



Combining Textual and Graphical Modeling with Next Generation Frameworks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Adam Lencses, BSc

Matrikelnummer 11708472

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork

Mitwirkung: Dipl.-Ing. Dr.techn. Philip Langer

Wien, 8. April 2024

Adam Lencses

Dominik Bork



Combining Textual and Graphical Modeling with Next Generation Frameworks

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Adam Lencses, BSc

Registration Number 11708472

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork

Assistance: Dipl.-Ing. Dr.techn. Philip Langer

Vienna, 8th April, 2024

Adam Lencses

Dominik Bork

Erklärung zur Verfassung der Arbeit

Adam Lencses, 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, 8. April 2024

Adam Lencses

Kurzfassung

Die Kombination von textueller und graphischer Modellierung, d.h. die Darstellung von textuellen Modellen in Form von Diagrammen, ist seit langem ein beliebtes Thema in Model Engineering. Modellierungswerkzeuge bieten den Benutzern oftmals nur die Möglichkeit, Modelle entweder in textueller oder in graphischer Form zu editieren. Bisher wurden Modellierungswerkzeuge, die beide Ansätze kombinieren, nur mit traditionellen Frameworks wie z.B. Xtext und EMF entwickelt. Die Frameworks der nächsten Generation, Langium und die Graphical Language Server Platform (GLSP), bieten neue Möglichkeiten wie erhöhte Modularität in der Architektur und Bereitstellungsoptionen, mehr Flexibilität im Design der Benutzeroberfläche, webbasierte und Cloud-freundliche Entwicklungsmöglichkeiten, während sie die Abhängigkeit von Java eliminieren.

Das Ziel dieser Arbeit ist die Kombination von textueller und graphischer Modellierung erneut zu erforschen und weiterzuentwickeln mit den next-generation Frameworks, Langium und GLSP. Es wird ein Konzept für kombinierte textuell-graphische Modellierung auf Basis dieser neuen Frameworks entwickelt, das einen Modellservice nutzt, um die Modifikationsmodelle des textuellen und graphischen Editors gemeinsam zu verwalten. Das Konzept berücksichtigt, dass der grafische und textuelle Editor auf dem gleichen Modell arbeiten müssen, gleichzeitiges Editieren des Modells in beiden Editoren möglich sein muss und nicht-semantische Informationen wie zum Beispiel Kommentare und Formattierungen in der textuellen Darstellung des Modells erhalten bleiben müssen. Das Konzept wird als Artefakt basierend auf der Workflow domänenspezifische Sprache realisiert. Ein bestehendes GLSP-Framework existiert bereits für die Workflow Sprache. Dies wird durch einen Langium-basierten Language Server erweitert, um textuelle Modellierung zu ermöglichen, sowie durch einen Modellservice, der den Zugriff auf das Modell und die Bereitstellung und Updates des Modells zwischen den textuellen und grafischen Editoren verwaltet.

Um die entwickelten Konzepte und Artefakte zu evaluieren, werden die implementierten Lösungen auch mit zwei UML-Anwendungsfälle des BIGUML Modellierungswerkzeug instantiiert: das Paketdiagramm und das Klassendiagramm. Diese beiden Anwendungsfälle werden gegen die vordefinierten Anforderungen an die textuell-graphische Modellierung evaluiert.

Abstract

Combining textual and graphical modeling i.e., representing textual models in the form of diagrams, has been a popular topic ever since in the field of model engineering. Most often modeling tools only provide users the possibility to create models either in textual form or in the form of a diagram, and the users have to decide upon initial creation of the model whether they would like to use a textual or a graphical model editor. So far, blended modeling tools combining both approaches have only been developed based on traditional frameworks e.g., Xtext and EMF. The next generation frameworks Langium and the Graphical Language Server Platform (GLSP) promote new opportunities such as increased modularity in architecture and deployment options, more flexibility in user interface design, web-based and cloud-friendly development possibilities, while eliminating the dependency to Java.

