shall be created that sets up Langium and the required services. Additionally, Command Line Interface (CLI) commands will be provided to update the metamodel in line with the current definition effortlessly.

4.1.1 General approach for the prototype implementation

The approach to implementing the required artifacts is to start by implementing the model server. For this, an already existing solution from CrossBreeze [NL] will be used as a starting point and expanded with the required functionalities.

For the creation of the (TypeScript-native) grammar language, first, Ecore's capabilities will be analyzed to find the requirements of the grammar definition. Following that, the grammar definition that uses classes in combination with annotations, interfaces, and types will be created.

To easily set up a project using the newly created grammar language with Langium, a Yeoman generator, as described in Section 2.5, will be created that parses the grammar definition and creates the Langium services according to the specification.

Finally, two approaches are used to evaluate the artifacts. First, a unit testing framework will be set up to evaluate the generator against predefined criteria. Second, to evaluate all artifacts, the workflow diagram example [Eclc] and the bigUML tool [BIGa] will be recreated using the new TypeScript-only technology stack. Both these modeling tools are built using GLSP (see Section 2.3). This means that both a GLSP client and a server will be required for the evaluation. For the workflow diagram example, a TypeScript solution already exists for both the client and the server, which means that it is only necessary to create the connection from the GLSP server to the Langium model server. However, for the bigUML tool, there is only a TypeScript solution for the client, which means that for the evaluation, a new TypeScript GLSP server has to be implemented that has (mostly) the same functionalities as the Java-based server.

4.2 Model Server API

The model server is the central artifact in the TypeScript-based model management solution. It has to provide multiple different functionalities, including loading, saving and editing models.

As already discussed, CrossBreeze [NL] provides an existing solution for the model server API. This API has been implemented to connect seamlessly with the Graphical Language Server Protocol, which makes it a perfect starting point for the implementation of the TypeScript-based model management. It offers two ways to access the AST of a model. Firstly, via a client-server structure that communicates through JSON-RPC, and secondly, as a service integrated into the Langium language server.

The following sections discuss the requirements for the model server API in more detail. They focus on the requirements for model management and the requirements for the model service integrated within the Langium language server.

4.2.1 Requirements for the model management

The access to a model and its AST is a very important aspect of this thesis. The following requirements are particularly important for (**RQ1**):

- **open**: The model server has to provide functionality to load a file from the file system. A model can not be modified before the model server has loaded the associated file into its storage.
- **save**: The model server has to provide functionality to save a model in its current state to the file system.
- **close**: The model server has to provide functionality to close a model so it can no longer be modified. Upon closing, the model is removed from the model server's storage.
- **request**: The model server has to provide the functionality to request the current state of a model. If a model is not currently opened, it must be opened and loaded into the server's storage before returning the current model state.

To allow users to update the model's current state within the model server, a functionality intended for this purpose has to be implemented. This will be achieved using two different techniques:

- **update**: In this approach, the graphical client sends the entire updated model to the model server. Although this method is simple, it can lead to significant communication overload as the size of the request from the client to the server will increase with the size of the model.
- **patch**: In this approach, the graphical client sends the current model update by providing a JSON patch. With this, only the changes for a selected path must be sent from the client to the server, which keeps the communication from client to server very lightweight.

4.2.2 Requirements for the model service

In the current implementation of the model server API, the Langium language server is enhanced by a custom service called the `ModelService`. However, this `ModelService` is closely tied to CrossBreeze's specific implementation of the Langium language server. To make it easier to integrate the `ModelService` with other language servers, it's essential that it is decoupled from the language server and instead included as an external npm package.

4.3 TypeScript-based grammar language

Before defining the grammar language using TypeScript to create a metamodel, certain considerations must be taken into account. The following section presents these considerations. Additionally, the requirements for the grammar's new definition will be outlined.

In Section 2.2, a kernel version of Eclipse Ecore's metamodel has been presented. In this version, the main elements that define a metamodel have been highlighted. Therefore, these elements have to be included within the newly defined TypeScript-based grammar definition.

As the TypeScript grammar definition should be parsed and mapped into a Langium grammar (see Section 2.4), the structure of Langium's grammar definition must also be considered in the gathering of requirements of the new grammar definition. This includes, for example, the entry element rule. In the TypeScript grammar definition, it is required to be able to define the entry element, which defines which elements can sit on the root level of the model.

Considering both the Langium grammar definition and the kernel version of Ecore, the following requirements need to be fulfilled by the grammar:

- **Root/Entry element definition** : The grammar definition has to provide a notation to define a root/entry element.
- **Element definition**: The grammar definition has to provide a notation to define model elements.
- **Attribute definition**: The grammar definition has to provide a notation to define attributes of model elements.
- **Inheritance definition**: The grammar definition has to provide a notation to enable inheritance between different model elements.
- **Multiplicity definition**: The grammar definition has to provide a notation to define the multiplicity of an attribute. This multiplicity definition has to support at least the multiplicities: *exactly one* (1), *zero or one* (?), *zero or more* (*), and *one ore more* (+).
- **Reference definition**: The grammar definition has to provide a notation to define the following references:
  - **Containment Reference**: A containment reference is a reference to a model element that is contained within the referencing model element.
  - **Cross Reference**: A cross-reference is a reference to a model element within the model, meaning that the referenced element can exist independently without the referencing element.

- **Type alias definition**: The grammar definition has to provide a notation to define type alias elements.
- **Grammar validation**: It is required to define validation for the grammar definition to ensure that the parser is able to parse the definition into a valid representation of the grammar and transform it into a Langium grammar definition.

The definition of the TypeScript-based grammar language plays a vital role in answering (**RQ2**).

4.4 Generator

It would be useful to have a tool that combines the functionalities of the two previous artifacts, enabling the creation of a sample project using these artifacts. Thus, this section presents the requirements that need to be fulfilled to implement a generator that can perform this task efficiently.

As already mentioned, the implementation of a generator for this system consists not only of the initial project setup but also of the regeneration of the metamodel after changes to the definition file. The different types of requirements that need to be specified for the generators are discussed in the following sections.

To test the initial setup without creating a graphical model client, the generator should also create a Visual Studio Code extension [Mica]. This extension can be used to test the created metamodel in a textual notation, but on startup, it also starts the model server API, which can then be accessed via JSON-RPC calls.

4.4.1 Initial project properties

Before the generator can create an initial project, it is required to be able to gather the most important properties of the newly created project:

- **Name definition**: The generator has to include an option to set the name of the project that is set up.
- **Modeling Language definition**: The generator has to include an option to set the modeling language name.
- **File extension definition**: The option to set file extensions for language identification should be included in the generator.
- **Entry/Root element definition**: The generator has to include an option to set the modeling language's entry/root element name.
- **Reference Property definition**: The generator has to include an option to set the reference property of model elements. This reference property is used to identify elements in cross-references.

4.4.2 Parsing

The generator must be able to create the initial project and regenerate the project files according to the definition files. To be able to read the definition files, a parsing mechanism must be implemented.

In the context of this thesis, the generator needs to be able to read three different kinds of definition files: JSON configuration files, TypeScript files that hold the current metamodel definition in the TypeScript-based grammar language, and `.ecore` definition files. This leads to the following requirements for the parsing of files:

- **JSON parser**: The generator must be able to read JSON files and its configuration data.
- **TypeScript grammar language parser**: The generator must be able to parse the TypeScript-based grammar language so that it can be validated and transformed into a Langium grammar.
- **Ecore definition files**: To ease the transition from the Java-based metamodeling stack, which uses Ecore, to the TypeScript-based stack, the generator must be able to parse `.ecore` definition files so that these definitions can be transformed into the TypeScript-based grammar language.

4.4.3 Validation & Transformation

After the generator parses the definition files, it is required to validate whether the definition file has been correctly created. Therefore, the following requirement needs to be included:

- **Validation**: The generator must provide validation for the TypeScript-based definition language. The generator should provide the user with a helpful validation message upon validation errors.

Before and after the validation step, a transformation must be done inside the generator. This transformation turns the parsed definition file into a format that can be validated more easily. After the validation, it turns it into a format that can be easily used to generate new files.

- **Transformation**: The generator must provide a transformation from the parsed definition file to a format that can be easily used to generate different types of files (Langium files as well as TypeScript files).

4.4.4 File creation

Before a generator can create files, it is required to understand what files should be created. In the case of this thesis, multiple TypeScript files, Langium files, and JSON configuration files need to be created. A distinction needs to be made for these files: either they can be predefined using template files, or they need to be created from scratch based on the generator's inputs. Therefore, the following requirements need to be fulfilled:

- **Template file creation**: The generator must be able to create files from templates. In these templates, the generator has to be able to replace the elements according to the provided definition.
- **Scratch file creation**: The generator must be able to create files from scratch according to the current definitions/inputs of the generator. The most important files that need to be created this way include:
  - **TypeScript-based grammar language definition**: This type of file creation is optional for the case that the generator takes an `.ecore` definition file as input, and should create the TypeScript-based grammar definition from it.
  - **Langium Grammar Definition (`.langium`)**: The Langium grammar definition file is used by Langium to generate the metamodel from the definition. In this thesis, the Langium grammar must be created as a generic JSON grammar so that it can be easily reused in other contexts. This requirement is particularly important for research question (**RQ3**).
  - **Langium Services (`.ts`)**: During file creation, different Langium services need to be recreated according to the current definition of the language. The Langium serializer service is particularly important, as this service is always used in the communication between the language client and the server.

4.4.5 Package Installation & Build

As the generator should provide a ready-to-use solution right after project generation, the following requirements need to be included for the generator:

- **Install Packages**: After the project files have been successfully created/updated, the generator must provide the possibility to install all needed packages for the project.
- **Build Project**: After the packages have been installed, the generator must be able to build the project. After the build process is finished, a Visual Studio Code extension for the current project is ready to use.

4.5 Summary

This chapter presented the main idea of this thesis, followed by the requirements for the browser-based model management solution. Additionally, the requirements for the TypeScript-based grammar language and the generator functionality have been discussed. Moving forward, the subsequent chapter will focus on the solution concepts for these requirements.

CHAPTER 5

Concept

This chapter will present the solution concepts for the different requirements presented in Chapter 4.

5.1 Model Server API

As discussed in Section 4.2, the model server API must provide access to a model's Abstract Syntax Tree. While the concept of implementing the open, close, save, request, and update functionality is very straight-forward and therefore not discussed in more depth in this section, the concept of the patch functionality required various considerations which are listed in the following.

As a basis for the patch functionality, the model's AST has to be mapped into a JSON representation. Langium already provides a service to do that, but in this implementation, cross-references to elements inside another document are not mapped in a sufficient way, as they only save the path to the element inside the respective document without providing the information in which document the source element is. Therefore, a custom implementation of the JSON serializer service needs to be implemented. The concept of this implementation will be discussed in more detail in Section 5.3.4.

After a model has been serialized into a JSON format, it is possible to execute the JSON patch. However, for this step one needs to consider the existence of cross-references inside one JSON document to other elements inside the same or other documents. Typically, these cross-references are saved in the JSON document via a `$ref` property, which holds the path to the element that the reference refers to. In the custom JSON serializer, this property can be written in different manners according to the element's properties. The first manner is shown in Listing 5.1. In this case, the structure of the `$ref` property consists of the properties `__id` and `__documentUri`, which hold the reference property

element and the document URI of the source element. This is the preferred structure (which is also enabled per default).

```
"$ref": {
    "__id": "ID_OF_THE_REFERENCED_ELEMENT",
    "__documentUri": "DOCUMENT_URI_OF_THE_REFERENCED_ELEMENT"
}
```

Listing 5.1: Structure of `$ref` element if the referenced element includes the reference property

Unfortunately, it is possible that the referenced element does not have the referencing property; this causes the structure of the `$ref` property to consist of the path and the document URI of the source element, which can be seen in Listing 5.2.

```
"$ref": {
    "__path": "PATH_TO_THE_REFERENCED_ELEMENT_IN_THE_DOCUMENT",
    "__documentUri": "DOCUMENT_URI_OF_THE_REFERENCED_ELEMENT"
}
```

Listing 5.2: Structure of `$ref` element if the referenced element does not include the reference property

For the JSON patch, special handling must be considered for the second case, because the path of elements can change during a JSON patch operation.

Listing 5.3 shows an example JSON structure containing two `nodes` and a `nodeReference` that holds a reference to the second node.

```
{
    "nodes": [
        {"name": "Node1"},
        {"name": "Node2"}
    ],
    "nodeReference": [
        {
            "name": "nodeReference",
            "reference": {
                "$ref": {
                    "__path": "/nodes/1",
                    "__documentUri": "#"
                }
            }
        }
    ]
}
```

Listing 5.3: Example JSON structure

In the following the application and the thereby caused problems of selected patches are applied to Listing 5.3:

- **Delete Patch in Listing 5.4**: After this patch is executed, the nodes array only consists of the first node. Now it is impossible for the nodeReference to rebuild its reference, as the second node (`/nodes/1`) can not be found anymore. This leads to an invalid state inside the document. In this case, the problem can be fixed rather easily, as it would be possible to look for invalid references and remove the nodes that hold them. However, in case the deleted element is referenced in another document in the same manner invalid states can still appear.

```
{
    "op": "remove",
    "path": "/nodes/0"
}
```

Listing 5.4: Example JSON patch deleting the first node

- **Add Patch in Listing 5.5**: After this patch is executed, the nodes array consists of three nodes, with the node at position 1 being the newly created element. After rebuilding the references, the rebuilt reference holds the wrong node. Once again, it would be possible to recreate the reference by checking what operation is done during the patch, but this could lead to recursive rework, as each node that holds a corrupted reference could be referenced somewhere else. Therefore, the rework needs to be done not only for the initial reference but also for those references who are referencing the referenced element.

```
{
    "op": "add",
    "path": "/nodes/1",
    "value": {"name": "Node69"}
}
```

Listing 5.5: Example JSON patch adding a node on the second position

Patch Concept To mitigate the problems with path-referenced elements inside a JSON document, this thesis uses a new concept to ensure that the rebuilding of references works correctly. Within this concept, a service called the `PatchManager` is used, which has to perform multiple steps before and after a patch to ensure the references are not corrupted.

Before each JSON patch, the model that should be changed has to be serialized into JSON format. However, it is possible that the serialized model is not the only model that needs to be looked at during a JSON patch, as Langium provides support for references across the entire workspace. Using the `IndexManager`-service from Langium, it can be checked whether a document could be affected by a change of an element inside a selected document. With this, it is possible to collect all affected documents and their associated models and then serialize them into JSON format.

After the serialization, the PatchManager parses through each JSON document and adds UUIDs to every element that is of the type object, except for the elements holding a reference. This UUID is used to uniquely identify each object inside the JSON model.

Subsequently, the PatchManager collects all references inside each document and replaces their reference path with the UUID of the element at the reference's path. As an example, Listing 5.6 shows the JSON model from Listing 5.3 after the PatchManager prepared the model for the JSON patch.

```
{
    "__tmp_uuid__": "UUID1",
    "nodes": [
        {"name": "Node1", "__tmp_uuid__": "UUID2"},
        {"name": "Node2", "__tmp_uuid__": "UUID3"}
    ],
    "nodeReference": [
        {
            "__tmp_uuid__": "UUID4"
            "name": "nodeReference",
            "reference": {
                "__tmp_uuid__": "UUID5",
                "$ref": "UUID3"
            }
        }
    ]
}
```

Listing 5.6: JSON document after PatchManager prepared Listing 5.3 for JSON patch

Once the document has been prepared, patches can be executed and the previously mentioned problems with path-referencing problems can be avoided as can be seen in the following descriptions. For example, after the patch in Listing 5.4 is applied to the prepared model in Listing 5.6, the document would again consist of exactly one node. Contrary to the previous execution of the patch, this time it is still possible to recreate the reference.

To do that, the PatchManager first collects all objects that include a __tmp_uuid__-property. Next, it collects the path to each element with a __tmp_uuid__ and the value of the element holding the property. With these elements, it creates a Map, using the UUID as the key and the path and the value of the node as the value. Using this Map, the PatchManager parses through each JSON document and replaces every referencing element with the reference that was used before the PatchManager prepared the model, which consists of the document URI and either the referencing property or the path to the referenced element.

When this is done, all references have been correctly rebuilt, and all the PatchManager has left to do is remove the UUID properties from all objects and remove elements, that are referencing another element, which have not been correctly rebuilt, i.e. deleted.

Though the JSON references have been correctly rebuilt at this point, the `PatchManager` still needs to ensure, that the Langium references are rebuilt correctly. For this, the AST is parsed and every reference is replaced with the value of the referenced element. Following this, the `LangiumDocument` is back to a valid state inside the language server, but the changes have not been saved yet. To do that, the updated documents are serialized into the Langium grammar and then the documents are updated using the Langium `DocumentManager`.

Additionally, the `PatchManager` saves the current and previous state of the documents after each JSON patch, to provide a simple functionality to be able to undo or redo any change that has been done.

5.2 TypeScript-based grammar language

In the definition of the language concepts, it has to be ensured that the requirements defined in Section 4.3 are fulfilled while also using TypeScript-native notations. To illustrate the different language concepts, in Figure 5.1, an example metamodel can be seen, whose elements will be rebuilt using the TypeScript-based grammar language.

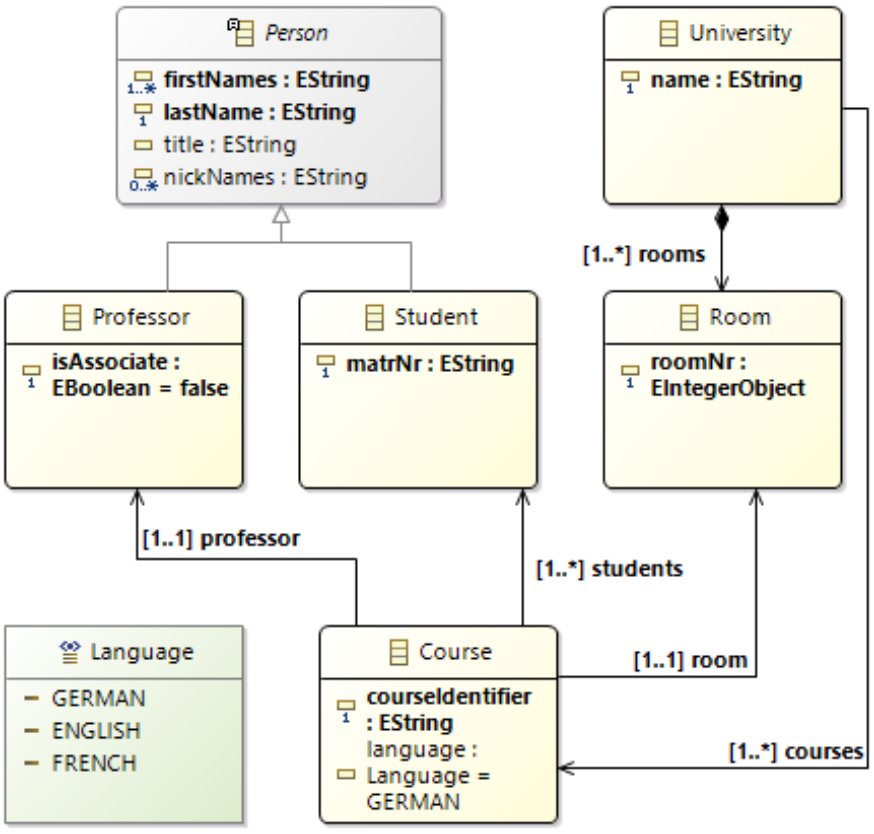


Figure 5.1: Example metamodel built using Ecore

To ease the understanding of the TypeScript-based grammar language that will be described in the following, a graphical representation of the metamodel can be seen in Figure 5.2. The metamodel consists of `Types` and `ModelElements`, which can be either of type `Class` or `Interface`. As seen, `ModelElements` can consist of multiple `properties`, while `Types` can consist of multiple `types`. It has to be noted that the `DataType` represents built-in data types (like `string` or `number`), constant data types (like actual strings or numbers, e.g., "ENGLISH" or 10), as well as complex data types, which includes `ModelElements` and `Types`.