This thesis aims to revisit and explore the topic of combining textual and graphical modeling with the next-generation frameworks Langium and GLSP. A concept for blended textual-graphical modeling based on these frameworks is developed, which utilizes a model service to jointly manage the textual and graphical editor's underlying modification model. The concept considers that the graphical and textual editor must operate on the same model, simultaneous updates must be possible between the two editors and non-semantic information of the model must be maintained during updates of the model. The concept is realized as an artifact based on the Workflow language. An existing GLSP framework for the Workflow language providing graphical modeling is extended by a Langium language server providing textual modeling, and a model server handling model access, provision and updates between the textual and graphical editors.

To evaluate the developed concepts and artifacts, the implemented solution concepts are instantiated by two UML use cases of the BIGUML modeling tool: the package diagram and the class diagram. These two use cases are evaluated against the conceptualized requirements of blended textual-graphical modeling.

Contents

Kurzfassung	vii
Abstract	ix
Contents	xi
1 Introduction	1
1.1 Problem Statement & Motivation	1
1.2 Aim of this Thesis and Expected Results	4
1.3 Methodology	5
1.4 Summary and Structure of the Work	6
2 Background	9
2.1 Terminology	9
2.2 Graphical Language Server Platform (GLSP)	12
2.3 Langium	14
2.4 Summary	16
3 State of the Art	17
3.1 Combining Textual and Graphical Modeling	17
3.2 Textual-graphical Modeling Frameworks	18
3.3 Drawbacks Observed in the Listed Frameworks	24
3.4 Summary and Comparison of the Listed Frameworks	25
4 Concept	27
4.1 Main Idea and General Approach	27
4.2 Requirements for the Blended Textual-Graphical Modeling Framework	28
4.3 Framework Architecture	31
4.4 Model Server Concept	32
4.5 Solution Concepts for the Requirements	35
4.6 Summary	39
5 Prototype Implementation	41
5.1 Implementation of the Language Server	41
	xi

5.2	Implementation of the GLSP Server	58
5.3	Model Server	77
5.4	Model Synchronization	81
5.5	VS Code Extension	84
5.6	Summary	85
6	Evaluation	89
6.1	Evaluation Procedure	89
6.2	BIGUML Artifacts	89
6.3	Implementation of the BIGUML Scenarios	90
6.4	Scenario Evaluation	94
6.5	Summary and Discussion	96
7	Conclusion	99
7.1	Conclusion	99
7.2	Future Work	102
	List of Figures	103
	List of Tables	105
	List of Listings	107
	Acronyms	109
	Bibliography	111

Introduction

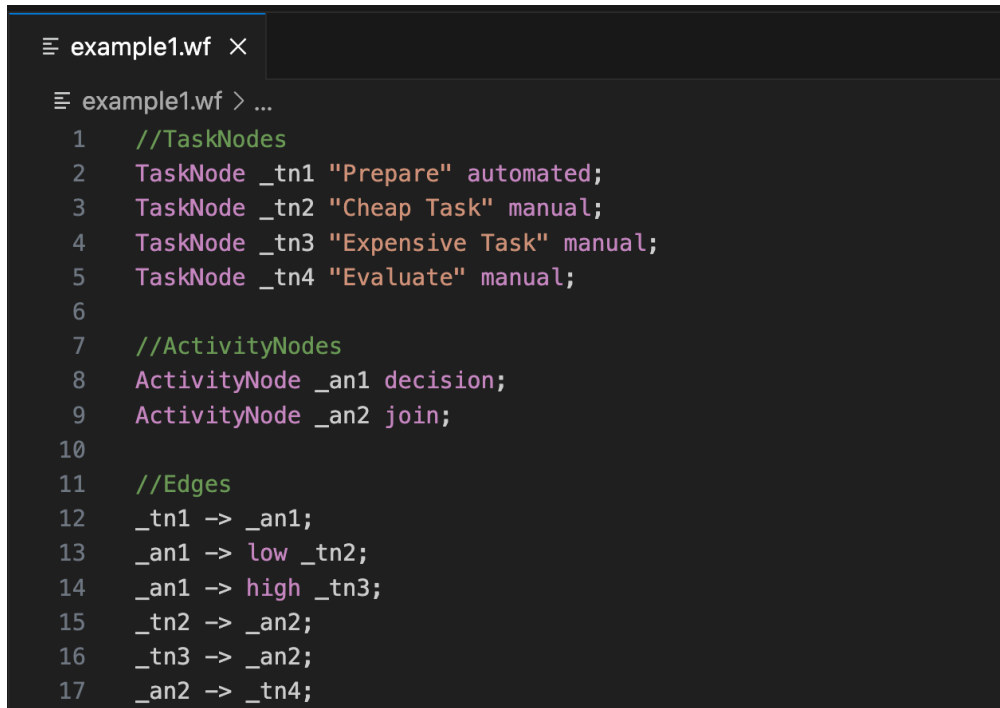
This chapter introduces the domain of this work. Firstly, it defines the motivation and problem statement for this thesis. Secondly, the aim of the thesis and expected results are elaborated providing four research questions that assist the implementation process of this thesis. Afterwards, the description of the applied methodological approach follows. Finally, the chapter gives an overview of the subsequent chapters of this thesis.

1.1 Problem Statement & Motivation

Combining textual and graphical modeling, i.e., representing textual models in the form of diagrams, has been a popular topic ever since in the field of model engineering [DLP⁺22] [Sch08]. Most often modeling tools only provide users the possibility to create models either in textual form or in the form of a diagram, and the users have to decide upon initial creation of the model whether they would like to use a textual or a graphical model editor. Both representation forms have their benefits and drawbacks. For example, it is easier to define fine-grained details and attributes of a model textually compared to editing small details in a diagram editor. The small details are easy to look up and edit in textual form, but can be difficult to find and modify in a diagram, depending on how the attributes of the model elements are visualized. On the other hand, it is easier to create and understand relationships between elements of a model when they are visually displayed in a diagram. Links between two elements of a diagram e.g., displayed in the form of edges are straightforward to recognize in a diagram, as edges are directly connected to their source and target elements. In a textual representation, the edges might not be defined directly next to the connecting elements, therefore the relationships between the elements of the model might not be recognized immediately.

Figures 1.1 and 1.2 illustrate the aforementioned differences between a textual and graphical model editor, both displaying the same model. The relationships (i.e., the

edges) between the elements of the model (i.e., the nodes) are straightforward to recognize on the graphical model, and harder to understand on the textual one.



```
example1.wf x
example1.wf > ...
1 //TaskNodes
2 TaskNode _tn1 "Prepare" automated;
3 TaskNode _tn2 "Cheap Task" manual;
4 TaskNode _tn3 "Expensive Task" manual;
5 TaskNode _tn4 "Evaluate" manual;
6
7 //ActivityNodes
8 ActivityNode _an1 decision;
9 ActivityNode _an2 join;
10
11 //Edges
12 _tn1 -> _an1;
13 _an1 -> low _tn2;
14 _an1 -> high _tn3;
15 _tn2 -> _an2;
16 _tn3 -> _an2;
17 _an2 -> _tn4;
```

Figure 1.1: VS Code’s Monaco editor: An example of a textual model editor.

Tools combining these two forms of modeling integrate the benefits of both textual and graphical modeling tools [DLP⁺22]. Users can usually decide whether they want to interact with the model textually or visually, depending on their preferences or even edit the model in both representations simultaneously.

Different modeling tools use different approaches to combine textual and graphical modeling. They can, for example differ in the level of granularity they offer users to interact with each representation of the model, or in the way the different representations are synchronized. Tools that derive a read-only visual representation of the textual model, only allow users to edit and interact with the textual model, and the visual representation is generated from the textual model afterward. Some tools can embed one representation within the other, such as embedding a textual editor in the visual representation of the model. This enables the users to textually edit small details of the model while maintaining the benefits of the visual representation for the bigger picture. Other tools enable editing a model both textually and graphically simultaneously, others only allow editing a model either textually or graphically at the same time, while still enabling cross-references between the elements of the graphical and textual models.