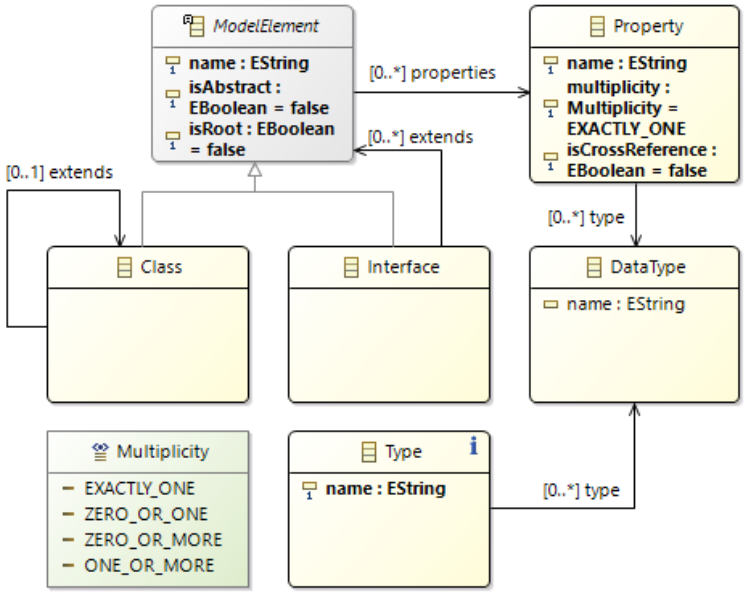


Figure 5.2: Graphical representation of the metamodel of the TypeScript-based grammar language

The TypeScript-native `class` and `interface` structures can be used to define model elements. Listing 5.7 demonstrates the use of both keywords. They can be used interchangeably, as both allow the definition of properties and their types. However, according to the needs of the language that needs to be defined using the TypeScript-based grammar language, one of them may be preferred. This is due to the limitations of the `extends` keyword in `classes` and the limitations of the usage of decorators within `interfaces`. In the following, next to the definition of language concepts, also the differences in the usage of `classes` and `interfaces` will be discussed.

```
class University {}
interface Person {}
```

Listing 5.7: Definition of model elements

To give meaning to model elements, it is necessary to be able to add attributes to them. This is again done using a default TypeScript notation by assigning properties to elements. As can be seen in Listing 5.8, the previously defined `University` element now contains a `name`, and the `Person` element contains an attribute `lastName`.

```
class University {
    name: string;
}
interface Person {
    lastName: string;
}
```

Listing 5.8: Definition of attributes for model elements

With this definition, it is possible to define which attributes a model element can have, but it is not clear how many attributes of each type the model element can have. For this, a notation for different types of multiplicities is introduced, as can be seen in Listing 5.9. The defined grammar supports four different types of multiplicity, which are listed in the following:

- **exactly one**: The `lastName` attribute is defined using only the property name and the property type. No additional information is given, thereby showing that each `Person` has exactly one name.

- **zero or one**: The `title` attribute is defined using the property type and the property name in combination with the question mark operator, which signals for the element to be optional.

- **one ore more**: The `firstNames` attribute is defined using the property name and the property type, which is a container of type Array. This signalizes, that the model expects one or more elements of the type of `nationality`.

- **zero or more**: The `nickNames` attribute is defined using the property name in combination with the question mark operator and the property type, which is a container of type Array. As the question mark operator signals for the attribute to be optional, and the Array type signalizes for one or more elements, the combination of these two elements leads to the multiplicity of zero or more elements.

```
class Person {
    lastName: string;
    title?: string;
    firstNames: Array<string>;
    nickNames?: Array<string>;
}
```

Listing 5.9: Definition of multiplicity for model elements

Up until now, in all definitions, the model elements consisted only of attributes that were simple types (e.g., `string`, `number`, `boolean`). However, the grammar should provide two possibilities to create a reference to another model element. This can be done using either containment references (see Listing 5.10) or cross-references (see Listing 5.11). The containment references can be easily built using the default TypeScript notation for properties. The annotation `@crossReference` is added to the grammar language to create the notation for cross-references. Unfortunately, annotations are only supported for classes. Therefore, if the model element is defined using an interface structure, a custom container type `CrossReference<T>` has been defined to be able to create cross-references in the interface context. By providing the annotation for a property or setting the type of a property to the container type, it is signalized that the property has a reference to an element that can exist on its own.

In the example shown in Listing 5.10, the model element `University` has a containment reference to `Room` elements. Due to the containment reference, it is clear that the `Room` can not exist on its own, but it can be created within a model element of type `University`.

```
class University {
    name: string;
    rooms: Array<Room>;
}
class Room {
    roomNr: number;
}
```

Listing 5.10: Definition of containment reference

In the example shown in Listing 5.11, the model element `Course` is defined using both a `class` and a `interface` structure. The cross-references signalize, that the `Room`, `Student`, and `Professor` can exist on their own, without being contained within an element of type `Course`.

```
class Course {
    courseIdentifier: string;
    @crossReference room: Room;
    @crossReference students: Array<Student>;
```

```
    @crossReference professor: Professor;
}
interface Course {
    courseIdentifier: string;
    room: CrossReference<Room>;
    students: Array<CrossReference<Student>>;
    professor: CrossReference<Professor>
}
```

Listing 5.11: Definition of cross-reference

The model element `Person` has been defined in Listings 5.7 to 5.9. In some cases, some specialization of this type of element may be needed. The TypeScript-native `extends` keyword can be used for this. In Listing 5.12, the model elements `Student` and `Professor` are defined, extending the base model element `Person`. A major advantage of using `interfaces` for defining model elements is the ability to extend multiple interfaces, which is not possible with `classes`.

```
class Student extends Person {
    matNr: string;
}
interface Professor extends Person {
    isAssociate: boolean;
}
```

Listing 5.12: Definition of sub-model elements

Sometimes, it may be necessary to define base elements but it should not be possible to create instances of them, meaning that the base elements are abstract. This can be done using the `abstract` keyword for model elements that have been defined using classes. For interfaces, this keyword can not be used. Therefore, a type named `ABSTRACT_ELEMENT` has been added to the grammar language to signalize for an interface model element to be abstract. In Listing 5.13, examples of the abstract notation for classes and interfaces are shown.

```
abstract class Person {
    lastName: string;
    title?: string;
    firstNames: Array<string>;
    nickNames?: Array<string>;
}
interface Person extends ABSTRACT_ELEMENT {
    lastName: string;
    title?: string;
    firstNames: Array<string>;
    nickNames?: Array<string>;
```

```
}
```

Listing 5.13: Definition of abstract model elements

Another requirement for the grammar language is the ability to create type alias elements, which can be done by the creation of union types. An example of these union types used in a TypeScript-native way can be seen in Listing 5.14, where the union type `Language` is created, which can be either `"GERMAN"`, `"ENGLISH"`, or `"FRENCH"`.

```
type Language = "GERMAN" | "ENGLISH" | "FRENCH";
```

Listing 5.14: Definition of type alias element

Using all discussed definitions, it is possible to define the elements which belong to a model, their multiplicity and how they are connected. However, until now it is impossible to define the actual structure of the model, i.e., it is not clear how a model can include the defined model elements. For this, it is necessary to be able to define an entry or root element, which is the element on the root level of the model.

To enable the entry/root-element functionality for classes, the annotation `@root` is added to the grammar language. Additionally, a custom type named `ROOT_ELEMENT` has been added to the grammar language for interfaces. To illustrate this functionality, in Listing 5.15, the root element `Model` is defined using the annotation, while in Listing 5.16, the root element `Model` is defined using the type definition.

```
@root
class Model {
    persons: Array<Person>;
    universities: Array<University>;
}
```

Listing 5.15: Definition of root model element using class

```
interface Model extends ROOT_ELEMENT {
    persons: Array<Person>;
    universities: Array<University>;
}
```

Listing 5.16: Definition of root model element using interface

After combining all the presented rules, the metamodel, which is illustrated in Figure 5.1, can be built using the TypeScript-based grammar definition as can be seen in Listing 5.17.

```
@root
class Model {
    persons: Array<Person>;
    universities: Array<University>;
}
abstract class Person {
    lastName: string;
    title?: string;
    firstNames: Array<string>;
    nickNames?: Array<string>;
}
class Student extends Person {
    matNr: string;
}
interface Professor extends Person {
    isAssociate: boolean;
}
class Course {
    courseIdentifier: string;
    language: Language;
    @crossReference room: Room;
    @crossReference students: Array<Student>;
    @crossReference professor: Professor;
}
class University {
    name: string;
    rooms: Array<Room>;
}
class Room {
    roomNr: number;
}
type Language = "GERMAN" | "ENGLISH" | "FRENCH";
```

Listing 5.17: Definition of the metamodel from Figure 5.1 in the TypeScript-based grammar language

The last requirement of the TypeScript-based grammar language is the validation requirement. For this, the following validation rules have been created:

- **The grammar definition must consist of exactly one root element**: If the grammar definition does not include a root element, it is unclear how the model can contain all defined model elements. In contrast, if the grammar definition includes multiple root elements, it is unclear which of the root elements is on the model's root level.

- **The grammar definition has to be serializable**: To be able to send the model from the server to the client, the model must be serializable.

5.3 Generator

As already mentioned in Section 4.4, the generator functionality has many requirements. In Figure 5.3, the workflow of the generator functionality, which will be discussed in the following subsections, is illustrated.

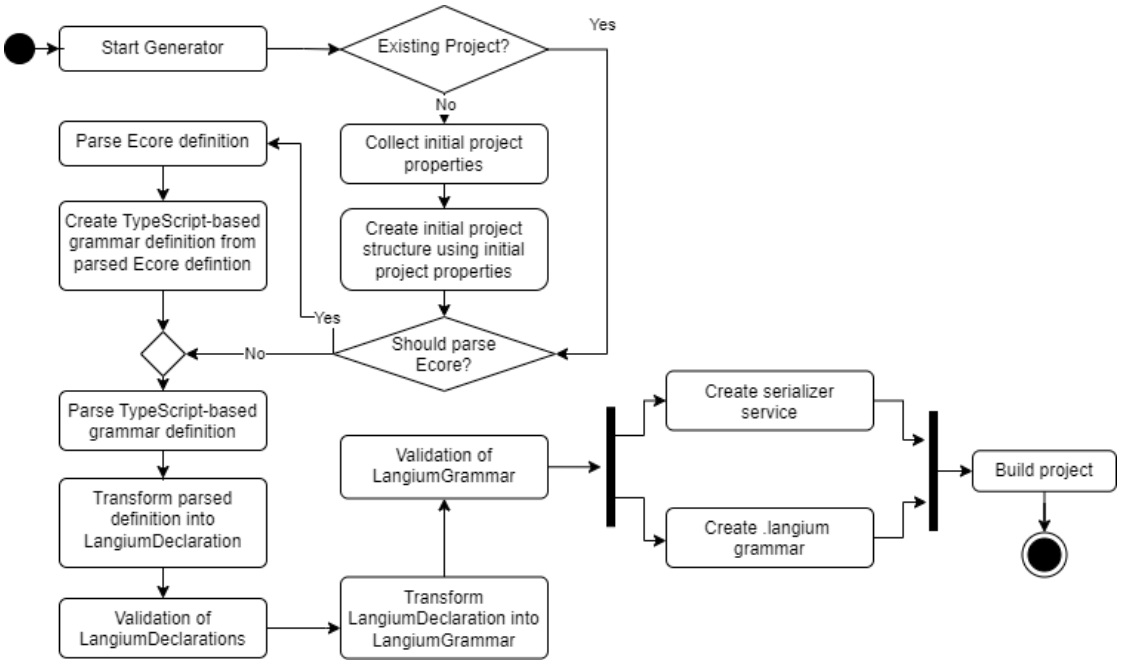


Figure 5.3: Workflow of the generator functionality

5.3.1 Initial project properties

The generator will use prompting in the Command Line Interface to gather the initial project properties, such as the project name, language name, and file extension. Once the generator is started, it needs to check whether it is being executed within an already existing project. In case it is not, the generator must prompt the user to provide input for all relevant project properties. The generator should also assign a default value to each property in case no input is provided.

5.3.2 Parsing

The parsing feature of the generator consists of three types of parsers: a JSON parser, a TypeScript parser, and an Ecore parser (which is a type of XML parser). As JSON format is a default structure in TypeScript, JSON files can be easily parsed by just

reading the file data. For the TypeScript parser and the Ecore parser, a more enhanced parsing functionality has to be implemented. Therefore, the generator will include two parser modules that implement the TypeScript parsing mechanism and the Ecore parsing mechanism. The generator checks before the parsing stage whether it has to parse an Ecore definition file. If that is the case, the generator parses and transforms the Ecore definition into the TypeScript-based grammar definition. Otherwise, the generator moves on to the parsing step of the TypeScript-based grammar language. The functionality of both parsing modules will be explained in the following paragraphs.

Ecore parser For this parsing module, the functionality of the npm package `fast-xml-parser`[1] will be utilized, to parse the `.ecore` file into a format that can be easily transformed into the required structure to be able to generate the TypeScript-based grammar language definition. The data structures that are used can be seen in Listings 5.18 to 5.21. As can be seen, this data structure collects information about the features of `EClasses`, `EAttributes`, `EEnums`, `EDataTypes`, and `EReferences`. The information of `EAttributes` and `EReferences` are collected together within the `EcoreAttribute` interfaces (Listing 5.20).

```
export interface EcoreDefinition {
  classes: EcoreClass[];
  dataTypes: string[];
  types: EcoreType[];
}
```

Listing 5.18: Defintion of the data structure used for the entire Ecore definition

```
export interface EcoreClass {
  attributes: EcoreAttribute[];
  extends: string[];
  instanceClassName?: string;
  instanceTypeName?: string;
  isAbstract: boolean;
  isInterface: boolean;
  isRoot: boolean;
  name: string;
}
```

Listing 5.19: Defintion of the data structure used for `EClasses`

```
export interface EcoreAttribute {
  changeable?: boolean;
  containment?: boolean;
  defaultValueLiteral?: string;
  derived: boolean;
```

[1] https://github.com/NaturalIntelligence/fast-xml-parser

```
  ID?: string;
  lowerBound: string;
  multiplicity: string;
  name: string;
  ordered?: boolean;
  reference: boolean;
  transient: boolean;
  type: string;
  unique?: boolean;
  unsettable: boolean;
  upperBound: string;
  volatile: boolean;
}
```

Listing 5.20: Defintion of the data structure used for `EAttributes` and `EReferences`

```
export interface EcoreType {
  name: string;
  types: string[];
}
```

Listing 5.21: Defintion of the data structure used for `EEnums`

TypeScript parser This parsing module will use the `typescript`[2] npm package to be able to parse elements according to the AST of the TypeScript file. To validate and transform the definition file into a Langium grammar, the parser creates a data structure to store parsed elements. The data structure to be used can be seen in Listings 5.22 to 5.25.

```
export interface Declaration {
  type: "class" | "type";
  name?: string;
  isAbstract?: boolean;
  decorators?: string[];
  properties?: Array<Property>;
  extends?: string[];
}
```

Listing 5.22: Data structures used by the parser - `Declaration`

```
export interface Property {
  name: string;
  isOptional: boolean;
  decorators: string[];
```

[2] `https://github.com/Microsoft/TypeScript`

```
  types: Type[];
  multiplicity: Multiplicity;
}
```

Listing 5.23: Data structures used by the parser - `Property`

```
export interface Type {
  type: "constant" | "simple" | "complex";
  typeName: string;
}
```

Listing 5.24: Data structures used by the parser - `Type`

```
export enum Multiplicity {
  ZERO_TO_N = "*",
  ONE_TO_N = "+",
  ONE_TO_ONE = "1",
}
```

Listing 5.25: Data structures used by the parser - `Multiplicity`

As the TypeScript-based grammar language, discussed in Section 5.2, supports a limited amount of elements of the TypeScript definition, special handling needs to be added for the additional elements.

The base elements the parser needs to be able to understand are classes, interfaces, and types. These elements are parsed into the `Declaration` interface. As classes and interfaces are used to create model elements, they are summarized into the same type `class` in the data structure. Additionally, for type aliases, the type `type` is used as seen in Listing 5.22. The `name` attribute of the `Declaration` should be self-explanatory. The `isAbstract` property signalizes whether class definitions include the abstract keyword, while for interfaces, it is checked if they are extending the `ABSTRACT_ELEMENT` interface. For types, this property is always true, as types are used to create unions that represent abstract elements.

The `decorators` property is used to collect the annotations from classes, and for interfaces, it is used to check if it extends the interface `ROOT_ELEMENT`. In type alias declarations, this property is ignored.

The `extends` property collects for each class and interfaces the classes and interfaces they are extending. This property ignores the elements `ABSTRACT_ELEMENT` and `ROOT_ELEMENT` as they are handled within other properties of the `Declaration` interface.

The `properties` property is used to store the properties of a class, interface, or type. To be able to parse properties, there is also special handling needed.

The `name`, `isOptional` and `multiplicity` properties of a `Property` should be self-explanatory.

The `decorators` property checks if a property is of type `CrossReference` or if the property has the annotation `@crossReference`.

The `types` property collects the types of an attribute. As it is possible that the type is defined using a union type, this property is an array type. The `Type` interface consists of two attributes: `typeName`, which saves the name of the type, and `type`, which is used to save whether the type argument is a simple type (`string`, `number`, `boolean`), complex type (`type`, `class`, `interface`) or a constant type (fixed string, fixed number or fixed boolean).

5.3.3 Validation & Transformation

There are two different kinds of transformations that are implemented in the generator workflow. In the first transformation, the `Declarations` that have been created by the parser have to be transformed into an easily validatable format. The new format that can be seen in Listing 5.26 is very similar to the initial `Declaration`-format, which was shown in Listing 5.22. The only difference in the structure is that the `extends`-property has been replaced with the `extendedBy`-property. So instead of the lower-level model elements knowing which elements they extend, the upper-level model elements know by which elements they are extended. This structure facilitates building a tree model, which eases the transformation later on to the Langium grammar.

```
export interface LangiumDeclaration {
  type: "class" | "type";
  name?: string;
  isAbstract?: boolean;
  decorators?: string[];
  properties?: Array<Property>;
  extendedBy?: string[];
}
```

Listing 5.26: Data structures used in first transformation

This first transformation consists of the following steps:

1. **Map `Declaration` to `LangiumDeclaration`**: In this step, the `type`, `name`, `isAbstract`, and `decorators` properties are mapped one-to-one to the respective properties in the `LangiumDeclaration`. The `extendedBy`-property is initially left as an empty array. Additionally, the properties of each `Declaration` are combined with the properties of the element it is extending.

2. **Fill the `extendedBy`-property**: For this, the initial declarations are used to search for elements that extend other elements, and according to this property, in the newly created `LangiumDeclarations` the `extendedBy`-property is filled with the associated elements.

3. **Remove properties from abstract elements**: As these abstract elements can not be instantiated on their own, it is not necessary for them to hold properties. Furthermore, if they are extended by another model element, the extending model element already holds the required properties after step one of the transformation. Therefore, in the last step of this transformation, the abstract elements are removed.

After the transformation is completed, the definition can be validated. As declared in Section 5.2, the definition is only allowed to hold exactly one root element. To validate this, the validator iterates through all `LangiumDeclarations` and counts the elements that include the root-element decorator. If the amount of elements equals one, the validation is successful.

Following this first validation part, the second required transformation is performed to transform the `LangiumDeclarations` into a `LangiumGrammar`, whose data structure and sub-elements can be seen in Listings 5.27 to 5.31. The `Multiplicity` and `Type` types are the same as in the previous `Declaration` definition.

```
export interface LangiumGrammar {
  entryRule: EntryRule;
  typeRules: Array<TypeRule>;
  parserRules: Array<ParserRule>;
}
```

Listing 5.27: Data structures used in second transformation - `LangiumGrammar`

```
export interface EntryRule {
  name: string;
  definitions: Array<Definition>;
}
```

Listing 5.28: Data structures used in second transformation - `EntryRule`

```
export interface TypeRule {
  name: string;
  definitions: Array<Type>;
}
```

Listing 5.29: Data structures used in second transformation - `TypeRule`

```
export interface ParserRule {
  name: string;
  isAbstract: boolean;
  extendedBy: string[];
  definitions: Array<Definition>;
}
```

Listing 5.30: Data structures used in second transformation - `ParserRule`

```
export interface Definition {
  name: string;
  type: Type[];
  multiplicity: Multiplicity;
  crossReference: boolean;
  optional: boolean;
}
```

Listing 5.31: Data structures used in second transformation - `Definition`

As can be seen in Listings 5.27 to 5.31, this new data structure has three different types of rules. The steps for this second transformation are the following:

1. **Create the `EntryRule` from the `LangiumDeclaration`**: The array of `LangiumDeclarations` is searched for the element that contains the `@root` decorator. The properties of the `LangiumDeclaration` are mapped to definitions.
2. **Create array of `TypeRule` from the `LangiumDeclaration`**: A `TypeRule` represents an unassigned rule call. This transformation itself consists of three steps:
   a) Map all elements of type `type` to `TypeRules`.
   b) Map all elements of type `class`, which are abstract to `TypeRules`.
   c) Search all properties of all elements of type `class`. If a property has a type that consists of multiple different types (i.e., union type), and there does not already exist a `TypeRule` that represents this union type, create a `TypeRule` for exactly this union type.
3. **Create array of `ParserRule` from the `LangiumDeclaration`**: Search for all elements that are of type `class`, do not include the `@root`-decorator and are not abstract. For all these elements, create a mapping to the `ParserRule` interface. Following the creation of the `ParserRules`, check if each definition includes the reference property (see Section 5.1). If the property is not included, add a definition for it. After that, iterate through all definitions of each `ParserRule`, and if a definition has multiple different types, replace this type with the respective `TypeRule` representing this type.