Depending on the approach, different conceptual challenges [SL] arise while combining textual and graphical modeling. Firstly, graphical and textual models both need an

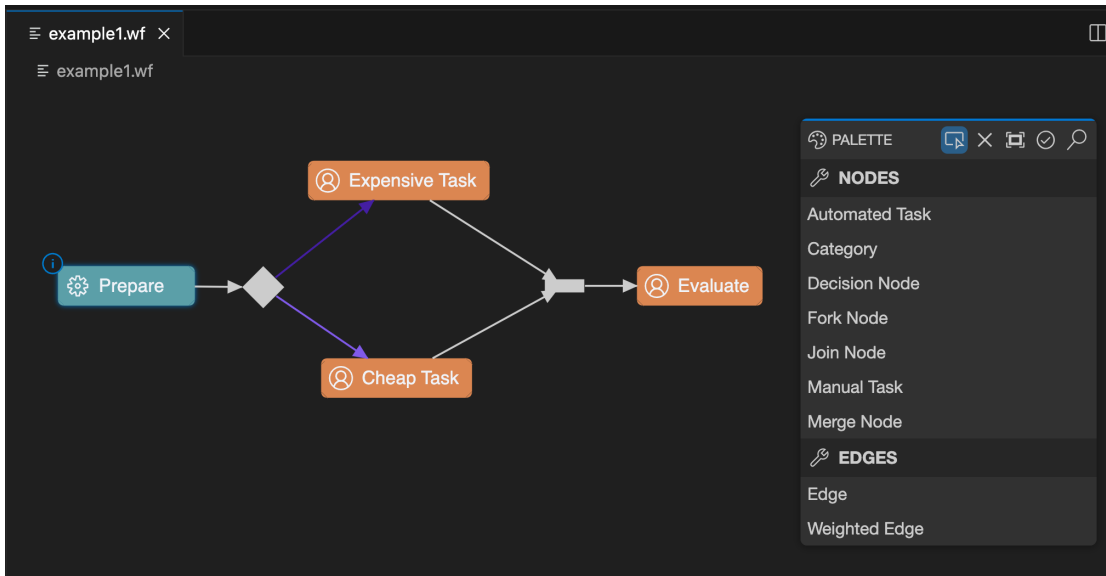


Figure 1.2: Graphical Language Server Platform (GLSP) Client: An example of a graphical model editor.

underlying modification model. A graphical editor, for example, might need additional attributes for an element to display it correctly, such as the position or the style of this element. These attributes, however, might not be relevant for a textual model, therefore, the underlying modification model must deal with this problem. Secondly, cross-references should be resolved correctly both in the textual and graphical representation of the model. A graphical editor should be aware of cross-references in textual models and display the corresponding elements accordingly and vice versa. For example, an edge between a source and a target element in a textual model should be displayed between the same source and target elements in the graphical representation of the model. Moreover, the non-semantic information of the graphical and textual representations must also be considered. The textual representation of a model might contain comments, special formatting or white spaces, that are irrelevant for the graphical representation. Also, the graphical representation might contain information such as the size or position of the elements that are irrelevant for the textual representation.

So far, these challenges have only been evaluated based on traditional frameworks e.g., Xtext [Foue] and EMF [Foub]. A re-evaluation based on the next-generation frameworks Langium [Typa] and the GLSP [Foua] [MB23b] is important as they promote new opportunities such as increased modularity in architecture and deployment options, more flexibility in user interface design, web-based and cloud-friendly development possibilities, while eliminating the dependency to Java [BLO23].

1.2 Aim of this Thesis and Expected Results

This thesis aims to revisit and explore the topic of combining textual and graphical modeling with the next-generation frameworks Langium and GLSP. The following results are expected from the new textual-graphical modeling framework, followed by the research questions that this thesis aims to provide answers for:

- **Model service:** An API to make Langium's abstract syntax tree (AST) accessible to GLSP and other model-oriented clients. This API must allow for accessing, querying, and manipulating models. The following question will follow the implementation of the model service API:
 1. **Model service:**
How can the model service API allow textual and graphical editors to manage and manipulate the underlying AST jointly?
- **Solutions to the conceptual challenges:** solutions and approaches that address the conceptual challenges from Section 1.1 utilizing the next-generation frameworks Langium and GLSP. The framework must propose solutions for the modification model, the reference resolution mechanism, and non-semantic information handling. The conceptualization and implementation of the textual-graphical modeling framework will provide example solutions to the following questions:
 2. **Modification model:**
How to implement the modification model to allow for simultaneous modifications on the textual and graphical models?
 3. **Cross-references:**
How can the Langium-GLSP framework resolve cross-references in the textual and graphical representations?
 4. **Non-semantic information:**
How to handle non-semantic information with the Langium-GLSP framework for textual and graphical models?

The integration of Langium and GLSP will be implemented via the following approach. A blended graphical and textual modeling framework will be developed, where the underlying modification model is shared between the textual and graphical language servers. This approach allows for editing models both graphically and textually, either simultaneously or in one representation at a time.

Successful integration of Langium and GLSP via this approach comprehends the implementation of the model service API, and proposing solutions for the mentioned conceptual challenges, contributing to the outcome of the problem statement of this thesis.

1.3 Methodology

The methodological approach of this work is based on the Design Science Research methodology [HRM⁺04]. The methodology of this thesis entails the problem's relevance, the conceptualization of the proposed design, the artifact creation and evaluation and the research contribution. The following steps will be executed during the implementation of the thesis.

1. Literature review

State-of-the-art approaches to combine textual and graphical modeling have to be reviewed and documented. Both approaches utilizing conventional frameworks (EMF, Xtext) and next-generation frameworks will be reviewed and categorized by the method of combining textual and graphical modeling. This step is part of the Design Science Research rigor cycle [Hev07], and is necessary to understand approaches that were already implemented, to document their features and to highlight what is missing and yet to be implemented. This provides a ground truth for this thesis and ensures that the research done is innovative.

2. Library and framework analysis

Several tools, libraries, and frameworks already exist that utilize Langium as a language server and a diagramming tool e.g., Sprotty [Ecle] to combine textual and graphical modeling. Investigating and analyzing these are crucial to understand the state of the art and to identify what is still missing and needs improvement. Furthermore, a review of Langium and GLSP will be provided, as the artifacts will be implemented based on these frameworks.

3. Conceptualization

Based on the previous steps, a concept of the blended modeling framework will be implemented. This concept will provide a detailed description of an approach on how textual and graphical modeling can be combined and which benefits and drawbacks this approach has. The concept will be based on the next-generation modeling tools Langium and GLSP. The concept will provide answers to the previously stated research questions in Section 1.2: it provides a concept for the model service to be used with multiple clients (textual and graphical), provides a concept for the common modification model for the textual and graphical editors, describes how unresolvable elements - common when interacting with text-based models, but uncommon for graphical model editors - will be handled, and provides a concept on how to deal with non-semantic information. This step is part of the Design Science Research design cycle [Hev07].

4. Artifact implementation

Langium and GLSP will be extended with the following artifacts:

- **Model service API:** a generic model service API as a Langium service that allows for accessing, querying, and manipulating models, as well as representing and resolving cross-references.
- **Langium-GLSP framework:** a framework to support and simplify combining textual and graphical modeling based on Langium and GLSP.

The Langium-GLSP framework will be implemented based on the Workflow domain-specific language (DSL) [Foud], as a Node.js based GLSP server and client are already implemented, and can be used as a basis for the Langium-GLSP framework.

5. Artifact evaluation

The implemented artifacts will be evaluated on their usability and reusability on another DSL based on descriptive evaluation using scenarios [Hev07]. The implemented artifacts will be instantiated and evaluated based on two BIGUML [Bor] [MB23a] use cases. The evaluation criteria is the successful instantiation of the Langium-GLSP framework and the model service with BIGUML as a proof of concept.

1.4 Summary and Structure of the Work

This chapter provided an introduction to the topic of graphical and textual modeling, stated the aim and expected results of the thesis and introduced the methodology of the work. Six more chapters are following.

Chapter 2 provides a background to the reader on modeling in general and on the technologies that will be further used for conceptualization and artifact implementation.