Following this transformation, a second validation phase is started to check whether the grammar can be serialized. Here, it is necessary to define the criteria for determining if a `LangiumGrammar` can be serialized.

- Rules without definitions are serializable.
- Rules that include only optional definitions are serializable.
- Rules that include only definitions whose type is simple or constant are serializable.

- Rules with definitions whose type is complex (i.e., other model elements, type aliases) are only serializable if the complex type is serializable.

The concept to enable these validations is to create an initial serializable set (which includes `string`, `number`, and `boolean` types) and iteratively extend this set. After each iteration, it is checked whether the set has changed between the start and end of the iteration, and if so, a new iteration is started. After the loop has stopped, i.e. no new element was added in an iteration, it is checked if all model elements are in the serializable set. If that is the case, the definition is serializable, and the validation is successful.

5.3.4 File creation

File creation is a key aspect of the generator's functionality. It consists of creating files using templates and the current definition file.

Template File Creation For the template file creation, the files that need to be created can be predefined according to a certain structure, which includes wildcard phrases. These phrases are replaced with the actual needed value during the file creation process. The template file creation step is executed only within the first execution of the generator after the initial project properties have been defined as can be seen in Figure 5.3. In Listing 5.32 an example template can be seen.

```
export class <%= LanguageName %>Example {
   getExampleString() {
        return "Example";
   }
}
```

Listing 5.32: Template file including wildcard phrase

In this example, a class is created. Notice that before the phrase `Example` in the class declaration, there is a string `<\%= LanguageName \%>`. This string is used as an identifier for the generator. It tells the generator to replace the identifier with the actual language name. For instance, if the language name was "WorkflowLanguage", the class name would be `WorkflowLanguageExample`.

In the scope of this thesis, the following seven identifiers are handled by the generator:

- `<\%= RawLanguageName \%>`: Replaced by the language name including white-spaces.
- `<\%= LanguageName \%>`: Replaced by the language name without white-spaces.

- `<\%= language-id \%>`: Replaced by the language name, where white spaces have been replaced with hyphens and the characters have been transformed to lowercase.
- `<\%= EntryElementName \%>`: Replaced by the entry/root element name.
- `<\%= ReferenceProperty \%>`: Replaced by the reference property, defined in the generator config file.
- `<\%= file-extension \%>`: Replaced by the defined file-extension.
- `<\%= extension-name \%>`: Replaced by the project name.

Scratch File Creation The second type of files that need to be created in the generation process need to be regenerated on each change of the definition file. This kind of file creation can be executed within three different contexts in the generator workflow: First, it can be optionally executed, if Ecore is used for the metamodel definition to create the TypeScript-based grammar definition file. Secondly, it is required to be executed to create the `.langium` grammar definition. Finally, it is also required to be executed to create the serializers for the grammar language. The scratch file creation is done either after parsing the Ecore definition or after the validation of the `LangiumGrammar` has been successful.

If Ecore is used for the definition, the parsing mechanism, which has been explained in Section 5.3.2, puts the definition into the `EcoreDefinition` data structure (see Listing 5.18) that can be used to create the TypeScript-based grammar language definition. For all properties of the `EcoreDefinition`, either classes, interfaces or types must be created considering the concept of the TypeScript-based grammar language, which has been described in Section 5.2. After the TypeScript definition file has been created, the TypeScript parser parses the definition and transforms it into the `LangiumGrammar` data structure.

After this transformation step (explained in Section 5.3.3), the current language definition is in a format that can be easily transformed into a Langium grammar. An important aspect of the grammar definition created in this step is that it should be a generic JSON grammar. Therefore, for each element, not only its attributes are included in the language definition, but also its type. In Listing 5.33, an example definition in the Langium grammar is shown.

```
Person: '{' '__type' ':' 'Person', 'age' ':' age=INT '}'
```

Listing 5.33: Example parser rule for a model element `Person`

There do exist three types of elements in the current `LangiumGrammar` definition: `EntryRule`, `TypeRule`, and `ParserRule` elements. For every type of rule, a different handling in the mapping to the Langium grammar needs to be added:

- **`EntryRule`**: The entry rule is the entry element rule for the Langium grammar. As explained in Section 2.4, the entry rule has to start with the identifier `entry`, followed by the rule name. In the body of the entry rule, the structure of the model is defined. The structure has to be in a valid JSON format; therefore, an opening or closing curly bracket is added before the first and after the last attribute definition. Furthermore, colons are added between the different attribute definitions.
- **`TypeRule`**: The type rule is used for unassigned rule calls. Therefore, creating a correct JSON grammar is unnecessary for this type of rule, as this should be handled within the child rule elements.
- **`ParserRule`**: The parser rule is used to define the correct structure of a rule element. The parser rule has the same mapping procedure as the `EntryRule`, with the only difference being that the `entry` keyword is left out in the mapping of the `ParserRule`.

In the data structure of the `LangiumGrammar`, the types of the definitions are saved as the TypeScript types. The `.langium` grammar can not handle the assignment to TypeScript types but rather expects terminal rule, type rule, or parser rule assignments.

Furthermore, if the defined type is a complex type, a parser rule must exist for this type. Therefore, for complex types, no special handling is required. However, creating a mapping for the simple types, which include the data types string, number, and boolean, is necessary. For these elements, the following terminal rules have to be created:

- **`boolean`**: terminal `LANGIUM_BOOL` returns boolean: /true|false/;
- **`number`**: terminal `LANGIUM_INT` returns number: /(-)?[0-9]+(.[0-9]*)?/;
- **`string`**: terminal `LANGIUM_ID`: /[\w_-]+/;

After adding these terminal rules, an attribute of type number can be assigned as seen in Listing 5.33.

The serializer is the second type of file that needs to be created after each change to the definition file. To do that, a serialize function is created for every type of rule, and according to the type of each definition, the respective serialize function is called, with the exception of simple and constant types, as they can be directly used in the serialization.

5.3.5 Package Installation and Build

After all files have been created, the generator executes the `npm install` command inside the project. Following that, the `npm run build` command is executed, which creates the build.

If both actions are successful, a ready-to-use extension can be selected and started in the Visual Studio Code debugging window.

5.4 Summary

In this chapter the solution concepts for the requirements that have been presented in Chapter 4 have been discussed. To achieve this, the challenges that some requirements may face have been identified and concepts for the solution of them have been presented. Additionally, the concept for the TypeScript-based grammar language has been defined. In the following chapter, the actual implementation of the concepts discussed in this section will be presented.

CHAPTER 6

Implementation

In this chapter, the implementation of the model server API and the generator will be presented.

6.1 Model Server API

The model server API allows graphical editors to access the Abstract Syntax Tree of a Langium language server. This API provides the functionalities to open, close, load, save, and update a document.

As the starting point of the implementation, the model server API from CrossBreeze [NL] has been used. In their implementation, the `ModelService` component is responsible for the loading, saving, and manipulation of the model state inside a `LangiumDocument`. This `ModelService` can be utilized in two different ways: Either via an RPC connection to the `ModelServer`, which forwards the actions to the `ModelService`, or it can be directly integrated into Langium as a custom added service.

As part of the implementation of the model server API, the `TextDocuments` service, which is a manager for simple text documents, has been extended so that it is possible to invoke events from within the language server. This service already provides different kinds of events that are fired whenever a `TextDocument` has been opened, updated, saved, or closed. It also holds the state of documents that are currently open. The name of this extended service is `OpenableTextDocuments`.

Another service that has been created in CrossBreeze's solution is the `OpenTextDocumentManager`, which is a manager class that supports the handling of text documents by providing methods to open, update, save, and close documents.

6.1.1 Model Server API Integration

One of the requirements of the model server API is its easy integration with Langium. Therefore, the model server API is implemented as an npm package.

This package consists of a module file, which defines extensions to the default Langium services, including language-specific extensions and extensions that enhance the functionalities of the language server in general.

In Listing 6.1, the definition for the `ExtendedLangiumServices` can be seen. This extension is created to add two different kinds of serializers: The `JsonSerializer`, which is essential for the implementation of the JSON patch functionality in the model server, and the `Serializer`, which is responsible for the transformation of the AST of a model into its textual representation.

```
export interface ExtendedLangiumServices extends LangiumServices {
    serializer: {
        JsonSerializer: JsonSerializer;
        Serializer: Serializer<AstNode>;
    };
}
```

Listing 6.1: Definition of the `ExtendedLangiumServices`

Due to the addition of the `ExtendedLangiumServices`, the default implementation of `ServiceRegistry`, which provides access to language-specific services, also needs to be extended. To determine which services to provide based on the language specification, the `ServiceRegistry` retrieves the necessary services using the URI of the model. In Listing 6.2, the definition of the new `ExtendedServiceRegistry` can be seen. The implementation of the new service registry is the same as the implementation of the base registry. Therefore, in the `register` and the `getServices` functions, the implementation of the base registry are called respectively.

```
export interface ExtendedServiceRegistry extends ServiceRegistry {
    register(language: ExtendedLangiumServices): void;
    getServices(uri: URI): ExtendedLangiumServices;
}
```

Listing 6.2: Definition of the `ExtendedServiceRegistry`

To be able to add the newly created `ExtendedServiceRegistry` to a language server built with Langium, a custom type named `AddedSharedServices` is created, which consists of the key `ServiceRegistry` and the type `ExtendedServiceRegistry`.

Another definition in the model module is the `AddedSharedModelServices`, which can be seen in Listing 6.3. This definition is used to add the `ModelService`, as well as the custom services `OpenTextDocumentManager` and `OpenableTextDocuments` to the language server.

```
export interface AddedSharedModelServices {
    workspace: {
```

```
        TextDocuments: OpenableTextDocuments<TextDocument>;
        TextDocumentManager: OpenTextDocumentManager;
    };
    model: {
        ModelService: ModelService;client
    };
}
```

Listing 6.3: Definition of the `AddedSharedModelServices`

As can be deduced, the `AddedSharedServices` type is used to extend the language-specific services, while `AddedSharedModelServices` is used to extend the language server capabilities in general. Details on how these two definitions are added to the language server will be given in later sections.

6.1.2 Model Service functionalities

The base functionalities of the model server API are implemented in the `ModelService` as well as the `OpenableTextDocuments` and `OpenTextDocumentManager`.

Each function in the `ModelService` expects the URI of the document, for which an action should be executed, as a parameter. A second optional parameter that can also be provided for each function call is the client kind. It should be noted that the current implementation supports two kinds of clients: `text` and `glsp`, which is the graphical client.

In the following, the functionalities of the model server API are described in more detail.

open In order to open a document, the `ModelService` calls the `open`-method of the `OpenTextDocumentManager`. The parameters for this call include the URI of the file that should be opened, the language ID, and from which kind of client the request is coming. The `TextDocumentManager` checks whether a document with the provided URI is already contained in its state. If so, the document has already been opened, and there is nothing left to do. Otherwise, the text document is read from the file system, and its current state is saved in the state of the `TextDocumentManager`.

close To close a document, the `ModelService` calls the `close`-method of the `OpenTextDocumentManager`. The parameters for this call include the URI of the file that should be closed and the client kind. The `close`-function checks whether the `TextDocumentManager` holds a currently opened version of the provided URI. If so, the document is removed from the currently opened documents in the state.

request The request functionality is used to get the current state of a document. The `ModelService` opens the document, using its previously mentioned `open`-method. Subsequently, it uses `LangiumDocuments`-service to get the `LangiumDocument`-instance and returns its value.

save The save functionality is used to save the model's current state to the file system. The save function can be called with an optional parameter that holds the state that should be saved to the file system. If this parameter is undefined, the `ModelService` requests the current state of the model using the `LangiumDocuments`-service and saves the current state of the `LangiumDocument` instance to the file system.

update The update functionality offers two different ways to update the model. The reason for this is that the model server API has been implemented in cooperation with another master's thesis, which focuses on the implementation of a textual-graphical-hybrid model editor. As for this thesis, only the update from the graphical client is of importance, only this part of the update will be explained.

The update function expects the updated model as a parameter. The parameter can be sent either in textual form or as the complete AST of the model. If the model is sent as an AST, it is first serialized into a textual representation. Following that, the update method for the `OpenableTextDocuments`-service and the `OpenTextDocumentManager`-service is called to update the internal state. To update the AST, the update method from the Langium-`DocumentBuilder` is called. Finally, the current state of the model is returned.

patch The patch functionality expects as a parameter the JSON patch that should be executed on the current state of the document. This functionality is implemented in the `applyPatch` function of the `PatchManager`-service. The function starts by collecting all documents that could be affected by a change in the document with the provided URI. The implementation of this collection can be seen in Listing 6.4.

```
async collectAffectedDocuments(docs: Set<string>, uri: string) {
    docs.add(URI.parse(uri).toString());
    for (const doc of this.documents.all) {
        await this.indexManager.updateReferences(doc);
        if (this.indexManager.isAffected(doc, docs)) {
            if (!docs.has(doc.uri.toString())) {
                docs.add(doc.uri.toString());
                docs = new Set([
                    ...(await this.collectAffectedDocuments(docs, doc.
    uri.toString())),
                ]);
            }
        }
    }
    return docs;
}
```

Listing 6.4: Collection of documents that could be affected by a change

As can be seen, the function checks recursively if, for the current documents in the set, the possibility exists that they could be affected by a change to these documents.

After the URIs have been collected, for each URI,the `OpenTextDocumentManager` opens the document, and following that, the `LangiumDocuments`-service is used to load the current AST of the model inside the document. In succession, using the `ServiceRegistry`, the `JsonSerializer` for each document is loaded and the `serialize` function is used to transform each respective AST into JSON format. The combination of URI and JSON format is saved within two Maps, which are needed, as one of them is used to execute the required JSON patches, while the other one is utilized to create a JSON patch that represents the current previous version of the model. This eases the implementation of the redo/undo functionality in the `PatchManager`. The described procedure can be seen in Listing 6.5.

```
for (let uri of documents) {
    let documentUri = URI.parse(uri).path;
    await this.documentManager.open(documentUri, undefined, client);
    const document = this.documents.getOrCreateDocument(URI.parse(
    documentUri));
    const jsonSerializer = this.shared.ServiceRegistry.getServices(
        URI.parse(documentUri)
    ).serializer.JsonSerializer;
    documentMap.set(
        documentUri,
        JSON.parse(jsonSerializer.serialize(document.parseResult.value
    ))
    );
    originalDocumentMap.set(documentUri, document.textDocument.getText
    ());
}
```

Listing 6.5: Functionality to load document URIs with models in JSON format into two Maps

As discussed in Section 5.1, before the JSON patch can be executed, a UUID must be added to each element in the model. This is done using the `addUUID` function, whose implementation can be seen in Listing 6.6. The function defines another function `visit` that recursively visits child elements of objects in the JSON object and adds the `__tmp_uuid__` property to those elements that are of type object and whose parent key is not `$ref`.

```
export function addUUID(map: Map<string, any>) {
    function visit(obj) {
        if (obj && typeof obj === "object") {
            for (let key in obj) {
                if (obj[key] && obj.hasOwnProperty(key) && key !== "
    $ref") {
                    visit(obj[key]);
                }
                obj["__tmp_uuid__"] = uuidv4();
            }
        }
    }
    map.forEach((val) => visit(val));
```

```
}
```

Listing 6.6: Functionality that adds temporal UUIDs to the JSON model

After the UUIDs have been added to the respective elements in the JSON object, the references inside the document must be changed so that they use the newly added UUIDs instead of the initial structure to represent a reference. In Listing 6.7, the source code on how to replace the initial references can be seen. Once again, a nested function `visit` is created, which recursively visits each object and its child elements. If the key of a child element is `$ref`, then the `getReferenceUUID` (see Listing 6.8) function is called to find the referenced element and replace the current `$ref` value with the `__tmp_uuid__` value of the reference.

```
export function updateReferences(map: Map<string, any>) {
    function visit(obj, initial) {
        if (obj && typeof obj === "object") {
            for (let key in obj) {
                if (key === "$ref") {
                    let val = getReferenceUUID(
                        obj[key].__documentUri
                        ? map.get(obj[key].__documentUri)
                        : JSON.parse(JSON.stringify(initial)),
                        obj[key]
                    );
                    obj[key] = val ? val : obj[key];
                } else if (obj.hasOwnProperty(key)) {
                    visit(obj[key], initial);
                }
            }
        }
    }
map.forEach((value) => visit(value, value));
}
```

Listing 6.7: Update the references inside the JSON models after adding the UUIDs

```
export function getReferenceUUID(json, ref: any) {
    if (ref.__id) {
        let nodes = findNodes(json, `__id`, ref.__id) ?? [];
        if (nodes[0]?.value) {
            return nodes[0].value["__tmp_uuid__"];
        }
    } else {
        const path = ref.__path.substring(ref.__path.indexOf("/") + 1)
    .split("/");
        for (let pathComponent of path) {
            json = json[pathComponent];
        }
        return json["__tmp_uuid__"];
    }
}
```

Listing 6.8: Get the UUID of the referenced element

In the `getReferenceUUID` function, the utility function `findNodes` is used, which searches the JSON object for elements with an `__id` property. If this property is found, it is checked if the value of this property is the same as the provided one (`ref.__id`). The function returns the first element that fits the criteria.

After these steps are done, the JSON objects are ready to be patched. However, for some kinds of patches, it is necessary to make an adjustment to the patch value. If the patch executes a `replace` operation, a reference to the replaced element can get lost, as the replacing value does not hold the UUID of the initial reference. In such a case, before the patch can be executed, the value of the patch is altered so that it holds the UUID of the initial value. The UUID is found using the `findUUID` function, whose implementation can be seen in Listing 6.9.

```
export function findUUID(json, path) {
const pathComponents = path.substring(path.indexOf("/") + 1).split("/"
    );
    for (let pathComponent of pathComponents) {
        json = json[pathComponent];
    }
    return json["__tmp_uuid__"];
}
```

Listing 6.9: Get the UUID of the element that should be replaced

After the patch values have been updated, the patch can be executed. In Listing 6.10, the code for the patch functionality can be seen. For the execution of the patch, an external library called `fast-json-patch`[1] is used.

```
patch.forEach((patchOp) => {
    if ((patchOp.op === "replace" || patchOp.op === "add") &&
        typeof patchOp.value === "object") {
        if (patchOp.op === "replace") {
          patchOp.value["__tmp_uuid__"] = findUUID(
            JSON.parse(JSON.stringify(documentMap.get(_uri))),
            patchOp.path
          );
        } else if (patchOp.op === "add") {
          patchOp.value["__tmp_uuid__"] = uuidv4();
        }
    }
    result = jsonpatch.applyOperation(documentMap.get(_uri), patchOp);
});
```

Listing 6.10: Execute the JSON patch

After the JSON patch is executed, the JSON objects must be rebuilt. This includes rebuilding the references to the previous structure, which consists of the `__documentUri` and either the `__id` of the referenced element or the `__path` to the referenced element.

[1] `https://github.com/Starcounter-Jack/JSON-Patch`

This is implemented in the `rebuildReferences` function, which can be seen in Listing 6.12. The function starts by collecting all nodes that include a property with the name `__tmp_uuid__`. Following that, the found nodes are mapped to elements that consist of the path and value of the found node. Here, the value of the node is provided by the `getNode` function whose implementation can be seen in Listing 6.11.

```
export function getNode(json, path) {
    for (let pathComponent of path.split("/")) {
        if (pathComponent) {
            json = json[pathComponent];
        }
    }
    return json;
}
```

Listing 6.11: Functionality to get the value of the element by searching the JSON object with the given path

After the nodes have been collected and mapped into their path and value structure, the rebuild functionality can be executed. As can be seen in Listing 6.12, the nested function `visit` is used to recursively visit all elements inside a JSON object and checks for the `$ref` property. If this property is found, the prepared nodes are searched, and if a node is found whose `__tmp_uuid__` is the same as the `$ref`, then the UUID is replaced with the `__documentUri` and the `__id` or `__path` of the found element. If no element is found, the value of the `$ref` property is set to an empty object.

```
export function rebuildReferences<T extends AstNode>(map: Map<string,
    any>) {
  let nodes: Record<string, any> = {};
  map.forEach((val, key) => {
    const tmpNodes = (nodes[key] = findNodes(val, "__tmp_uuid__") ??
    []);
    nodes[key] = tmpNodes.map((node) => ({
      path: node.path,
      value: getNode(val, jsonPathToJsonPatchPath(node.path)),
    }));
  });

  function visit(obj, docUri) {
    if (obj && typeof obj === "object") {
      for (let key in obj) {
        if (key === "$ref" && typeof obj[key] !== "object") {
          let node;
          let nodeKey;
          for (let _key in nodes) {
            node = nodes[_key].find(
              (node) => node.value.__tmp_uuid__ === obj[key]
            );
            if (node) {
              nodeKey = _key;
```

```
            break;
          }
        }
        if (node) {
          obj[key] = {
            __id: node.value.__id,
            __path: node.value.__id
              ? undefined
              : jsonPathToJsonPatchPath(node.path),
            __documentUri: nodeKey === docUri ? undefined : nodeKey,
          } as unknown as T;
        } else {
          obj[key] = {};
        }
      } else if (obj && obj.hasOwnProperty(key)) {
        obj[key] = visit(obj[key], docUri);
      }
    }
  }
  return obj;
}
map.forEach((value, key) => visit(value, key));
}
```

Listing 6.12: Functionality to rebuild the references to the previous reference structure

Once the references have been rebuilt, the JSON object needs to be cleaned. This includes removing all UUIDs and elements whose references were not rebuildable (which means that the referenced element has been deleted in the patch). An element that could not be rebuilt can be recognized as an empty object by the `$ref` value.

In succession to the cleanup, the Langium references need to be reconstructed. For this, the three Langium services `AstNodeLocator`, `NameProvider`, and `LangiumDocuments` are needed. Here, the `AstNodeLocator` service is used to get an AST node or the path to an AST node. Additionally, the `NameProvider` service is used to get the `name` of an AST node; however, this service will be extended by the generator to return the value of the `referenceProperty` of an AST node. Finally, the `LangiumDocuments` service has already been described in Section 2.4.

The `rebuildLangiumReferences` function consists of four nested functions:

- **`linkNode`**: This function recursively visits child elements of AST nodes to recreate the Langium references. It also makes sure that the reconstructed references are correct Langium nodes, i.e., it adds the `$container`, `$containerProperty` and `$containerIndex` properties to the reconstructed references.
- **`reviveReference`**: This function transforms a JSON reference into a Langium reference.

- **getRefNode**: This function is used to search for an AST node, given a reference element, which consists of a `__documentUri` and a `__id` or `__path`.
- **getAstNodeById**: This function expects the root AST node of a document and an id as input parameters. Using the Langium utility function `streamAst`, a stream of AST nodes is created, which is searched for the AST node with the given id.

After the Langium references have been restored, the AST is serialized, and an update similar to the discussed `update` functionality of the `ModelService` is executed.

Finally, after each `LangiumDocument` has been updated, the state before and after the patch is saved in a redo/undo Map, which is used in the redo/undo implementation of the `PatchManager`.

redo and undo To redo or undo a patch, the `ModelService` calls the respective functions inside the `PatchManager`. In the `PatchManager`, it is checked if there does exist a redo or undo action by searching the corresponding Maps. If a patch is found, the `redoUndoPatch` function is called, whose implementation can be seen in Listing 6.13.

```
async redoUndoPatch(map: Map<string, string>, uri: string, client?:
    string) {
    let retVal;
    for (let [key, value] of map.entries()) {
      await this.documentManager.open(
        URI.parse(key).path,
        undefined,
        client ?? "glsp"
      );
      const document = this.documents.getOrCreateDocument(URI.parse(
    key));
      await this.documentManager.update(
        URI.parse(key).path,
        document.textDocument.version + 1,
        value,
        client ?? "glsp"
      );
      await this.documentManager.save(URI.parse(key).path, value);
      if (URI.parse(key).path === URI.parse(uri).path) {
        await this.documentBuilder.update([URI.parse(key)], []);
        retVal = document.parseResult.value;
      }
    }
    return retVal;
}
```

Listing 6.13: Execution of redo/undo patch for the given Map of redo/undo actions

6.2 Generator

The generator consists of multiple implementation modules. The following sections will explain the implementation of all the generator modules in more detail.

6.2.1 Parsing

There are two modules for parsing: the Ecore parsing module, which can be used optionally, and the general parsing module, which is used for different types of parsing, including JSON, TypeScript-config, and TypeScript-based grammar language parsing. In the following paragraphs, the implementation of these parsing modules will be presented.

Ecore parsing The Ecore parsing module is an extra feature of the implementation. It is used to parse Ecore definitions into a format that can be used to easily generate a definition in the newly created TypeScript-based grammar language.

The implementation of the Ecore parser consists of an entry function, which uses the `fast-xml-parser` library to parse the `.ecore` definition into an easily iterable structure. Listing 6.14 shows the implementation part of the entry function for the Ecore parser that reads the `.ecore` file according to the given file path, transforms it using the `XMLParser` and starts the parsing mechanism.

```
export async function parseEcoreDefinitionFile(_path: any): Promise<any> {
  const parser = new XMLParser({
    ignoreAttributes: false,
    attributeNamePrefix: "",
  });

  let definitionFileContent = fs.readFileSync(_path, "utf8");
  const parsedXml = parser.parse(definitionFileContent);
  const ecoreDefinition = parseEcoreDefinition(parsedXml);
}
```

Listing 6.14: Entry function for the Ecore parser

The `parseEcoreDefinition` file analyzes each property of the given XML structure, and if the key of a property is `ecore:EPackage`, gathers the information from this property and visits its child properties, as can be seen in Listing 6.15.

```
export function parseEcoreDefinition(parsedXml: any): EcoreDefinition {
  const definition: EcoreDefinition = {
    classes: [],
    types: [],
    dataTypes: [],
  };
  Object.keys(parsedXml).forEach((key) => {
    switch (key) {
      case "ecore:EPackage":
        visitPackage(parsedXml[key], definition);
        break;
```

```
    }
  });
  return definition;
}
```

Listing 6.15: Function that parses the provided XML structure into an `EcoreDefintion` structure

In the `visitEPackage` function, again, all properties of the structure are analyzed; if a property is an `eClassifier`, the information of this property is gathered, and according to the `xsi:type` property of the classifier, the respective `visit`-function is called, which is either `visitEClass` or `visitEEnum`. In Listing 6.16, the implementation of the `visitEClass` is illustrated. As can be seen, for each child element of the `EClass`, it is checked whether the child is of type `eStructuralFeatures`. If so, the child element is further examined using the `visitStructuralFeatures` method (which can be seen in Listing 6.17). This function checks if the structural feature is of type `EAttribute` or `EReference` and creates an attribute according to its specifications.

```
function visitEClass(node: any, eClass: EcoreClass) {
  Object.keys(node).forEach((key) => {
    switch (key) {
      case "eStructuralFeatures":
        if (Array.isArray(node[key])) {
          node[key].forEach((feature: any) => {
            visitStructuralFeatures(feature, eClass);
          });
        } else {
          visitStructuralFeatures(node[key], eClass);
        }
        break;
    }
  });
}
```

Listing 6.16: Function that parses elements of type `EClass`

```
function visitStructuralFeatures(node: any, eClass: EcoreClass) {
  switch (node["xsi:type"]) {
    case "ecore:EAttribute":
    case "ecore:EReference":
     eClass.attributes.push({
       changeable: node["changeable"],
       containment: node["xsi:type"] === "ecore:EAttribute" ? true : node["
   containment"] === "true",
       defaultValueLiteral: node["defaultValueLiteral"],
       derived: node["derived"],
       ID: node["iD"],
       lowerBound: node["lowerBound"],
       multiplicity:getMultiplicity(node["lowerBound"],node["upperBound"]),
       name: cleanName(node["name"]),
       ordered: node["ordered"],
       reference: node["xsi:type"] === "ecore:EAttribute" ? false
```

```
            : !node["containment"] || node["containment"] === "false",
        transient: node["transient"],
        type: mapType(node["eType"]),
        unique: node["unique"],
        unsettable: node["unsettable"],
        upperBound: node["upperBound"],
        volatile: node["volatile"],
      });
  }
}
```

Listing 6.17: Function that parses elements of type `EAttribute` and `EReference`

The parsing of `EEnum` elements can be seen in Listing 6.18, while the parsing of `EDataTypes` is not handled in its own function, as the `EDataType` information is gathered in a simple string array.

```
function visitEEnum(node: any, eType: EcoreType) {
  Object.keys(node).forEach((key) => {
    switch (key) {
      case "eLiterals":
        node[key].forEach((literal: any) => {
          eType.types.push(
            JSON.stringify(literal["literal"] ?? literal["name"])
          );
        });
        break;
    }
  });
}
```

Listing 6.18: Function that parses elements of type `EEnum`

During the parsing, multiple utility functions, which map Ecore types to the types that represent them in a TypeScript definition (e.g., `EInteger` elements are mapped to `number`), or clean the name of a data type (e.g., if a data type in the Ecore definition uses a name, that is a reserved keyword in TypeScript).

Furthermore, after all elements in the Ecore definition have been parsed, a root class definition must be created that defines which classes can be created at the root level of the model. This is done by the function seen in Listing 6.19.

```
function collectRootClasses(definition: EcoreDefinition) {
  return definition.classes
    .filter((eClass) => {
      return !definition.classes.some((c) =>
        c.attributes.some(
          (attribute) => attribute.type === eClass.name && attribute.
    containment
        )
      );
    })
    .filter((eClass) => {
```

```
      return (
        eClass.extends.length === 0 ||
        !definition.classes.some((c) =>
          c.attributes.some((attribute) =>
            collectSuperClasses(definition, eClass)
              .map((sc) => sc.name)
              .includes(attribute.type)
          )
        )
      );
    })
    .filter((eClass) => !eClass.isAbstract)
    .map((eClass) => eClass.name);
}
```

Listing 6.19: Function that collects the root-level classes of the model

After the root elements have been collected, the parsing of the Ecore definition is finished.

General parsing The general parser is responsible for parsing documents into the data structure presented in Section 5.3. The parser module consists of three entry functions used to parse JSON-based configuration files, TypeScript-based configuration files, or TypeScript files that include a TypeScript-based grammar language definition.

The parser for the JSON configuration files is very simple; it just needs to read a file from the file system and return its value.

The parsers for the TypeScript files are more advanced and need to be looked at in more detail. In Listing 6.20, the entry into the definition file parser can be seen. At startup, the function uses `prettier`[2], to format the file according to the provided configurations. This has to be done, as the parser expects strings to have double quotes. This is particularly important for parsing constant elements representing a fixed element value. Next, using `typescript` (`ts`), a program is created, and a function to visit each child in the file is called.

```
export async function parseDefinitionFile(
  path: any
): Promise<Array<Declaration>> {
  let definitionFileContent = fs.readFileSync(path, "utf8");
  let formattedDefinitionFile = await prettier.format(definitionFileContent
    , {
    parser: "typescript",
    trailingComma: "es5",
  });
  fs.writeFileSync(path, formattedDefinitionFile);
  const program = ts.createProgram([path], {
    target: ts.ScriptTarget.ES2022,
  });
```

[2] https://prettier.io/docs/en/install.html

```
  const checker = program.getTypeChecker();
  const source = program.getSourceFile(path);
  const declarations: Array<Declaration> = [];
  ts.forEachChild(source, visit(declarations));
  return declarations;
}
```

Listing 6.20: Entry function to the parsing of definition files

In the `visit` function, the parser looks at every child element of the TypeScript file and checks for their types. Additionally, a `Declaration` is created according to the respective type, as shown in Listing 6.21.

```
export const visit = (target: Array<Declaration>) => (node: ts.Node) => {
  const declaration: Declaration = {
    type:
      ts.isClassDeclaration(node) || ts.isInterfaceDeclaration(node)
        ? "class"
        : "type",
    isAbstract: ts.isTypeAliasDeclaration(node),
    decorators: [],
    properties: [],
    extends: [],
  };
  if (ts.isClassDeclaration(node)) {
    target.push(declaration);
    ts.forEachChild(node, visitClassDeclaration(declaration));
  } else if (ts.isTypeAliasDeclaration(node)) {
    target.push(declaration);
    ts.forEachChild(node, visitTypeDeclaration(declaration));
  } else if (ts.isInterfaceDeclaration(node)) {
    target.push(declaration);
    ts.forEachChild(node, visitInterfaceDeclaration(declaration));
  }
};
```

Listing 6.21: Functionality to create `Declaration` according to node kind

Following the declaration's creation, the node's child elements are visited using different parser functions according to their type. Though classes and interfaces are mapped into declarations that represent model elements, the parser handles the child nodes differently. The reason for this distinction is that decorators are not supported for interfaces in TypeScript.

In Listing 6.22, the parser implementation that visits the child nodes of an interface declaration can be seen. It is noteworthy that nodes of kind `Identifier` and `PropertyDeclaration` are handled the same way in the parser that visits class declarations. The `Identifier` is used to set the model element's name, while the `PropertyDeclaration` is used to parse properties correctly. As decorators and some keywords are not supported by interfaces, metainformation, such as whether a model element is abstract or a root element, has to be obtained using a different approach.

Therefore, this information is gathered using heritage clauses. If an interface extends the interface `ABSTRACT_ELEMENT`, the `isAbstract` property of the declaration is set to true, and if it extends the interface `ROOT_ELEMENT`, the `root` decorator is added to the `decorators` array of the declaration.

```
export const visitInterfaceDeclaration =
  (target: Declaration) => (node: ts.Node) => {
    if (ts.isIdentifier(node)) {
      target.name = node.text;
    } else if (ts.isPropertySignature(node)) {
      const property: Property = {
        decorators: [],
        isOptional: false,
        types: [],
        multiplicity: Multiplicity.ONE_TO_ONE,
      } as Property;
      target.properties.push(property);
      ts.forEachChild(node, visitPropertyDeclaration(property));
    } else if (ts.isHeritageClause(node)) {
      ts.forEachChild(node, (child) => {
        if (ts.isExpressionWithTypeArguments(child)) {
          ts.forEachChild(child, (child) => {
            if (ts.isIdentifier(child)) {
              if (child.text === "ABSTRACT_ELEMENT") {
                target.isAbstract = true;
              } else if (child.text === "ROOT_ELEMENT") {
                target.decorators.push("root");
              } else {
                target.extends.push(child.getText());
              }
            }
          });
        }
      });
    }
  };
```

Listing 6.22: Functionality to parse an interface declaration

In contrast to the interface declaration, a class declaration would ignore the heritage clauses `ABSTRACT_ELEMENT` and `ROOT_ELEMENT`. However, for a class declaration, it is possible to check for the `abstract` keyword to set the `isAbstract` property, and it is possible to iterate over all decorators that a class can have to collect them in the `decorators` array of the declaration.

Another kind of root-level parser is the type alias parser, whose implementation can be seen in Listing 6.23.

```
export const visitTypeDeclaration =
  (target: Declaration) => (node: ts.Node) => {
    const property = {
      decorators: [],
      isOptional: true,
```

```
      types: [],
      multiplicity: Multiplicity.ONE_TO_ONE,
    } as Property;
    if (ts.isIdentifier(node)) {
      target.name = node.text;
    } else if (ts.isUnionTypeNode(node)) {
      target.properties.push(property);
      ts.forEachChild(node, visitUnionType(property));
    } else if (ts.isTypeReferenceNode(node)) {
      target.properties.push(property);
      ts.forEachChild(node, visitTypeReferenceNode(property));
    } else if (ts.isLiteralTypeNode(node)) {
      target.properties.push(property);
      ts.forEachChild(node, (child) => {
        if (
          ts.isStringLiteral(child) ||
          ts.isNumericLiteral(child) ||
          child.kind === ts.SyntaxKind.TrueKeyword ||
          child.kind === ts.SyntaxKind.FalseKeyword
        ) {
          property.types.push({ type: "constant", typeName: child.getText()
    });
        }
      });
    }
  };
```

Listing 6.23: Functionality to parse a type declaration

For the child nodes of `PropertyDeclarations`, seven different types of nodes need to be handled:

- `Identifier`: Used to set the name of the property.
- `Decorator`: Used to check for the `@crossReference` decorator.
- `TypeReference`: This represents a node of a referenced type. The node can be a container type or another class, interface, or type declaration in the definition file. `TypeReferences` are handled separately in their own parser function.
- `UnionType`: It is possible for the type of an element to have multiple different valid types. For this, it is possible to define a union type. The parsing of a union type is handled separately in the `visitUnionType` function.
- `NumberKeyword` or `BooleanKeyword` or `StringKeyword`: These kinds represent the simple types `number`, `boolean` and `string`. If the type of a property is one of them, the type in the declaration is defined as `simple`.
- `LiteralTypeNode`: This element represents nodes with a constant value. The type of the property declaration is set to `constant` if this kind is encountered.

- `QuestionToken`: Used to represent elements that are optional. Hence, if this element is found, the property declaration is set to optional.

In Listing 6.24 and Listing 6.25, the implementations to parse `TypeReferenceNodes` and `UnionTypes` can be seen. In the `visitTypeReferenceNode` function, the special handling for the definition of cross-references within interface declarations is implemented. Cross-references are typically defined using the `@crossReference` decorator inside a class declaration. However, as has already been discussed, in interface declarations, decorators are not supported; therefore, to mark a property as a cross-reference within an interface declaration, the type of the property has to be set to `CrossReference<T>`, where `T` stands for the type to which a cross-reference should be created.

```
export const visitTypeReferenceNode = (target: Property) => (node: ts.Node)
    => {
  if (ts.isIdentifier(node)) {
    if (node.text === "Array") {
      target.multiplicity = target.isOptional
        ? Multiplicity.ZERO_TO_N
        : Multiplicity.ONE_TO_N;
    } else if (node.text === "CrossReference") {
      target.decorators.push("crossReference");
    } else {
      target.types.push({ type: "complex", typeName: node.text });
    }
  } else if (ts.isTypeReferenceNode(node)) {
    ts.forEachChild(node, visitTypeReferenceNode(target));
  }
};
```

Listing 6.24: Functionality to parse a type reference declaration

```
export const visitUnionType = (target: Property) => (node: ts.Node) => {
  if (
    node.kind === SyntaxKind.NumberKeyword ||
    node.kind === SyntaxKind.BooleanKeyword ||
    node.kind === SyntaxKind.StringKeyword
  ) {
    target.types.push({ type: "simple", typeName: node.getText() });
  } else if (ts.isTypeReferenceNode(node)) {
    ts.forEachChild(node, visitTypeReferenceNode(target));
  } else if (ts.isLiteralTypeNode(node)) {
    ts.forEachChild(node, (child) => {
      if (
        ts.isStringLiteral(child) ||
        ts.isNumericLiteral(child) ||
        child.kind === ts.SyntaxKind.TrueKeyword ||
        child.kind === ts.SyntaxKind.FalseKeyword
      ) {
        let text = child.getText().replace(/\'/g, ""); //'
        target.types.push({ type: "constant", typeName: JSON.stringify(text
    ) });
      }
```

```
    });
  }
};
```

Listing 6.25: Parsing of a union type declaration

6.2.2 Validation & Transformation

After parsing the definition and configuration files, the generator transforms and validates the result. In this section, the implementation of the transformation steps will be shown first, and then the validation will be explained in more detail.

Transformation of `Declaration` to `LangiumDeclaration` In this step, the data structure returned from the parsing step is transformed into the `LangiumDeclaration` data structure. The transformation is done in the `transformDeclaration` method, which consists of three steps, as described in Section 5.3.

In the first step, the `Declarations` are mapped to `LangiumDeclarations` as can be seen in Listing 6.26. After that, the `extendedBy` property is filled correctly by checking whether each declaration is included in the extends array of another declaration as can be seen in Listing 6.27. Finally, for every abstract declaration, the properties are removed, as implemented in Listing 6.28

```
const langiumDeclarations: Array<LangiumDeclaration> = declarations.map(
    (declaration) => {
      if (declaration?.extends.length > 0) {
        declaration.extends.forEach((extend) => {
          declaration.properties = declaration.properties.concat(
            declarations.find((d) => d.name === extend)?.properties || []
          );
        });
      }
      return {
        type: declaration.type,
        name: declaration.name,
        isAbstract: declaration.isAbstract,
        decorators: declaration.decorators,
        properties: declaration.properties,
        extendedBy: [],
      };
    }
  );
```

Listing 6.26: Map `Declaration` to `LangiumDeclaration`

```
declarations.forEach((declaration) => {
    if (declaration.extends?.length > 0) {
      declaration.extends.forEach((extend) => {
        langiumDeclarations
```

```
            .find((langiumDeclaration) => langiumDeclaration.name === extend)
            ?.extendedBy.push(declaration.name);
        });
    }
  });
```

Listing 6.27: Fill the `extendedBy` property of the `LangiumDeclarations`

```
langiumDeclarations.forEach((langiumDeclaration) => {
    if (langiumDeclaration.isAbstract &&
        langiumDeclaration.type === "class") {
      langiumDeclaration.properties = [];
    }
  });
```

Listing 6.28: Remove all properties from abstract `LangiumDeclarations`

Transform `LangiumDeclarations` to `LangiumGrammar` This step is used to prepare `LangiumGrammar` data structure, which can easily be transformed into the actual Langium grammar. Again, three steps have to be implemented, as described in Section 5.3. First, the `EntryRule` is created. This is done by searching all `LangiumDeclarations` for the declaration that includes the `@root` decorator in its `decorators` property. The implementation of the mapping to the `EntryRule` can be seen in Listing 6.29.

```
function transformLangiumDeclarationToEntryRule(
  langiumDeclaration: LangiumDeclaration
): EntryRule {
  return {
    name: langiumDeclaration.name,
    definitions: langiumDeclaration.properties.map((property) => ({
      name: property.name,
      type: property.types,
      multiplicity: property.multiplicity,
      crossReference: property.decorators.includes("crossReference"),
      optional: property.isOptional,
    })),
  };
}
```

Listing 6.29: Mapping of `LangiumDeclaration` to `EntryRule`

Next, the `TypeRules` are created by collecting `LangiumDeclarations`, which represent an unassigned rule call (see Section 2.4).