Chapter 3 elaborates the state of the art approaches to combine textual and graphical modeling. Approaches using conventional frameworks and next-generation frameworks will be introduced and discussed.

Chapter 4 presents the concept that proposes solutions to the research questions. It describes which approaches were chosen to find answers to the graphical-textual modeling problems, and why. It also describes which decisions should work in general, and which decisions might need other solutions for arbitrary DSLs.

Chapter 5 gives details on the implementation of the model service and the Langium-GLSP framework. This chapter highlights which concepts were used to implement the framework tailored for the Workflow DSL, and which decisions were made to provide support for this language.

Chapter 6 elaborates the evaluation process. It provides details on how the implemented framework was instantiated with two BIGUML use cases: the package diagram and the

class diagram, which problems arose and what solutions were provided to solve these problems, or argues why the problem is not solvable.

Chapter 7 concludes the thesis, provides answers to the research questions and possibilities for further work on this topic.

Background

This chapter gives an overview on the basic terminology used in this thesis. Furthermore, this chapter also provides an introduction to Langium and GLSP, as these frameworks will be used for the artifact implementation.

2.1 Terminology

This section provides an overview on the most important terms in regards of the topic of this thesis.

2.1.1 Model

Several different definitions are given in the literature for models.

Thalheim states that a generic notation of a model is the following [Tha22]: "A model is a well-formed, adequate, and dependable instrument that represents 'something' (called origin as a source, archetype, starting point) and functions in scenarios of use."

Kühne defines models as follows [Kü06]: "A model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made."

Selic states [Sel03] that "Models help us understand a complex problem and its potential solutions through abstraction."

All the above definitions state that models are abstract and hence represent something. This representation helps us understand the bigger picture (the 'something' the model represents) in different scenarios, allows us to make predictions and to establish potential solutions to the represented problem.

In the scope of this thesis the following example can be constituted: a model is for example a representation of a workflow written in the Workflow language, that describes

the tasks of a workflow and the dependencies between these tasks i.e., which tasks need to be finished before the next tasks can be started. An example for a model written in the Workflow language is pictured in Figure 1.2.

2.1.2 Modeling languages

Modeling Language

A modeling language is a textual or graphical language that is used to establish a model. The most important structural elements of a modeling language are its abstract syntax, concrete syntax, and semantics [CGR09].

Abstract syntax

The abstract syntax of a model defines the elements of a model and how these elements are related to each other and can be combined. Wile states that [Wil97] "The goal of an abstract syntax is to describe the structural essence of a language." An abstract syntax tree represents the abstract syntax in a form of a tree, while there are many more formalisms and techniques to specify the abstract syntax (i.e., the metamodel) of a modeling language [BKP20].

Concrete syntax

The concrete syntax of a model is the language's notation [HR00] [BKP18] i.e., the notation to illustrate the elements of a model textually or graphically. A textual model is a model notated with textual concrete syntax and a graphical model is notated with graphical concrete syntax. An example for a textual model is in Figure 1.1 and for a graphical model is in Figure 1.2.

Semantics

Harel et al. [HR04] state that "A language's semantics must provide the meaning of each expression, and that meaning must be an element in some well- defined and well-understood domain." Semantics define the actual meaning of the elements of a model and how they should be interpreted. Following the previous example the semantics of the model displayed in Figure 1.2 can be interpreted as four tasks where the 'Expensive Task' and 'Cheap Task' both need the 'Prepare' task to be done, and the 'Evaluate' task can only begin when either the 'Expensive Task' or the 'Cheap Task' is done.

Domain-specific language (DSL)

A domain-specific language is a language established for a specific domain that is not intended to support usage in an arbitrary area. Within the scope of this thesis a language called Workflow language will be used during the artifact development. This language is specifically established for demo purposes on the used architecture and is an example for a domain-specific language.

General-purpose language (GPL)

A general-purpose language is designed to be used in all kinds of scenarios, they can be applied to any domain for model creation. An example for a general-purpose language is the Unified Modeling Language (UML) [Obj]. This language will be used for artifact evaluation in this work.