First, all `LangiumDeclarations` whose type is `type` and those which are abstract and whose type is `class` are mapped into the `TypeRule` format. The implementation of this mapping can be seen in Listing 6.30.

```
const typeRules: Array<TypeRule> = langiumDeclarations
    .filter((declaration) => declaration.type === "type")
    .map((declaration) => ({
      name: declaration.name,
```

```
      definitions:
        declaration.properties.map((property) => property.types)?.flat() ??
    [],
  }))
  .concat(
    langiumDeclarations
      .filter(
        (declaration) =>
          declaration.isAbstract && declaration.type === "class"
      )
      .map((declaration) => ({
        name: declaration.name,
        definitions: declaration.extendedBy.map((extendedBy) => ({
          typeName: extendedBy,
          type: "simple",
        })),
      }))
  );
```

Listing 6.30: Transformation of `LangiumDeclarations` to `TypeRules`

Using these initial `TypeRules`, the remaining `LangiumDeclarations` are iterated, and their properties and associated types are checked. If a type represents a union type, it is checked if a `TypeRule` that represents this union type already exists, and if so, the union type is replaced with the name of the found `TypeRule`. However, if no such `TypeRule` can be found, a new `TypeRule` is created, and the type of the property is replaced with the name of the newly created `TypeRule`. After this, the creation of the `TypeRules` is complete.

Finally, the mapping of non-abstract `LangiumDeclarations` whose type is `class` needs to be performed. This step includes the mapping and adds a new property for declarations that do not have the referenceProperty, defined in the generator config file. Typically, the name of this property is `__id`.

Validation of `LangiumDeclarations` This validation step implements the initial validation, which checks for obvious errors in the declaration. This includes checking for the existence and uniqueness of the root element and if the root element defines any properties. Furthermore, it is checked for abstract elements and whether elements extending these abstract definitions exist. The implementation of these checks is very straightforward and therefore not explained in more detail within this section.

Validation of `LangiumGrammar` The validation of the `LangiumGrammar` is twofold:

- **Check for unused elements**: This check is a soft check, which means that if there do exist definitions for elements that are not used in any context in the overall definition, a warning is shown. The implementation of this check uses the `typescript-graph`[3] library to build a graph that consists of all rule elements

[3] `https://segfaultx64.github.io/typescript-graph/`

and the connections between them. After the graph has been constructed, the `indegree`-value is checked for every node, which is a property that counts the number of incoming edges. If the number of incoming edges is zero and the graph node does not represent the root element, it is clear that no other definition uses the definition. In Listing 6.31, the implementation for this check can be seen.

```
graph.getNodes().forEach((node) => {
    if (
      !["string", "number", "boolean"].includes(node.typeArgument) &&
      graph.indegreeOfNode(node.typeArgument) === 0 &&
      node.typeArgument !== langiumGrammar.entryRule.name
    ) {
      console.log(
        chalk.yellow(
          `WARNING: Type ${node.typeArgument} has been defined but is never
    used.`
        )
      );
    }
  });
```

Listing 6.31: Implementation of the check for unused elements

- **Check if grammar definition is serializable**: For this check, first the initial serializable elements are defined, which are the simple types `string`, `number` and `boolean`. After they have been defined, a while loop is executed, which searches for serializable elements in each iteration. If, in an iteration, a new serializable element is found, it is added to the serializable set. The iteration is stopped if the serializable set consists of the same elements at the start and end of the while loop. After the loop is exited, it is checked whether the serializable set contains all rules. If not, the grammar can not be serialized, and a validation error is shown. If this validation fails, the generator is unable to create the Langium grammar. In Listing 6.32 an example code snippet can be seen, which executes a check in the iteration for elements of type `ParserRule`. As can be seen, for the definitions, it is checked if the type of a definition is already included in the serializable set, or the type of the definition is `constant`, or the definition is optional (which can be checked either by the optional flag or the multiplicity). Additionally, for abstract rule elements, it is checked if all elements that extend the rule element are in the serializable set, and if so, the `ParserRule` becomes serializable.

```
lanigiumGrammar.parserRules.forEach((parserRule) => {
      if (
        parserRule.definitions &&
        parserRule.definitions.length > 0 &&
        parserRule.definitions.every(
          (definition) =>
            serializableRuleElements.has(definition.type[0].typeName) ||
```

```
            definition.type[0].type === "constant" ||
            definition.optional ||
            definition.multiplicity === Multiplicity.ZERO_TO_N
        )
      ) {
        serializableRuleElements.add(parserRule.name);
      } else if (parserRule.isAbstract && parserRule.extendedBy) {
        if (
          parserRule.extendedBy.every((extendedBy) =>
            serializableRuleElements.has(extendedBy)
          )
        ) {
          serializableRuleElements.add(parserRule.name);
        }
      }
    });
```

Listing 6.32: Check for a `ParserRule`, to validate whether it is serializable

6.2.3 File Creation

The file creation process starts after the `LangiumGrammar` has been validated successfully. When first called, the generator creates various files based on templates, which do not need to be regenerated. These files include:

- **Configuration files**: The configuration files that are created include the `package.json` file, which defines which packages need to be installed for the newly created project but also defines what commands do exist.
- **Visual Studio Code Extension code**: As the generator builds the project initially, it would be good to test if the setup has worked correctly and the language is built as expected. Therefore, a VSCode extension is also created next to the initial project, which can be started using the extensions debug window in VSCode. This extension opens a workspace, in which - on the creation of a file with the file extension of the created language - the language client starts editing the model in a textual format and starts the model server API in the background.
- **Custom implementations of Langium services**: Some services are required to ease the usage of the model server API easier. This includes the `JsonSerializer` service, which is used within the `PatchManager`, but also a custom validator that is used to validate that in the entire workspace, the `referenceProperty` is unique.
- **Langium module**: In the Langium module, all custom services, as well as the extension to existing services, can be defined so the language server can use them.

Two files are recreated in every generation process: The `.langium` definition and the `Serializer`. Optionally, at the beginning of the generation, the TypeScript-based

grammar language definition can be created by parsing a `.ecore` definition and mapping the parsed structure to the TypeScript grammar language.

TypeScript-based grammar definition Using the data structure created by parsing an `.ecore` file (see Section 5.3), the TypeScript-based grammar language definition for the parsed definition is created according to the concept presented in Section 5.2. In Section 8.2 the implementation of the mapping can be looked up.

`.langium` grammar definition To create a `.langium` grammar definition, which is a generic JSON grammar, the `EntryRule`, `TypeRules`, and `ParserRules` have to be mapped according to the rules described in Section 5.3. In Listing 6.33, the implementation of the mapping of `TypeRules` can be seen, which is the most straightforward mapping, as in the case of `TypeRules`, unassigned rule calls are used, which assign the parsing of an element to the subelements.

```
function typeRuleToLangiumText(typeRules: Array<TypeRule>) {
  return typeRules
   .map((typeRule) => {
     const returnType = getReturnTypeFromDefinitions(typeRule.definitions);
     return `${typeRule.name}${
       returnType ? " returns " + returnType : ""
     }: ${typeRule.definitions
       .map((element) =>
         element.type === "constant"
           ? getLangiumType(JSON.parse(element.typeName))
           : getLangiumType(element.typeName)
       )
       .join(" | ")};`;
   })
   .join("\n");
}
```

Listing 6.33: Map `TypeRule` to Langium definition

This function uses two utility functions: `getReturnTypeFromDefinitions` and `getLangiumType`. If an unassigned rule call is a data type rule, it parses an element of a simple type (like string or number), and the type that the rule should be parsed to can be defined. For this, the `getReturnTypeFromDefinitions` function checks if all definitions of a rule are of type `simple`, and then checks if all definitions have the same simple type. If this is the case, the found simple type is used as the return type; otherwise, the return type defaults to `string`. If all definitions are of type `constant`, it is checked if the type of the constant value is of the same data type, and if so, this data type is used.

Langium uses terminal rules to parse strings, numbers, and booleans; therefore, in case the type of a definition is one of these types, they must be mapped to terminal rules. The generator creates some initial terminal rule definitions representing these types.

The `EntryRule` and the `ParserRules` are mapped using the same functionality, which makes some adjustments according to the rule type as described in the following:

- **Rule Declaration**: The rule declaration differs from others so that for `EntryRules`, the keyword `entry` is added before the declaration, as can be seen in Listing 6.34.

```
let text = `${entry ? "entry " : ""}` + `${rule.name}`;
  text += ": ";
```

Listing 6.34: Create Declaration for `EntryRule` and `ParserRules`

- **Alternatives**: As it is possible to extend classes that are not abstract, `ParserRules`, which are extended by other rules, have to define alternatives. Listing 6.35 shows the implementation of the alternatives. Additionally, within the code the type-cast to `any` is needed, as for the `EntryRule`, the `extendedBy` property does not exist.

```
if ((rule as any).extendedBy && (rule as any).extendedBy.length > 0) {
    text += (rule as any).extendedBy.join(" | ");
    text += " | ";
  }
```

Listing 6.35: Create alternatives for non-abstract `ParserRules`

- `Rule Body`: In the previous two code examples, the declaration and the alternatives for a rule have been defined, but not the `Rule Body`. For alternatives, their own rule declaration defines the `Rule Body`. The body consists of the actual JSON grammar definition, as the structure of a rule is defined here. Listing 6.36 shows the implementation of the `Rule Body`. As can be seen, an opening curly brace is added at the beginning of the code, and at the end, a closing one is added. In between, a property with the name `__type` is added, which defines the type of the rule inside the JSON structure. Furthermore, for each definition in the rule, the `getProperty` function is called, which maps the definition to a JSON property. To check if a definition is optional in the JSON structure, the question mark operator is added at the end of the definition. This signals to Langium that this property is optional.

  The mapping of the properties inside the `getProperty` function checks for each definition the multiplicity, as well as if the definition is used to define a cross-reference to another element inside the grammar.

```
text += " '{' ";
  text += entry ? "" : `'"__type"' ':' '"${rule.name}"' `;
  text += ` ${rule.definitions
    .map(
```

```
      (property, index) =>
        `( ${entry && index === 0 ? "" : "','"}  ${getProperty(
          property,
          rules
        )} )` +
        (property.optional || property.multiplicity === Multiplicity.
  ZERO_TO_N
          ? "?"
          : "")
    )
    .join(" ")} `;
  text += " '}' ;";
```

Listing 6.36: Create body for rule definition

In Listing 6.37, an example `LangiumGrammar` can be seen, and in Listing 6.38, the resulting Langium grammar definition is illustrated. Listing 6.39 illustrates how an example model instance of this definition can look like.

```
{
  "entryRule": {
    "name": "Model",
    "definitions": [
      {"name": "nodes", "multiplicity": "*", "crossReference": false,
          ↪ "optional": true, "type": {"type": "complex", "typeName":
          ↪ "Node"}},
      {"name": "refs", "multiplicity": "*", "crossReference": false,
          ↪ "optional": true, "type": {"type": "complex", "typeName":
          ↪ "RefNode"}}
    ]
  },
  "typeRules": [
    {"name": "Node", "definitions": [{"type": "complex", "SubNode"}]}
  ],
  "parserRules": [
    {
      "name": "SubNode",
      "isAbstract": false,
      "extendedBy": [],
      "definitions": [
        {"name": "__id", "multiplicity": "1", "crossReference": false,
            ↪ "optional": false, "type": {"type": "simple", "typeName":
            ↪ "string"}},
      ]
    },
    {
      "name": "RefNode",
      "isAbstract": false,
      "extendedBy": [],
      "definitions": [
        {"name": "ref", "multiplicity": "1", "crossReference": true,
            ↪ "optional": true, "type": {"type": "complex", "typeName":
            ↪ "SubNode"}},
      ]
```

```
    },

  ]
}
```

Listing 6.37: Example `LangiumGrammar`

```
entry Model: '{' '"nodes"': '[' (nodes+=Node) (',' nodes+=Node)* ']' ',' '"
    refs"': '[' (refs+=RefNode) (',' refs+=RefNode)* ']' '}';
Node: SubNode;
SubNode: '{' '"__id"' ':' '"'__id=ID'"' '}';
RefNode: '{' '"ref"' ':' '{' '"__type"' ':' '"Reference"' ',' '"__refType
    "' ':' '"Node"' ',' '"__value"' ':' '"'ref=[Node:ID]'"' '}' '}';
```

Listing 6.38: Example Langium grammar definiton

```
{
  "nodes": [
    {"__id": "Id_SubNode1"},
    {"__id": "Id_SubNode2"}
  ],
  "refNodes": [
    {
      "ref": {
        "__type": "Reference",
        "__refType": "Node",
        "__value": "Id_SubNode1"
      }
    }
  ]
}
```

Listing 6.39: Example Model instance of the example grammar definiton

`Serializer` In the implementation of the serializer, a model instance has to be mapped to the Langium grammar definition in the discussed JSON format. This is done by creating multiple serializer functions, which are used to serialize the `EntryRule`, `ParserRules`, and `TypeRules`. In each serializer function, the serialization of each property of a type is done by either directly returning the value of the property, in case the type of the property is a `complex` or `simple` type; otherwise, the serialization of the property is done using its own serializer function.

If, for example, a `ParserRule` has two properties as can be seen in Listing 6.40, where one property is a simple type (string) and the other property is a complex type (another `ParserRule`), the generator would create a serializer function for this rule as can be seen in Listing 6.41.

```
{
  "name": "Node",
  "isAbstract": false,
  "extendedBy": [],
```

```
  "definitions": [
    {"name": "__id", "multiplicity": "1", "crossReference": false,
        ↪ "optional": false, "type": {"type": "simple", "typeName":
        ↪ "string"}},
    {"name": "otherProperty", "multiplicity": "1", "crossReference":
        ↪ false, "optional": false, "type": {"type": "complex",
        ↪ "typeName": "OtherProperty"}},
  ]
}
```

Listing 6.40: Example `ParserRule` with two properties

```
serializeNode(element: Node): string {
  let str: Array<string> = [];
  str.push('"__type": "Node"');
  if (element.__id !== undefined && element.__id !== null) {
    str.push('"__id": ' + '"' + element.__id + '"');
  }
  if (element.otherProperty !== undefined && element.otherProperty !==
      ↪ null) {
    str.push('"otherProperty": ' +
        ↪ this.serializeOtherProperty(element.otherProperty));
  }
  return "{" + str.join(",\n") + "}";
}
```

Listing 6.41: Example `ParserRule` with two properties

6.2.4 Initial Project Properties & Package Installation & Build

These three functionalities are implemented in the `index.ts` file of the generator. For the initial project properties, prompting is used to gather user inputs for the selected properties (language-id, language name, etc.). Following the input of the properties, the previously described functionalities are used to create the initial project structure (parsing, file creation). After this is done, the generator spawns three commands on the created project: `npm install` to install the needed `node_modules`, `npm run generate`, to create the Langium grammar from the initial definition file, and `npm run build` to build the initial VSCode extension, to be able to test the functionality of the newly created project.

6.3 Summary

This chapter provided some insight into the implementation of the model server API and the generator functionality considering the concepts that have been discussed in Chapter 5. For the model server API, its most important features and how it can be utilized as a language server have been discussed, while for the generator, the parsing features, the validation features, and the file creation features have been presented in some detail. The following chapter will discuss the evaluation of the implementation of the artifacts.

CHAPTER 7

Evaluation

This chapter presents the results of evaluating the implemented artifacts. The first step includes functional testing, which aims to verify whether the artifacts' implementations are correct. Within the second step, an informed argument will evaluate the differences between the Typescript-based grammar definition and the Ecore metamodel definition. Finally, the implementation of the whole system, including the generator, Typescript-based grammar definition, and the model server API, will be evaluated using two different scenarios.

7.1 Functional Testing

The evaluation step, which includes functional testing, is a verification step for the implementation rather than an evaluation of the scientific work in this thesis. However, with this verification step, it can be ensured that the implementation of selected software parts is correct. The testing has been done utilizing the functionalities of `vitest`[1].
The following parts of the software have been tested with specified unit tests:

- Parsing of `EClasses`
- Parsing of `EAttributes`
- Parsing of `EReferences`
- Parsing of `EEnums`
- Parsing of `EDataTypes`
- Transformation of `Declarations` to `LangiumDeclarations`

[1] `https://vitest.dev/`

- Transformation of `LangiumDeclarations` to `LangiumGrammar`
- Validation of the `generator-config.ts` file
- Validation of the `LangiumDeclarations`
- Validation of the `LangiumGrammar`
- Generation of definition in Langium grammar language from the TypeScript-based grammar definition
- Generation of the Langium serializer service from the TypeScript-based grammar definition

The testing of the parsing functionality of the TypeScript-based grammar language is included in the tests for the generation of the Langium grammar and Langium serializer service. Correct definition files and the resulting serializer and Langium grammar files have been created for these tests. The process of this test is the following:

1. Parse definition file using the parsing functionality of the generator
2. Transform the parsed `Declarations` into `LangiumDeclarations`
3. Transform the `LangiumDeclarations` into the `LangiumGrammar`
4. Validate that the grammar definition is correct
5. Generate the Langium grammar text from the `LangiumGrammar`
6. Check if the created Langium grammar equals the definition in the correct Langium grammar file
7. Generate the serializer text from the `LangiumGrammar`
8. Check if the created serializer text equals the serializer in the correct serializer file

After these software parts were successfully tested, it was ensured that the generator functionality was implemented correctly.

7.2 Descriptive Evaluation - Informed Argument

This evaluation step tests the TypeScript-based grammar language against the Ecore metamodel definition. For this, the generator transforms three Ecore definitions into the TypeScript-based grammar language. Then, the newly created language is tested in the initial VSCode extension to see if it conforms to the initial Ecore definition.

In Section 8.2 the `.ecore` definition and the resulting TypeScript-based grammar language, as well as an example model instance in the JSON notation, can be looked up for the three evaluation examples.

In the three examples, the following features of `EClasses`, `EAttributes`, `EReferences`, and `EEnums` are included:

1. **`EClass` features**:
   a) Definition of `EClass` element
   b) Definition of abstract `EClass` element
   c) Definition of `EClass` interface element
   d) Definition of abstract `EClass` interface element
   e) Definition of `EClass` element with one super type
   f) Definition of `EClass` element with multiple super types
2. **`EAttribute` features**:
   a) Definition of `EAttribute`
   b) Definition of derived `EAttribute`
   c) Definition of transient `EAttribute`
   d) Definition of ordered `EAttribute`
   e) Definition of unique `EAttribute`
   f) Definition of changeable `EAttribute`
   g) Definition of volatile `EAttribute`
   h) Definition of unsettable `EAttribute`
   i) Definition of ID `EAttribute`
   j) Definition of `EAttribute` with lower bound
   k) Definition of `EAttribute` with upper bound
   l) Definition of `EAttribute` with built-in type
   m) Definition of `EAttribute` with custom type
3. **`EReference` features**:
   a) Definition of `EReference`
   b) Definition of `EReference` with lower bound
   c) Definition of `EReference` with upper bound
   d) Definition of `EReference` with containment
   e) Definition of `EReference` without containment

4. **`EEnum` features**:
    a) Definition of `EEnum`
    b) Definition of `EEnumLiteral` for `EEnum`

In comparison to the Ecore `EClass` definitions, in the TypeScript-based grammar language, all functionalities except the multiple super types could be represented. However, it is possible to find a workaround for that by defining the element that should have two super types as an interface. With this, it is possible to support the definition with multiple `extends` clauses in the TypeScript-native representation.

Some features that the Ecore `EAttribute` supports could not be included in the TypeScript-based grammar language. This is for two reasons: the functionality is not supported in a TypeScript-native way, and Langium does not support the functionality.

The TypeScript-based grammar language does not support the following features:

- Definition of `derived` property
- Definition of `transient` property
- Definition of `ordered` property
- Definition of `unique` property
- Definition of `changeable` property
- Definition of `volatile` property
- Definition of `unsettable` property

As can be seen, the list of functionalities not directly supported by the TypeScript-based grammar language includes many features. However, most of these features can be added by extending the TypeScript-based grammar language and creating custom services for each needed functionality. For example, the `derived` and `volatile` features could be created by adding two new decorators to the TypeScript-based grammar language (`@derived`, `@volatile`). Then, the generator could be extended to create custom services for Langium, which set the value of a derived or volatile attribute whenever a document changes.

Next, the definition of ID attributes is not supported, as in the generator, per default an `__id` attribute is created, which is used to identify each element in the document.

The lower-bound and upper-bound features of attributes are supported, though there are limitations. In Ecore, it is possible to define an arbitrary value as lower and upper bound, whereas, in the TypeScript-based definition, it is only possible to define a lower bound of zero or one and an upper bound of one or more. To enable arbitrary values for the upper

bound and lower bound of attributes, the grammar language could be extended by either a custom type or two decorators `@lowerBound` and `@upperBound`. After these have been added, the generator implementation needs to be adjusted so that it is possible to parse the values of lower bound and upper bound and then create the Langium grammar with these bounds.

Finally, setting the type of an attribute is a very important feature, which is supported in the TypeScript-based grammar definition.

For `EReferences` the limitations of the lower and upper bound are the same ones as for `EAttributes`. However, it is possible to define attributes as containment attributes and also as non-containment attributes (which signalizes cross-references) without any limitations.

The functionality of enums is natively supported in TypeScript, though, in the TypeScript-based grammar language, enums are represented as union types.

Table 7.1 compares the features supported in Ecore, the TypeScript-based grammar language, and the TypeScript-based grammar language combined with Langium custom services.

7.3 Descriptive Evaluation - Scenarios

In this evaluation step, existing graphical modeling tools will be rebuilt and tested for selected criterias. This evaluation not only tests whether the implementation using the Langium-based model management reflects the initial implementation but also checks whether all artifacts created in the course of this thesis work correctly together. In the first two subsections, the implementation of the scenarios will be discussed, and following that, in the final subsection, the evaluation of the results of these scenarios will be discussed.

7.3.1 Workflow diagram - scenario implementation

In this scenario, the workflow diagram model editor, which is the default example of the Graphical Language Server Platform (GLSP), needs to be rebuilt.

For this rebuild, an implementation of the Langium model management and a GLSP server and client are required.

Langium model management The workflow diagram is a rather small diagram consisting of only a few different types of nodes and one type of edge. The definition of the workflow diagram language, using the TypeScript-based grammar language, can be seen in Listing 7.1.

Ecore feature	Supported
Definition of `EClass` element	✓
Definition of abstract `EClass` element	✓
Definition of `EClass` interface element	✓
Definition of abstract `EClass` interface element	✓
Definition of `EClass` element with one super type	✓
Definition of `EClass` element with multiple super types	(✓)
Definition of `EAttribute`	✓
Definition of derived `EAttribute`	✗
Definition of transient `EAttribute`	✗
Definition of ordered `EAttribute`	✗
Definition of unique `EAttribute`	✗
Definition of changeable `EAttribute`	✗
Definition of volatile `EAttribute`	✗
Definition of unsettable `EAttribute`	✗
Definition of ID `EAttribute`	✗
Definition of `EAttribute` with lower bound	(✓)
Definition of `EAttribute` with upper bound	(✓)
Definition of `EAttribute` with built-in type	✓
Definition of `EAttribute` with custom type	✓
Definition of `EReference`	✓
Definition of `EReference` with lower bound	(✓)
Definition of `EReference` with upper bound	(✓)
Definition of `EReference` with containment	✓
Definition of `EReference` without containment	✓
Definition of `EEnum`	✓
Definition of `EEnumLiteral` for `EEnum`	✓

Table 7.1: Comparison of features supported by Ecore and the TypeScript-based grammar language [✓=supported, (✓)=supported with limitation, ✗=not supported]

```
type NodeType = "decision" | "fork" | "join" | "merge";
type TaskType = "manual" | "automated";
type Weight = "low" | "medium" | "high";
@root
class Model {
  nodes?: Array<Node>;
  edges?: Array<Edge>;
  metaInfos?: Array<MetaInfo>;
}
abstract class Node {
  name: string;
}
```

```
class TaskNode extends Node {
  label?: string;
  duration?: number;
  taskType?: TaskType;
  reference?: string;
}
class Category extends Node {
  children?: Model;
  label?: string;
}
class ActivityNode extends Node {
  nodeType?: NodeType;
}
class Edge {
  @crossReference source: Node;
  @crossReference target: Node;
}
class WeightedEdge extends Edge {
  weight: Weight;
}
abstract class MetaInfo {
  @crossReference node: Node;
}
class Size extends MetaInfo {
  height: number;
  width: number;
}
class Position extends MetaInfo {
  x: number;
  y: number;
}
```

Listing 7.1: Definition of the workflow diagram language using the TypeScript-based grammar language

Using this definition of the workflow diagram language, the generator creates a Langium grammar, which can be seen in Listing 7.2. By comparing the readability of Listing 7.1 and Listing 7.2, one can see the advantage of using the TypeScript-based grammar language, which is then used to automatically generate the not so easily readable Langium grammar.

```
grammar WorkflowDiagram
import 'terminals'
entry Model: '{' ( '"nodes"' ':' '[' ( (nodes+=Node) ( ',' nodes+=Node)
    * )? ']' )? ( ',' '"edges"' ':' '[' ( (edges+=Edge) ( ',' edges+=Edge)
    * )? ']' )? ( ',' '"metaInfos"' ':' '[' ( (metaInfos+=MetaInfo) ( ','
    metaInfos+=MetaInfo)* )? ']' )? '}' ;
Node: TaskNode | Category | ActivityNode;
TaskNode: '{' '"__type"' ':' '"TaskNode"' ( ',' '"__id"' ':' '"'__id=ID
    '"' ) ( ',' '"label"' ':' '"'label=ID'"' )? ( ',' '"duration"' ':'
    duration=INT )? ( ',' '"taskType"' ':' '"'taskType=TaskType'"' )? (
    ',' '"reference"' ':' '"'reference=ID'"' )? ( ',' '"name"' ':' '"'
    name=ID'"' ) '}' ;
```

```
Category: '{' '"__type"' ':' '"Category"' ( ',' '"__id"' ':' '"'__id=ID
    '"' ) ( ',' '"children"' ':' children=Model )? ( ',' '"label"' ':'
    '"'label=ID'"' )? ( ',' '"name"' ':' '"'name=ID'"' ) '}' ;
ActivityNode: '{' '"__type"' ':' '"ActivityNode"' ( ',' '"__id"' ':' '"'
    __id=ID'"' ) ( ',' '"nodeType"' ':' '"'nodeType=NodeType'"' )? ( ','
    '"name"' ':' '"'name=ID'"' ) '}' ;
Edge: WeightedEdge | '{' '"__type"' ':' '"Edge"' ( ',' '"__id"' ':' '"'
    __id=ID'"' ) ( ',' '"source"' ':' '{' '"__type"' ':' '"Reference"'
    ',' '"__refType"' ':' '"Node"' ',' '"__value"' ':' '"'source=[Node:
    ID]'"' '}' ) ( ',' '"target"' ':' '{' '"__type"' ':' '"Reference"'
    ',' '"__refType"' ':' '"Node"' ',' '"__value"' ':' '"'target=[Node:
    ID]'"' '}' ) '}' ;
WeightedEdge: '{' '"__type"' ':' '"WeightedEdge"' ( ',' '"__id"' ':' '"'
    __id=ID'"' ) ( ',' '"weight"' ':' '"'weight=Weight'"' ) ( ',' '"
    source"' ':' '{' '"__type"' ':' '"Reference"' ',' '"__refType"' ':'
    '"Node"' ',' '"__value"' ':' '"'source=[Node:ID]'"' '}' ) ( ',' '"
    target"' ':' '{' '"__type"' ':' '"Reference"' ',' '"__refType"' ':'
    '"Node"' ',' '"__value"' ':' '"'target=[Node:ID]'"' '}' ) '}' ;
MetaInfo: Size | Position;
Size: '{' '"__type"' ':' '"Size"' ( ',' '"__id"' ':' '"'__id=ID'"' ) (
    ',' '"height"' ':' height=INT ) ( ',' '"width"' ':' width=INT ) ( ','
     '"node"' ':' '{' '"__type"' ':' '"Reference"' ',' '"__refType"' ':'
    '"Node"' ',' '"__value"' ':' '"'node=[Node:ID]'"' '}' ) '}' ;
Position: '{' '"__type"' ':' '"Position"' ( ',' '"__id"' ':' '"'__id=ID
    '"' ) ( ',' '"x"' ':' x=INT ) ( ',' '"y"' ':' y=INT ) ( ',' '"node"'
     ':' '{' '"__type"' ':' '"Reference"' ',' '"__refType"' ':' '"Node"'
     ',' '"__value"' ':' '"'node=[Node:ID]'"' '}' ) '}' ;
NodeType returns string: "decision" | "fork" | "join" | "merge";
TaskType returns string: "manual" | "automated";
Weight returns string: "low" | "medium" | "high";
```

Listing 7.2: Generated Langium grammar from the TypeScript-based definition

Listing 7.2 is then used to build the metamodel utilizing the `generate` CLI command of Langium. After the metamodel has been created, the model management using Langium is ready to use. As the VSCode extension, which is created by the generator, only starts the model server API, but not the GLSP server on the extension startup, the implementation of the language server startup needs to be extended to also start the GLSP server. This is done in Listing 7.3.

```
// Start the language server with the shared services
startLanguageServer(shared);
shared.workspace.WorkspaceManager.onWorkspaceInitialized((workspaceFolders)
    => {
  // Start the graphical language server with the shared services
  startGLSPServer({ shared, language: WorkflowDiagram }, workspaceFolders
    [0]);
  // Start the JSON server with the shared services
  startModelServer({ shared, language: WorkflowDiagram }, workspaceFolders
    [0]);
});
```

Listing 7.3: Start the Langium language server, Model Server API and GLSP server

GLSP client and server The Graphical Language Server Platform already provides a client implementation and a TypeScript-based GLSP server implementation. Therefore, the existing implementation can be used to rebuild the workflow diagram example. However, it needs to be adjusted to create a connection to the newly created model server API so that the model server API can handle the model management. Also, the commands that are used to edit the model need to be adjusted so that the model editing is done using the JSON patch functionality of the model server API instead of directly editing the source model and sending the updated model to the server.

The following files need to be edited to enable the model loading and editing via the model server API:

- **SourceModelStorage**: As discussed in Section 2.3 the loading and saving of a model are done in the `SourceModelStorage`. Therefore, the methods `loadSourceModel` and `saveSourceModel` have to be adjusted, as seen in Listing 7.4 and Listing 7.5. As can be seen, the model is loaded using the model server API request method, providing the URI of the model to request.

```
async loadSourceModel(action: RequestModelAction): Promise<void> {
    const sourceUri = this.getSourceUri(action);
    const rootUri = sourceUri;
    const rootUriDetails = `${sourceUri}d`;
    const root = await this.state.modelService.request(
      rootUri,
      isModel,
      "glsp"
    );
    this.state.setSemanticRoot(rootUri, root, rootUriDetails, rootDetails);
}
```

Listing 7.4: Updated implementation of the `loadSourceModel` function

```
saveSourceModel(action: SaveModelAction): MaybePromise<void> {
    const saveUri = this.getFileUri(action);
    this.state.modelService.save(saveUri, this.state.semanticRoot);
    this.state.modelService.save(`${saveUri}d`, this.state.
    semanticRootDetails);
    streamReferences(this.state.semanticRoot)
      .map((refInfo) => refInfo.reference.ref)
      .nonNullable()
      .map((ref) => findRootNode(ref))
      .forEach((root) =>
        this.state.modelService.save(root.$document!.uri.toString(), root)
      );
}
```

Listing 7.5: Updated implementation of the `saveSourceModel` function

- **ModelState**: By using the model server API, the model state is represented using Langium's AST representation. Therefore, the `ModelState` class has to be

adjusted so that an AST node represents the current model state. In Listing 7.6, the implementation of the `setSemanticRoot` function is illustrated, which sets the model's current state. Another important task of the `ModelState` class is updating the model's state. As the model server API implements the model update using the JSON patch functionality, a method utilizing the JSON patch is implemented, as can be seen in Listing 7.7.

```
setSemanticRoot(
    uri: string,
    semanticRoot: Model,
    uriDetails?: string,
    semanticRootDetails?: Model
  ): void {
    this._semanticUri = uri;
    this._semanticRoot = semanticRoot;
    this._packageId =
      this.services.shared.workspace.PackageManager.getPackageIdByUri(
        URI.parse(uri)
      );
    this.index.indexSemanticRoot(this.semanticRoot, this.
    semanticRootDetails);
  }
```

Listing 7.6: Set the current model state in the updated `ModelState` class

```
async sendModelPatch(patch: string): Promise<void> {
    this._semanticRoot = await this.modelService.patch(
      this.semanticUri,
      patch,
      "glsp"
    );
    this.index.indexSemanticRoot(this.semanticRoot, this.
    semanticRootDetails);
  }
```

Listing 7.7: Set the current model state in the updated `ModelState` class

The connection to the model server API has been created with the updated implementation of the `SourceModelStorage` and `ModelState`. However, other changes are needed so that the GLSP server can correctly display model elements and send updates to the model server API, editing the model's state.

One of these changes is the custom implementation of the `GModelIndex`. This class is needed to provide utility functions to easily access model elements' IDs and their paths inside the AST. In Listing 7.8, the `setSemanticRoot` function of the model index implementation can be seen. This function is responsible for collecting the IDs of all elements in the AST, as well as collecting the path to each ID.

```
indexSemanticRoot(root: Model, rootDetails?: Model): void {
    this.idToSemanticNode.clear();
    this.idToPath.clear();
```

```
    streamAst(root).forEach((node) => this.indexAstNode(node));
    this.collectIdToPath(
        JSON.parse(this.services.language.serializer.JsonSerializer.
    serialize(root)
      )
    );
  }
```

Listing 7.8: Set the current model state in the updated `ModelState` class

To display model elements correctly, the `GModelFactory` class needs to map AST nodes to model elements. In Listing 7.9, it can be seen how the model is created, while in Listing 7.10, the (minified) implementation of a mapping to a `GNode` is illustrated.

```
protected createGraph(): GGraph | undefined {
    const model = this.modelState.semanticRoot;
    const modelDetails = this.modelState.semanticRootDetails;
    this.graphBuilder = GGraph.builder().id(this.modelState.semanticUri);
    model.nodes
      .map((node) => this.createNode(node))
      .forEach((node) => this.graphBuilder.add(node));
    this.createEdgesAndMissingNodes(model, modelDetails).forEach((element)
    =>
      this.graphBuilder.add(element)
    );
    return this.graphBuilder.build();
  }
```

Listing 7.9: Create the `GModelRoot` of a model class

```
protected createTaskNode(taskNode: TaskNode): GNode {
    const node = GTaskNode.builder().id(taskNode.__id).type(ModelTypes.
    AUTOMATED_TASK);
    const size = this.modelIndex.findSize(taskNode.__id);
    if (size) {
      node.addLayoutOption("prefWidth", size.width);
      node.addLayoutOption("prefHeight", size.height);
      node.size(size.width, size.height);
    }
    const position = this.modelIndex.findPosition(taskNode.__id);
    if (position) {
      node.position(position.x, position.y);
    }
    return node.build();
  }
```

Listing 7.10: Create a `GNode` for an AST node of type `TaskNode`

After the `GModelFactory` has been implemented, the final step for the adjusted implementation of the GLSP server is that the operation handlers need to be changed so that they send JSON patches instead of the updated model to update the model's state. For this, a custom `Command` implementation is added, as can be seen in Listing 7.11.

In the `execute` function, the `sendModelPatch` function of the custom `ModelState` implementation is called to patch the model.

Utilizing this custom `Command`, in Listing 7.12, the implementation for the creation of a new `TaskNode` is illustrated.

```
export class WorkflowCommand implements Command {
  constructor(
    protected state: WorkflowModelState,
    protected modelPatch?: string
  ) {}

  async undo(): Promise<void> {
    await this.state.undo();
  }

  async redo(): Promise<void> {
    await this.state.redo();
  }

  canUndo?(): boolean {
    return true;
  }

  async execute(): Promise<void> {
    if (this.modelPatch) {
      await this.state.sendModelPatch(this.modelPatch);
    }
  }
}
```

Listing 7.11: Custom implementation of `Command` that is used to update the model state

```
  override createCommand(operation: CreateNodeOperation): Command {
    const modelPatch = this.createNode(operation);
    return new WorkflowCommand(this.modelState, modelPatch);
  }
  createNode(operation: CreateNodeOperation, taskType: string): string {
    const patch = JSON.stringify({ op: "add", path: "/nodes/-",
      value: {
        $type: "TaskNode",
        __id: createRandomUUID(),
        name: "newTaskName",
        taskType: "automated",
      },
    });
    return patch;
  }
```

Listing 7.12: Code parts of an operation handler that utilizes the new `Command` implementation

7.3.2 BIGUML Modeling Tool - scenario implementation

The BIGUML modeling tool is a graphical editor that can create different kinds of UML diagrams, including the class diagram, state machine diagram, package diagram, and use-case diagram [BIGb] [BIGa]. In this scenario, the functionality of the BIGUML tool will be rebuilt. However, implementing the whole UML diagram functionality would exceed the scope of this thesis; therefore, two selected diagram types will be rebuilt: the class diagram, which is the biggest diagram in the BIGUML tool, and the package diagram, which is the smallest one.

The current implementation of the BIGUML tool [BIGa] uses a Java-based model server and a Java-based GLSP server. As this thesis aims to move away from the Java-based technology stack, a TypeScript implementation for the GLSP server of the BIGUML tool is required in this evaluation step. Also, the Java-based model server is replaced by an implementation of the Langium-based model management solution, which has been implemented in this thesis.

Langium model management In BIGUML, the metamodel used to implement the model server is the UML metamodel. As in this evaluation step, only two diagram UML diagram kinds need to implemented, the TypeScript-based grammar definition is a reduced metamodel of the UML metamodel. Listing 7.13 shows the root element of the definition of the reduced UML metamodel.

```
@root
class Diagram {
  diagram: ClassDiagram | PackageDiagram;
  metaInfos?: Array<MetaInfo>;
}
```

Listing 7.13: Root element definition for the UML metamodel

As can be seen, only the definitions for the `ClassDiagram` and `PackageDiagram` are implemented. However, other diagram kinds can be easily added by extending the `diagram` property of the `Diagram` class with the needed diagram type.

GLSP server

The implementation of the GLSP server for the BIGUML tool is based on the already existing TypeScript-GLSP-server of the workflow diagram example. In the previous section, it has already been discussed which adjustments are needed for this example GLSP server to work with the newly created Langium-based model management. Therefore, only the additional implementation steps required to rebuild the BIGUML tool are discussed in this section.

The BIGUML tool offers a property palette next to the graphical editor. This form-based view can be used to update some properties of model elements that can not be edited within the graphical model. For example, a class can include attributes and operations

in the class diagram. For operations, it is not possible to add parameters and their types within the graphical editor; hence, the property palette can be used. To be able to handle actions of the property palette, handlers for the following two actions need to be implemented:

- **`RequestPropertyPaletteAction`**: The handler for this action must prepare the property palette's form according to the selected element. Therefore, to prepare this form, a utility class named `PropertyPaletteBuilder` has been created, which eases the creation of property palette items using the builder pattern. In Listing 7.14, the implementation of the builder functions to create a form input for text elements, bool elements, and choice elements can be seen.

```
text(elementId: string, propertyId: string, text: string, label: string) {
    this.proxy.items.push({
        elementId,
        propertyId,
        type: 'TEXT',
        disabled: false,
        text,
        label
    } as ElementTextProperty);
    return this;
}
bool(elementId: string, propertyId: string, value: boolean, label: string)
    {
    this.proxy.items.push({
        elementId,
        propertyId,
        type: 'BOOL',
        value,
        label
    } as ElementBoolProperty);
    return this;
}
choice(elementId: string, propertyId: string, choices: Array<{ label:
    string; value: string }>, choice: string, label: string) {
    this.proxy.items.push({
        elementId,
        propertyId,
        type: 'CHOICE',
        choices,
        choice,
        label
    } as ElementChoiceProperty);
    return this;
}
```

Listing 7.14: Builder functions in the `PropertyPaletteBuilder` to create form inputs for text, bool and choice elements

- **`UpdateElementPropertyPaletteAction`**: The implementation for this handler creates a `UpdateOperation`, which updates the value of a selected property inside the property palette.

After these action handlers have been implemented, they need to be added to the `DiagramModule` of the selected diagram implementation. Additionally, the function `configureActionHandlers` needs to be implemented as seen in Listing 7.15.

```
protected override configureActionHandlers(binding: InstanceMultiBinding<
    ActionHandlerConstructor>): void {
    super.configureActionHandlers(binding);
    binding.add(RequestPropertyPaletteActionHandler);
    binding.add(UpdateElementPropertyActionHandler);
}
```

Listing 7.15: Configure action handlers for the property palette

Figure 7.1 shows an example of the property palette for a model element of type class in the class diagram. As can be seen, all properties of the class are listed in the property palette, and input fields, checkboxes, and choice elements exist to change the values of these properties. Furthermore, references exist for the child elements that can be used to open the property palette for the selected child element as can be seen next to the properties and operations.

7.3.3 Evaluation of the scenarios

In this section, the capabilities of the workflow diagram example and the BIGUML tool are gathered to evaluate the scenarios. Following that, these capabilities are compared to the ones of the rebuilt solutions.