2.1.3 Blended modeling

Blended modeling was first defined by Ciccozzi et al. [CTVW19]: "Blended modeling is the activity of interacting seamlessly with a single model (i.e., abstract syntax) through multiple notations (i.e., concrete syntaxes), allowing a certain degree of temporary inconsistencies." A single model means that the editing is done on the same model irrespective of which view of the model is currently being edited. Multiple notations means that the model for example can be edited in a textual view and in a graphical view simultaneously, and inconsistencies can result depending on the view's editing capabilities e.g., deleting a character from a cross-reference's ID in a textual model would already break the visualization of this reference in the graphical view, even though this step is probably only temporary between replacing IDs in the textual view.

2.1.4 Metamodeling

Metamodel

A metamodel of a model goes another abstraction layer further and defines the language elements and their grammar to specify the concrete syntax of the model [BCW17]. A grammar is a set of rules that predefines which language elements a model can have and their concrete notation.

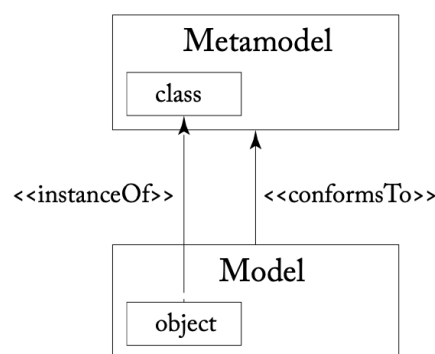


Figure 2.1: Metamodeling: 'conformsTo' and 'instanceOf' relationships [BCW17]. (page 15)

Meta-metamodel

A meta-metamodel is another layer of abstraction that the metamodel conforms to. This describes which language concepts the metamodel is allowed to contain and their grammar. Another level of abstraction is usually not necessary, as it is shown that meta-metamodels can be defined using themselves [BCW17].

Figure 2.1 shows the relationship between a model and its metamodel layer in correlation with the object-class relationship from object-oriented programming. A model can be always defined as conforming to the corresponding metamodel in the same way as an object is an instance of a class.

Continuing the previous examples based on the Workflow language, the following can be observed in regards to metamodeling. Figure 1.1 and Figure 1.2 show two examples of a model. Their corresponding metamodel is the Workflow language itself, i.e., the grammar defining the notation of elements and relationships between them. The meta-metamodel of the workflow metamodel within the scope of this thesis is the Langium grammar language [Typec], which is implemented using itself.

2.2 Graphical Language Server Platform (GLSP)

The GLSP is an extensible open-source framework for building custom diagram editors based on web technologies [Foua] [MB23b]. It is based on an extensible client-server architecture. The communication between the client and server is based on the Language Server Protocol (LSP) [Mich] [BL23], which is extended to provide features required for diagram editing. This way, the client only has to cope with the rendering of the diagram and providing editing possibilities for the users, while the server handles the more computational heavy tasks e.g., modifying the underlying abstract syntax of the diagram, loading the diagram and handling user inputs made on the client.

GLSP provides implementations for the client and server in several different frameworks and programming languages. For the scope of this thesis, the node.js server will be extended to match Langium, written in TypeScript. As for the client, the `glsp-vscode-integration`¹ package will be used that provides an example implementation of GLSP as a VS Code extension using Sprotty [Ecle] as a diagram editor integrated into VS Code's webview.

2.2.1 GLSP architecture

The GLSP framework is built up as follows. The server can be written in any programming language, most commonly either in Java or TypeScript. The server has to load the source model, which can be in an arbitrary format e.g., in JSON or a parsed AST from Langium and store it in the model state. It then has to create the graphical model of this loaded source model. The graphical model contains all the elements that the client will display