Workflow diagram capabilities The workflow diagram editor is a straight-forward graphical editor that provides the following functionalities:

1. **Create node**: It is possible to create nodes at the selected position.
2. **Edit node**: It is possible to edit the label of a node within the graphical editor.
3. **Move node**: It is possible to move the node to a different position in the view.
4. **Resize node**: It is possible to resize a node.
5. **Create edge**: If at least one node exists in the diagram, an edge can be created between a source node and a target node.
6. **Delete Node**: It is possible to delete nodes. If a node has incoming or outgoing edges, the edges are also deleted during the deletion process.

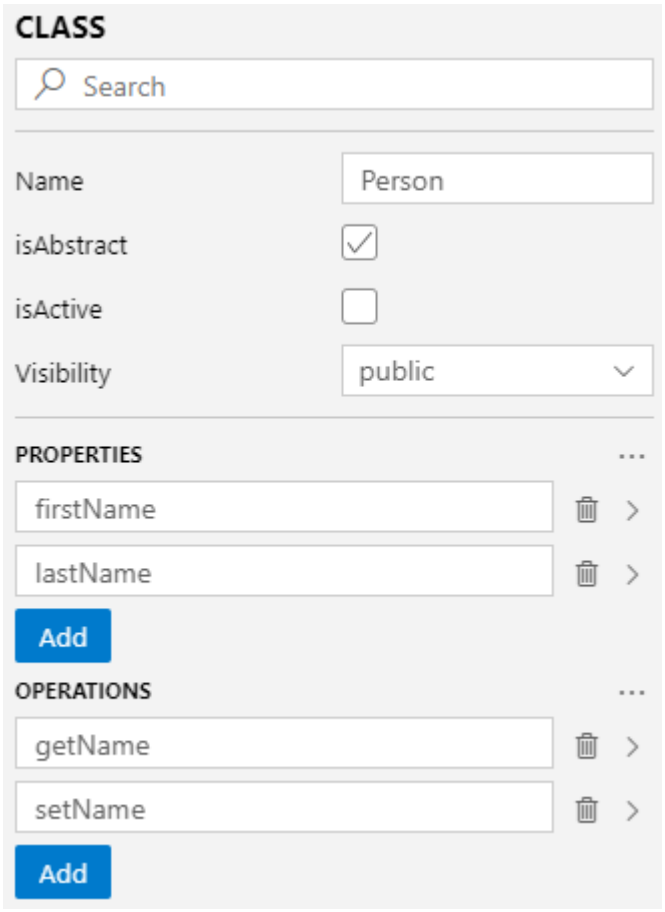


Figure 7.1: Property palette in the rebuilt BIGUML modeling tool

BIGUML modeling tool capabilities The capabilities of the BIGUML modeling tool include the previously mentioned points and the following additional functionalities:

7. **Show model element in property palette**: It is possible to load the properties of a model element into the property palette.
8. **Create child node via property palette**: It is possible to create child nodes for model elements via the property palette.
9. **Edit nodes via property palette**: It is possible to edit every property of a model element via the property palette.
10. **Delete child nodes via property palette**: It is possible to delete child nodes of model elements via the property palette.
11. **Show model structure in outline view**: It is possible to view the model's structure in a tree view.

Table 7.2 gives an overview of whether the respective editor's functionalities have been correctly rebuilt. As discussed in section 6.1, the patch functionality is used to make changes to the model. Therefore, every create and edit operation, as well as the delete and resize or move operations, have been successfully rebuilt in the tools, utilizing the model server API JSON patch. The previous section discussed the implementation of the property palette, which provides all the necessary functionality to rebuild the entire property palette feature of the BIGUML tool. The only functionality that could not be rebuilt is the outline tree view of the BIGUML tool. However, this functionality can be easily added to the current implementation by creating an action handler for the `RequestOutlineTreeView` action, which is responsible for loading and transforming the model data into the tree view data. The handler would need to access the current `ModelState` to request the model data and then transform it as needed for the outline view.

Requirement	**workflow diagram**	**bigUML**
Create node	✓	✓
Edit node	✓	✓
Move node	✓	✓
Resize node	✓	✓
Create edge	✓	✓
Delete Node	✓	✓
Show model element in property palette		✓
Create child node via property palette		✓
Edit nodes via property palette		✓
Delete child nodes via property palette		✓
Show model structure in outline view		(✓)

Table 7.2: Evaluation of the requirements of the rebuilt workflow diagram editor and the BIGUML modeling tool [gray background=initial solution does not support it; ✓=support implemented, (✓)=support partially implemented]

7.4 Interpretation of the evaluation results

In this section, the research questions of this thesis, which were defined in Section 1.2, are answered based on the evaluation results presented in Section 7.2 and Section 7.3.

Research Question 1 How can the Abstract Syntax Tree, which is created by Langium, be made available to model-oriented clients so that the editing of the model can be handled in the browser?

To enable model-oriented clients to access Langium's Abstract Syntax Tree (AST), the requirements for a model server API capable of handling such clients were analyzed by examining the Graphical Language Server Platform (GLSP). Following that, a concept

for the model server API (see Section 5.1) has been created, which provided the implementation ideas to fulfill these requirements. In the implementation process, two types of access possibilities to the model server API have been created: first, via a JSON RPC connection, and second, via a Langium service integrated into Langium's language server. For that, the language module for the language server has been extended with a package that implements the model server API.

After the model server API has been successfully integrated within Langium, in the evaluation, two state of the art graphical model editors utilizing the Graphical Language Server Protocol have been rebuilt and tested against some predefined criteria to check whether the newly created solution provides the same functionality as the initial implementation.

It was shown that the functionality of the initial solutions could be mainly rebuilt using the Langium-based model management solution, therefore realizing a browser-only model management solution for model-oriented clients.

Research Question 2 How should a type definition be conceptualized to create an accurate metamodel that includes all needed language concepts and cross-references?

Though Langium offers a grammar language to create a metamodel, to enable developers to stay within the TypeScript technology stack, a metamodeling language that utilizes TypeScript-native language concepts had to be conceptualized. For this, information about what elements are required to create a metamodel had to be gathered first. This has been done by analyzing the Eclipse Ecore metamodel and also the Ecore kernel [Ste09]. Following that, Langium's grammar language had to be analyzed to check what functionalities can be transformed from the Ecore metamodel to the TypeScript-based grammar language, which then is transformed into the Langium grammar definition.

After gathering all requirements, in Section 5.2, the concept of the TypeScript-based grammar language has been created. This encompasses the creation of simple model elements as well as the addition of cross-references between them.

In the evaluation, the TypeScript-based grammar language has been compared to Ecore's metamodel (see Section 7.2). Here, it has been highlighted that the TypeScript-based grammar language is able to mirror a lot of the functionalities that are implemented within Ecore, but also the limitations of the TypeScript-based grammar have been mentioned. These limitations exist both because of the limitations of Langium in comparison to Ecore and because of the limitations of the available language concepts in TypeScript. However, in spite of these limitations, in the scenario evaluation (see Section 7.3), it has been shown that the current TypeScript-based grammar language definition provides enough functionality to rebuild existing tools like the workflow-diagram example [Eclc] and parts of the functionality of the bigUML Modeling tool [BIGa].

Research Question 3 How can the previously defined type definition be used to generate a language specification in a generic JSON grammar?

This research question is not limited to developing a standard JSON grammar. This thesis also delves into generating the Langium-based model management, utilizing the TypeScript-based technology stack, using a custom generator implementation. Therefore, the capabilities of the generator functionality are also discussed in the answer to this question. For the generator functionality, the requirements to build a metamodel using the defined TypeScript-based grammar language had to be gathered first. After that, the required functionalities of the generator, which include the parsing of the TypeScript-based grammar language and the transformation of this definition to the Langium grammar, which creates a language in a generic JSON grammar, had to be collected. Subsequently, the concept for the generator has been presented in Section 5.3, which discusses how, on every change of the TypeScript definition file, the generator has to create a new Langium grammar definition, which creates parser rules, conforming a JSON structure.

After the generator functionality was implemented, the Langium-based model management for the scenarios was built using the generator. Furthermore, the generator's functionality generated the scenarios' metamodel. Finally, the resulting Langium grammar definition was tested in a textual editor to validate whether the models are always created as a generic JSON grammar.

CHAPTER 8

Conclusion

This chapter will conclude the thesis by summarizing the work done and presenting the main findings. Following that, possible future work for the discussed research topic will be presented.

8.1 Conclusions and Findings

The main objective of this thesis was to create a model management tool built entirely using a TypeScript-based technology stack, utilizing the next-generation language framework Langium. Furthermore, for the creation of the metamodel in Langium, a new grammar language, which uses valid TypeScript notations to define model elements, their attributes, and relations, had to be conceptualized. Finally, a generator had to be created, which combines the Langium model management with the TypeScript-based grammar definition, setting up the model management for the metamodel defined in the TypeScript definition.

The model management using Langium has been implemented by creating a model server API, which enables model-oriented clients to access the Abstract Syntax Tree (AST) created by Langium. The model server API had to include functionalities to open, request, update, close, and patch models. While the first four functionalities could be implemented straightforwardly, more consideration was needed for the patch functionality, as the patches could affect cross-references between different documents.

In the concept of the TypeScript-based grammar language, it had to be ensured that some functionalities like cross-references are supported; for this, decorators and custom interfaces have been created, which are recognized by the generator, so that it is possible to create the metamodel from the definition.

The implementation of this thesis has been completed by creating a generator, which combined all previously implemented artifacts to be able to create the TypeScript-based

model management from scratch. Due to the fact that the Java-based EMF modeling stack is widely used among model engineers, the implementation of the generator has been extended by functionality that eases the transition from Ecore metamodels to the TypeScript-based grammar language.

The evaluation of the work in this thesis has been performed in three steps: First, unit tests have been created for the generator functionality to verify the correctness of the implementation. Secondly, the newly created TypeScript-based grammar language has been tested against Ecore's metamodel to find its limitations. Finally, two state-of-the-art model editors utilizing GLSP have been rebuilt using the TypeScript-only technology stack. Here, it has been shown that both the grammar definition is expressive enough to create the metamodels for the respective editors' models, and the generator is able to set up a model management system that is capable of handling the requests of these state of the art model editors.

In summary, the work presented in this thesis demonstrates that the developed artifacts possess the ability to effectively manage the creation of metamodels and handle the model management process for advanced model editors.

8.2 Future Work

Although the artifacts developed in this thesis meet the presented requirements, there are still opportunities for improvement that could be explored in future works. The following sections discuss some potential improvements and challenges:

Model Server API The model server API currently only supports the editing of documents that are available via the local file system. Therefore, in future works, it could be analyzed how the model server API can be extended so that it can load, edit, and save files that are located on an external server.

Another interesting feature of the model server API would be to extend its functionalities to support multi-user editing, which would enable multiple users to edit a model simultaneously. To achieve this, the language server with the model server API must be hosted on an external server to provide centralized access. Additionally, the API needs to be extended with some functionality to notify all users who have the current model that it may require an update.

TypeScript-based grammar language In the evaluation, it has been shown that the TypeScript-based grammar language has some limitations in comparison to Ecore's metamodel. Therefore, in future works, it could be interesting to investigate ways to extend the grammar language so that it is possible to support more of Ecore's functionality, including exact multiplicities, derived attributes, and opposite references. Langium's grammar language limits the integration of some of these features. These functionalities may need to be added using services instead of being added directly in the grammar definition.

List of Figures

Listings

Acronyms

AST Abstract Syntax Tree. ix, xi, 2, 7, 16, 17, 20–22, 27–29, 35, 39, 48, 57, 58, 60, 61, 65, 66, 93–95, 101, 105, 111

CLI Command Line Interface. 20, 22, 27, 46, 92

DI Dependency Injection. 14, 16

DSL Domain-Specific Modeling Languages. 2, 5, 8, 11, 16, 26

EBNF Extended Backus-Naur Form. 17

EMF Eclipse Modeling Framework. xi, 3, 5, 8–11, 13, 27, 106, 107

GLSP Graphical Language Server Platform. ix, xi, 1, 3, 5, 12–16, 22, 24, 25, 28, 89, 92–95, 97, 101, 106, 107

GPL General Purpose Modeling Languages. 8

HTML Hypertext Markup Language. 8, 14

IDE Integrated Development Environment. ix, xi, 1, 11, 12, 107

LSP Language Server Protocol. ix, xi, 1, 5, 11–13, 16, 24, 27, 107

UML Unified Modeling Language. 8, 10, 97, 111

URI Uniform Resource Identifier. 36, 38, 58–61, 110

UUID Universally Unique Identifier. 38, 61–65, 110

XMI XML Metadata Interchange. 10

XML Extensible Markup Language. 46, 67, 68, 110

Bibliography

[BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Synthesis lectures on software engineering. Morgan & Claypool Publishers, [San Rafael, Calif.], second edition edition, 2017.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280. IEEE, 2001.

[BIGa] BIGModelingTools. BIGUML Modeling Tool. `https://marketplace.visualstudio.com/items?itemName=BIGModelingTools.umldiagram`. Accessed: 17.06.2023.

[BIGb] BIGModelingTools. BIGUML modeling tool. `https://github.com/borkdominik/bigUML`. Accessed 04.04.2024.

[BKP18] Dominik Bork, Dimitris Karagiannis, and Benedikt Pittl. Systematic analysis and evaluation of visual conceptual modeling language notations. In *12th International Conference on Research Challenges in Information Science, RCIS 2018, Nantes, France, May 29-31, 2018*, pages 1–11. IEEE, 2018.

[BKP20] Dominik Bork, Dimitris Karagiannis, and Benedikt Pittl. A survey of modeling language specification techniques. *Inf. Syst.*, 87, 2020.

[BL23] Dominik Bork and Philip Langer. Language server protocol: An introduction to the protocol, its use, and adoption for web modeling tools. *Enterp. Model. Inf. Syst. Archit. Int. J. Concept. Model.*, 18:9:1–16, 2023.

[BLO23] Dominik Bork, Philip Langer, and Tobias Ortmayr. A vision for flexible glsp-based web modeling tools. In João Paulo A. Almeida, Monika Kaczmarek-Heß, Agnes Koschmider, and Henderik A. Proper, editors, *The Practice of Enterprise Modeling - 16th IFIP Working Conference, PoEM 2023, Vienna, Austria, November 28 - December 1, 2023, Proceedings*, volume 497 of *Lecture Notes in Business Information Processing*, pages 109–124. Springer, 2023.

[Cho65] Noam Chomsky. *Aspects of the Theory of Syntax*. The MIT Press, 50 edition, 1965.

[CLB22] Giuliano De Carlo, Philip Langer, and Dominik Bork. Advanced visualization and interaction in glsp-based web modeling: realizing semantic zoom and off-screen elements. In Eugene Syriani, Houari A. Sahraoui, Nelly Bencomo, and Manuel Wimmer, editors, *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, Montreal, Quebec, Canada, October 23-28, 2022*, pages 221–231. ACM, 2022.

[DCLB22] Giuliano De Carlo, Philip Langer, and Dominik Bork. Advanced visualization and interaction in glsp-based web modeling: Realizing semantic zoom and off-screen elements. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS '22, page 221–231, New York, NY, USA, 2022. Association for Computing Machinery.

[Ecla] EclipseFoundation. Ecore. `https://wiki.eclipse.org/Ecore`. Accessed: 26.02.2024.

[Eclb] EclipseSource. Glsp. `https://eclipse.dev/glsp/`. Accessed: 14.02.2024.

[Eclc] EclipseSource. Workflow example. `https://www.eclipse.org/glsp/examples/#workflowoverview`. Accessed: 24.06.2023.

[Foua] Eclipse Foundation. The eclipse theia platform. `https://theia-ide.org/`. Accessed: 22.02.2024.

[Foub] Eclipse Foundation. Emf. `https://www.eclipse.org/modeling/emf/`. Accessed: 24.06.2023.

[Fouc] Eclipse Foundation. EMF.cloud. `https://www.eclipse.org/emfcloud/`. Accessed: 22.06.2023.

[Foud] Eclipse Foundation. Xtext. `https://www.eclipse.org/Xtext/`. Accessed: 24.06.2023.

[Fow10] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[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, 2021.

[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*, MODELS '22, page 22–26, New York, NY, USA, 2022. Association for Computing Machinery.

[Gro16] Object Management Group. Meta object facility. https://www.omg.org/spec/MOF/2.5.1, 2016. Accessed: 20.02.2024.

[Gro17] Object Management Group. Unified modeling language. https://www.omg.org/spec/UML/2.5.1, 2017. Accessed: 20.02.2024.

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.

[HR04] D. Harel and B. Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72, 2004.

[JBF11] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. *Model Driven Language Engineering with Kermeta*, pages 201–221. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[Kle08] Anneke Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008.

[Lan] Langium. Langium. https://langium.org/. Accessed: 17.06.2023.

[MB23] Haydar Metin and Dominik Bork. On developing and operating glsp-based web modeling tools: Lessons learned from BIGUML. In *26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023, Västerås, Sweden, October 1-6, 2023*, pages 129–139. IEEE, 2023.

[Mica] Microsoft. Extension api | visual studio code. https://code.visualstudio.com/api. Accessed: 22.02.2024.

[Micb] Microsoft. Language server protocol. https://microsoft.github.io/language-server-protocol/. Accessed 04.04.2024.

[Micc] Microsoft. Language server protocol - overview. https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/. Accessed 30.04.2024.

[Micd] Microsoft. Visual studio code. https://code.visualstudio.com/. Accessed: 22.02.2024.

[NL] CrossBreeze NL. Crossmodel. https://github.com/CrossBreezeNL/crossmodel. Accessed 05.03.2024.

[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*, MODELS '18, page 370–380, New York, NY, USA, 2018. Association for Computing Machinery.

[Sei03] Edwin Seidewitz. What models mean. *IEEE software*, 20(5):26–32, 2003.

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[SHK12] Martina Seidl, Marion Scholz Christian Huemer, and Gerti Kappel. *UML@ Classroom An Introduction to Object-Oriented Modeling*. Springer, 2012.

[Ste09] Dave Steinberg. *EMF : Eclipse Modeling Framework*. The eclipse series EMF. Addison Wesley, [Place of publication not identified], 2nd ed. edition, 2009.

[Tha22] Bernhard Thalheim. Models: the fourth dimension of computer science. *Software and Systems Modeling*, 21:1–10, 02 2022.

[VBD+13] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and GH Wachsmuth. Dsl engineering-designing, implementing and using domain-specific languages. 2013.

Appendix

Implementation of the mapping from EcoreDefinition TypeScript-based grammar language definition

```
export function transformEcoreToTsDefinition(
  definition: EcoreDefinition
): string {
  let tsDefinition = [
    'import { root, crossReference, CrossReference, ABSTRACT_ELEMENT, ROOT_ELEMENT }
    from "generator-langium-model-management";',
  ];
  definition.types.forEach((type) => {
    tsDefinition.push(`type ${type.name} = ${type.types.join(" | ")}`);
  });
  tsDefinition.push(
    definition.dataTypes.map((type) => `type ${type} = string;`).join("\n")
  );
  tsDefinition.push(
    ...definition.classes
      .sort(
        (classA, classB) =>
          (classA.extends?.length ?? 0) - (classB.extends?.length ?? 0)
      )
      .map((eClass) => {
        if (eClass.isInterface) {
          const extenders = eClass.extends ?? [];
          if (eClass.isAbstract) {
            extenders.push("ABSTRACT_ELEMENT");
          }
          if (eClass.isRoot) {
            extenders.push("ROOT_ELEMENT");
          }

          return `interface ${eClass.name} ${
            extenders.length > 0 ? "extends " + extenders.join(", ") : ""
          } {
            ${eClass.attributes
              .map(
                (attr) =>
                  `${attr.name}${
                    attr.multiplicity === "?" || attr.multiplicity === "*"
                      ? "?"
                      : ""
                  }: ${getTsType(attr, true)}`
              )
              .join("\n")}
          }`;
        } else {
          return `${eClass.isRoot ? "@root" : ""} ${
            eClass.isAbstract ? "abstract" : ""
          } class ${eClass.name} ${
            eClass.extends?.length > 0
              ? "extends " + eClass.extends.join(", ")
              : ""
```

```
          } {
            ${eClass.attributes
              .map(
                (attr) =>
                  `${attr.reference ? "@crossReference" : ""} ${attr.name}${
                    attr.multiplicity === "?" || attr.multiplicity === "*"
                      ? "?"
                      : ""
                  }: ${getTsType(attr)}`
              )
              .join("\n")}
          }`;
        }
      })
  );
  return tsDefinition.join("\n");
}
function getTsType(attribute: EcoreAttribute, isInterface: boolean = false) {
  let type = attribute.type;
  if (attribute.multiplicity === "*" || attribute.multiplicity === "+") {
    type = "Array<" + type + ">";
  }
  if (attribute.reference && isInterface) {
    type = `CrossReference<${type}>`;
  }
  return type;
}
```

Evaluation of Ecore models with TypeScript-based grammar language

Example1 - Ecore Model

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance"
  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="cpsml" nsURI="http://www.example.org/cpsml"
      nsPrefix="cpsml">
  <eAnnotations source="http://www.eclipse.org/OCL/Import">
    <details key="ecore" value="http://www.eclipse.org/emf/2002/Ecore"/>
  </eAnnotations>
  <eAnnotations source="http://www.eclipse.org/emf/2002/Ecore">
    <details key="invocationDelegates" value="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot"/>
    <details key="settingDelegates" value="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot"/>
    <details key="validationDelegates" value="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot"/>
  </eAnnotations>
  <eClassifiers xsi:type="ecore:EClass" name="CPS">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerBound="1" eType="ecore:EDataType http
        ://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="environment" lowerBound="1"
      upperBound="-1" eType="#//Environment" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="messagebroker" lowerBound="1"
      upperBound="-1" eType="#//MessageBroker" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="controller" lowerBound="1"
      upperBound="-1" eType="#//Controller" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="MessageBroker">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerBound="1" eType="ecore:EDataType http
        ://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="topics" upperBound="-1"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="position" lowerBound="1"
      eType="#//Position" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="connectionmodule" upperBound="-1"
      eType="#//ConnectionModule" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Environment">
```

```
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerBound="1" eType="ecore:EDataType http
      ://www.eclipse.org/emf/2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="node" lowerBound="1" upperBound="-1"
      eType="#//Node" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Controller">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerBound="1" eType="ecore:EDataType http
      ://www.eclipse.org/emf/2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="description" eType="ecore:EDataType http://www.
      eclipse.org/emf/2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="subscriptions" upperBound="-1"
      eType="#//MessageLink" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="publications" upperBound="-1"
      eType="#//MessageLink" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="topics" upperBound="-1"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Node" eSuperTypes="#//ComponentNode">
  <eAnnotations source="http://www.eclipse.org/emf/2002/Ecore">
    <details key="constraints" value="cons1 cons2"/>
  </eAnnotations>
  <eAnnotations source="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot">
    <details key="cons1" value="&#xA;&#x9;&#x9;&#x9;(self.status = Status::GOOD and self.component ->
        forAll(c : Component | c.status = Status::GOOD)) or self.status &lt;> Status::GOOD"/>
    <details key="cons2" value="&#xA;&#x9;&#x9;&#x9;self.publications -> forAll(p : MessageLink | self.
        connectionmodule -> exists(c : ConnectionModule | c.connectionmodule -> exists(cm :
        ConnectionModule | cm.MessageBroker = p.messagebroker))) and&#xA;&#x9;&#x9;&#x9;self.
        subscriptions -> forAll(s : MessageLink | self.connectionmodule -> exists(c : ConnectionModule | c.
        connectionmodule -> exists(cm : ConnectionModule | cm.MessageBroker = s.messagebroker)))"/>
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EReference" name="component" upperBound="-1"
      eType="#//Component" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="connectionmodule" upperBound="-1"
      eType="#//ConnectionModule" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="publications" upperBound="-1"
      eType="#//MessageLink" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="subscriptions" upperBound="-1"
      eType="#//MessageLink" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="position" eType="#//Position"
      containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EEnum" name="Status">
  <eLiterals name="GOOD"/>
  <eLiterals name="WARNING" value="1"/>
  <eLiterals name="CRITICAL" value="2"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Position">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="x" lowerBound="1" eType="ecore:EDataType http://
      www.eclipse.org/emf/2002/Ecore#//EFloatObject"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="y" lowerBound="1" eType="ecore:EDataType http://
      www.eclipse.org/emf/2002/Ecore#//EFloatObject"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="z" lowerBound="1" eType="ecore:EDataType http://
      www.eclipse.org/emf/2002/Ecore#//EFloatObject"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Component" abstract="true" eSuperTypes="#//ComponentNode">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="topic" lowerBound="1" eType="ecore:EDataType http
      ://www.eclipse.org/emf/2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="functions" upperBound="-1"
      eType="#//Function" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ConnectionModule" abstract="true">
  <eAnnotations source="http://www.eclipse.org/emf/2002/Ecore">
    <details key="constraints" value="constraintSameType constraintNameStartsWithTopicName
        compatibleProtocols"/>
  </eAnnotations>
  <eAnnotations source="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot">
    <details key="constraintSameType" value="&#xA;&#x9;&#x9;&#x9;self.connectionmodule -> forAll(c:
        ConnectionModule | c.oclIsTypeOf(self.oclType()) and c &lt;> self)"/>
    <details key="constraintNameStartsWithTopicName" value="&#xA;&#x9;&#x9;&#x9;(self.Node = null or
        self.name.size() > self.Node.name.size() and self.name.substring(1, self.Node.name.size()) = self.Node.name
        .toUpper()) and &#xA;&#x9;&#x9;&#x9;(self.MessageBroker = null or self.name.size() > self.
        MessageBroker.name.size() and self.name.substring(1, self.MessageBroker.name.size()) = self.
        MessageBroker.name.toUpper())"/>
```

```
    <details key="compatibleProtocols" value="&#xA;&#x9;&#x9;&#x9;self.connectionmodule->forAll(c:
        ConnectionModule | self.supportedProtocols->intersection(c.supportedProtocols)->notEmpty() )"/>
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerBound="1" eType="ecore:EDataType http
      ://www.eclipse.org/emf/2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="supportedProtocols" lowerBound="1"
      upperBound="-1" eType="#//CommunicationProtocol"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="connectionmodule" upperBound="-1"
      eType="#//ConnectionModule"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="WirelessModule" eSuperTypes="#//ConnectionModule" interface="
    true">
  <eAnnotations source="http://www.eclipse.org/emf/2002/Ecore">
    <details key="constraints" value="rangeRestriction cons4"/>
  </eAnnotations>
  <eAnnotations source="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot">
    <details key="rangeRestriction" value="&#xA;&#x9;&#x9;&#x9;range > 0"/>
    <details key="cons4" value="&#xA;&#x9;&#x9;&#x9;(self.MessageBroker = null or (self.connectionmodule->
        forAll(c: ConnectionModule | (&#xA;&#x9;&#x9;&#x9;&#x9;(self.MessageBroker.position.x - c.
        Node.position.x) * (self.MessageBroker.position.x - c.Node.position.x) + &#xA;&#x9;&#x9;&#x9;&#x9
        ;&#x9;(self.MessageBroker.position.y - c.Node.position.y) * (self.MessageBroker.position.y - c.Node.
        position.y) + &#xA;&#x9;&#x9;&#x9;&#x9;&#x9;(self.MessageBroker.position.z - c.Node.position.z) *
         (self.MessageBroker.position.z - c.Node.position.z)&#xA;&#x9;&#x9;&#x9;&#x9;).abs() &lt;= (self.
        range * self.range)) &#xA;&#x9;&#x9;&#x9;&#x9;)) and &#xA;&#x9;&#x9;&#x9;(self.Node = null or (self.
        connectionmodule->forAll(c: ConnectionModule | (&#xA;&#x9;&#x9;&#x9;&#x9;&#x9;(self.Node.
        position.x - c.MessageBroker.position.x) * (self.Node.position.x - c.MessageBroker.position.x) + &#xA
        ;&#x9;&#x9;&#x9;&#x9;&#x9;(self.Node.position.y - c.MessageBroker.position.y) * (self.Node.position.
        y - c.MessageBroker.position.y) + &#xA;&#x9;&#x9;&#x9;&#x9;&#x9;(self.Node.position.z - c.
        MessageBroker.position.z) * (self.Node.position.z - c.MessageBroker.position.z)&#xA;&#x9;&#x9;&#x9;&#x9
        ;&#x9;).abs() &lt;= (self.range * self.range)) &#xA;&#x9;&#x9;&#x9;))"/>
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="range" lowerBound="1" eType="ecore:EDataType http
      ://www.eclipse.org/emf/2002/Ecore#//EFloatObject"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="WiredModule" eSuperTypes="#//ConnectionModule"/>
<eClassifiers xsi:type="ecore:EEnum" name="CommunicationProtocol">
  <eLiterals name="DDS"/>
  <eLiterals name="MQTT" value="1"/>
  <eLiterals name="SMQTT" value="2"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="MessageLink">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="topics" upperBound="-1"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="messagebroker" lowerBound="1"
      eType="#//MessageBroker"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ComponentNode" abstract="true">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="status" lowerBound="1"
      eType="#//Status"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerBound="1" eType="ecore:EDataType http
      ://www.eclipse.org/emf/2002/Ecore#//EString"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Sensor" eSuperTypes="#//Component">
  <eAnnotations source="http://www.eclipse.org/emf/2002/Ecore">
    <details key="constraints" value="cons3"/>
  </eAnnotations>
  <eAnnotations source="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot">
    <details key="cons3" value="&#xA;&#x9;&#x9;&#x9;self.functions -> forAll(f : Function | f.hasReturn =
        true and f.returnDataType &lt;> DataType::NULL)"/>
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="frequency" lowerBound="1"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EIntegerObject"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Actuator" eSuperTypes="#//Component">
  <eAnnotations source="http://www.eclipse.org/emf/2002/Ecore">
    <details key="constraints" value="runsRestriction"/>
  </eAnnotations>
  <eAnnotations source="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot">
    <details key="runsRestriction" value="&#xA;&#x9;&#x9;&#x9;runs >= 0"/>
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="runs" lowerBound="1" eType="ecore:EDataType http
      ://www.eclipse.org/emf/2002/Ecore#//EIntegerObject"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="ratedRuns" lowerBound="1"
```

```
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EIntegerObject"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Function">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerBound="1" eType="ecore:EDataType http
        ://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="description" eType="ecore:EDataType http://www.
        eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="returnDataType" eType="#//DataType"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="hasReturn" lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="parameter" upperBound="-1"
        eType="#//Parameter" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EEnum" name="DataType">
    <eLiterals name="NULL"/>
    <eLiterals name="INTEGER" value="1"/>
    <eLiterals name="FLOAT" value="2"/>
    <eLiterals name="BOOLEAN" value="3"/>
    <eLiterals name="STRING" value="4"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Parameter">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerBound="1" eType="ecore:EDataType http
        ://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="dataType" lowerBound="1"
        eType="#//DataType"/>
  </eClassifiers>
</ecore:EPackage>
```

Example1 - TypeScript-based grammar language

```
import {
  root,
  crossReference,
  CrossReference,
  ABSTRACT_ELEMENT,
  ROOT_ELEMENT,
} from "generator-langium-model-management";
type Status = "GOOD" | "WARNING" | "CRITICAL";
type CommunicationProtocol = "DDS" | "MQTT" | "SMQTT";
type DataType = "NULL" | "INTEGER" | "FLOAT" | "BOOLEAN" | "STRING";
class CPS {
  name: string;
  environment: Array<Environment>;
  messagebroker: Array<MessageBroker>;
  controller: Array<Controller>;
}
class MessageBroker {
  name: string;
  topics?: Array<string>;
  position: Position;
  connectionmodule?: Array<ConnectionModule>;
}
class Environment {
  name: string;
  node: Array<Node>;
}
class Controller {
  name: string;
  description?: string;
  subscriptions?: Array<MessageLink>;
  publications?: Array<MessageLink>;
  topics?: Array<string>;
}
class Position {
  x: number;
  y: number;
  z: number;
}
abstract class ConnectionModule {
  name: string;
  supportedProtocols: Array<CommunicationProtocol>;
```

```
  @crossReference connectionmodule?: Array<ConnectionModule>;
}
class MessageLink {
  topics?: Array<string>;
  @crossReference messagebroker: MessageBroker;
}
abstract class ComponentNode {
  status: Status;
  name: string;
}
class Function_ {
  name: string;
  description?: string;
  returnDataType?: DataType;
  hasReturn: boolean;
  parameter?: Array<Parameter>;
}
class Parameter {
  name: string;
  dataType: DataType;
}
@root
class Cpsml {
  cps?: Array<CPS>;
}
class Node extends ComponentNode {
  component?: Array<Component>;
  connectionmodule?: Array<ConnectionModule>;
  publications?: Array<MessageLink>;
  subscriptions?: Array<MessageLink>;
  position?: Position;
}
abstract class Component extends ComponentNode {
  topic: string;
  functions?: Array<Function_>;
}
interface WirelessModule extends ConnectionModule {
  range: number;
}
class WiredModule extends ConnectionModule {}
class Sensor extends Component {
  frequency: number;
}
class Actuator extends Component {
  runs: number;
  ratedRuns: number;
}
```

Example1 - JSON Model

```
{
    "cps": [
        {
            "__type": "CPS",
            "__id": "id-0",
            "name": "prod",
            "environment": [
                {
                    "__type": "Environment",
                    "__id": "id-1",
                    "name": "ProductionFloor",
                    "node": [
                        {
                            "__type": "Node",
                            "__id": "id-2",
                            "component": [
                                {
                                    "__type": "Actuator",
                                    "__id": "id-6",
                                    "runs": 70,
                                    "ratedRuns": 100000,
```

```
                "topic": "temp",
                "functions": [
                    {
                        "___type": "Function_",
                        "___id": "id-7",
                        "name": "setTemperature",
                        "description": "Set_the_temperature␣
                            ↪ of_the_heated_print_bed",
                        "hasReturn": true,
                        "parameter": [
                            {
                                "___type": "Parameter",
                                "___id": "id-8",
                                "name": "temperature",
                                "dataType": "INTEGER"
                            }
                        ]
                    }
                ],
                "status": "CRITICAL",
                "name": "HeatedPrintBed"
            },
            {
                "___type": "Actuator",
                "___id": "id-123",
                "runs": 70,
                "ratedRuns": 200000,
                "topic": "production",
                "functions": [
                    {
                        "___type": "Function_",
                        "___id": "id-12312313",
                        "name": "startPrint",
                        "description":
                            ↪ "Start_the_printing_process",
                        "hasReturn": true
                    },
                    {
                        "___type": "Function_",
                        "___id": "id-1231231331",
                        "name": "getRemainingPrintDuration",
                        "description": "returns_the␣
                            ↪ remaining_print_duration",
                        "returnDataType": "INTEGER",
                        "hasReturn": true
                    }
                ],
                "status": "CRITICAL",
                "name": "HeatedPrintBed"
            }
        ],
        "connectionmodule": [
            {
                "___type": "WirelessModule",
                "___id": "id-9",
                "range": 100,
                "name": "THREEDPRINTER_wl1",
                "supportedProtocols": [
                    "MQTT"
                ],
                "connectionmodule": [
                    {
                        "___type": "Reference",
                        "___refType": "ConnectionModule",
                        "___value": "id-9"
                    }
                ]
            }
        ],
        "status": "CRITICAL",
        "name": "ThreeDPrinter"
    }
]
```

```
                }
            ],
            "messagebroker": [
                {
                    "___type": "MessageBroker",
                    "___id": "id-3",
                    "name": "id",
                    "position": {
                        "___type": "Position",
                        "___id": "id-4",
                        "x": 0,
                        "y": 0,
                        "z": 0
                    },
                    "connectionmodule": []
                }
            ],
            "controller": [
                {
                    "___type": "Controller",
                    "___id": "id-5",
                    "name": "id"
                }
            ]
        }
    ]
}
```

Example2 - Ecore Model

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="test" nsURI="http://www.example.org/test"
        nsPrefix="test">
  <eClassifiers xsi:type="ecore:EClass" name="League">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="newAttribute" eType="ecore:EDataType http://www.
        eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="players" upperBound="-1"
        eType="#//Player" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Game">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="frames" eType="ecore:EDataType http://www.eclipse.
        org/emf/2002/Ecore#//EIntegerObject"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="player" eType="#//Player"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Player">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="ecore:EDataType http://www.eclipse.
        org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="dateOfBirth" eType="ecore:EDataType http://www.
        eclipse.org/emf/2002/Ecore#//EDate"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="height" eType="ecore:EDataType http://www.eclipse.
        org/emf/2002/Ecore#//EDoubleObject"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="isProfessional" eType="ecore:EDataType http://www.
        eclipse.org/emf/2002/Ecore#//EBooleanObject"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Matchup">
    <eStructuralFeatures xsi:type="ecore:EReference" name="games" lowerBound="2" upperBound="2"
        eType="#//Game" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Tournament">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="type" eType="#//TournamentType"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="matchup" upperBound="-1"
        eType="#//Matchup" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EEnum" name="TournamentType">
    <eLiterals name="Pro"/>
    <eLiterals name="Amateur" value="1"/>
  </eClassifiers>
</ecore:EPackage>
```

Example2 - TypeScript-based grammar language

```
import {
  root,
  crossReference,
  CrossReference,
  ABSTRACT_ELEMENT,
  ROOT_ELEMENT,
} from "generator-langium-model-management";
type TournamentType = "Pro" | "Amateur";
class League {
  newAttribute?: string;
  players?: Array<Player>;
}
class Game {
  frames?: number;
  @crossReference player?: Player;
}
class Player {
  name?: string;
  dateOfBirth?: string;
  height?: number;
  isProfessional?: boolean;
}
class Matchup {
  games: Array<Game>;
}
class Tournament {
  type?: TournamentType;
  matchup?: Array<Matchup>;
}
@root
class Test {
  league?: Array<League>;
  tournament?: Array<Tournament>;
}
```

Example2 - JSON Model

```
{
    "league": [
        {
            "___type": "League",
            "___id": "id-0",
            "players": [
                {
                    "___type": "Player",
                    "___id": "id-1",
                    "name": "John_Doe",
                    "height": 6.5,
                    "isProfessional": true
                },
                {
                    "___type": "Player",
                    "___id": "id-2",
                    "name": "Jane_Doe",
                    "height": 5.5,
                    "isProfessional": false
                },
                {
                    "___type": "Player",
                    "___id": "id-3",
                    "name": "John_Smith",
                    "height": 6.0,
                    "isProfessional": true
                }
            ]
        }
    ],
    "tournament": [
        {
```

```
            "___type": "Tournament",
            "___id": "id-4",
            "type": "Pro",
            "matchup": [
                {
                    "___type": "Matchup",
                    "___id": "id-5",
                    "games": [
                        {
                            "___type": "Game",
                            "___id": "id-6",
                            "player": {
                                "___type": "Reference",
                                "___refType": "Player",
                                "___value": "id-1"
                            }
                        },
                        {
                            "___type": "Game",
                            "___id": "id-7",
                            "player": {
                                "___type": "Reference",
                                "___refType": "Player",
                                "___value": "id-3"
                            }
                        }
                    ]
                }
            ]
        }
    ]
}
```

Example3 - Ecore Model

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="Model" nsURI="http://www.example.org/test"
        nsPrefix="test">
  <eClassifiers xsi:type="ecore:EClass" interface="true" name="Fruit" abstract="true">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="fruitType" eType="#//FruitType"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" interface="true" name="WritingObject" abstract="true">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="writingObjectType" eType="#//WritingObjectType"/>

  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" interface="true" name="Fruit_WritingObject" eSuperTypes="#//Fruit #//
      WritingObject">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="mmmm" eType="#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EEnum" name="WritingObjectType">
    <eLiterals name="PEN"/>
    <eLiterals name="PENCIL" value="1"/>
    <eLiterals name="CRAYON" value="2"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EEnum" name="FruitType">
    <eLiterals name="APPLE"/>
    <eLiterals name="PINEAPPLE" value="1"/>
    <eLiterals name="ORANGE" value="2"/>
  </eClassifiers>
</ecore:EPackage>
```

Example2 - TypeScript-based grammar language

```
import {
  root,
  crossReference,
  CrossReference,
```

```
  ABSTRACT_ELEMENT,
  ROOT_ELEMENT,
} from "generator-langium-model-management";
type WritingObjectType = "PEN" | "PENCIL" | "CRAYON";
type FruitType = "APPLE" | "PINEAPPLE" | "ORANGE";
interface Fruit extends ABSTRACT_ELEMENT {
  fruitType?: FruitType;
}
interface WritingObject extends ABSTRACT_ELEMENT {
  writingObjectType?: WritingObjectType;
}
@root
class Model {
  fruit_writingobject?: Array<Fruit_WritingObject>;
}
interface Fruit_WritingObject extends Fruit, WritingObject {
  mmmm?: string;
}
```

Example3 - JSON Model

```
{
    "fruit_writingobject": [
        {
            "___type": "Fruit_WritingObject",
            "___id": "id-1",
            "mmmm": "mmmmm_APPLE_PEN",
            "fruitType": "APPLE",
            "writingObjectType": "PEN"
        },
        {
            "___type": "Fruit_WritingObject",
            "___id": "id-2",
            "mmmm": "MMMM_PINEAPPLE_PENCIL",
            "fruitType": "PINEAPPLE",
            "writingObjectType": "PENCIL"
        },
        {
            "___type": "Fruit_WritingObject",
            "___id": "id-3",
            "mmmm": "MMMMMMMM_ORANGE_CRAYON",
            "fruitType": "ORANGE",
            "writingObjectType": "CRAYON"
        }
    ]
}
```