¹<https://github.com/eclipse-glsp/glsp-vscode-integration>

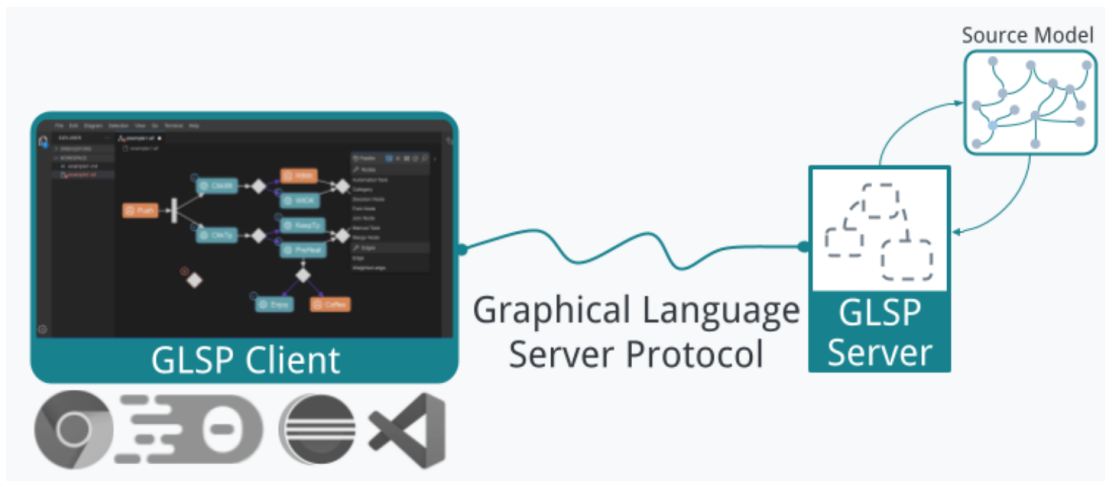


Figure 2.2: GLSP: client-server architecture [Foua]

and all their necessary attributes e.g., size, position, element type, label. This graphical model is then serialized by the server and sent to the client over the JSON-RPC² protocol. The server also provides implementations for actions that are coming from either the client or the server itself that modify the model state directly. After every modification on the model state, the server re-generates the graphical model and sends it to the client. This way, the architecture provides a clearly uni-directional flow of data, enabling the client to be a lightweight web-based editor.

The client is responsible for rendering the diagram and providing editing operations for the diagram, that are predefined by the server e.g., manipulating the size or position of the elements or renaming labels. The client however, does not directly apply changes to the diagram, it only sends a request to the server to do so, and re-renders the diagram after the server sends over the modified graphical model.

2.2.2 Components and services

To provide customizability and extensibility, the server and client are both implemented using an inversion of control pattern based on dependency injection³. On the server, all of the provided services i.e., the action handlers, and the components i.e., the model state are placed in a global dependency injection container and can be either extended or completely overwritten. The following section lists the most important services and components of the GLSP framework.

- **Source model storage:** is responsible for loading and saving the source model, which can be in arbitrary format.

²<https://www.jsonrpc.org/specification>

³<https://eclipse.dev/glsp/documentation/overview>

- **Model state:** holds information about the current model state and the original source model. The actions and services have access to this model state and can directly modify it.
- **Model index:** indexes the elements of the graphical model and the semantic elements of the model state they represent.
- **GModel factory:** the component that transfers the model state into the corresponding graphical model. It creates the diagram elements that are represented via a graph, the `GGraph`. This graph gets sent to the client for rendering. The `GModelFactory` re-creates and resends the graphical model to the client every time a change on the model state occurs.
- **Diagram module:** binds all the custom implemented or extended components and services to the framework.
- **Command:** is the core component of an action. A command defines how an action will be executed in regards of the model state and defines the necessary modifications to undo and redo the command.
- **Operation handler:** defines how an action modifies the model state and executes the corresponding command.
- **Provider:** providers define custom operations for the client that it can execute on a model i.e., navigation between elements or defining actions of a command palette.
- **Validator:** custom validators can validate the structure and attributes of the model, and can define error markers to show on the graphical model for the client.

2.3 Langium

"Langium is an open source language engineering tool with first-class support for the Language Server Protocol, written in TypeScript and running in Node.js [Typa]." Langium provides the possibility to create domain-specific languages together with an out of the box TypeScript-based language server, that can be easily integrated into VS Code as an extension or other web applications, and can be arbitrarily customized to meet the language creators' needs.

Giner-Miguelez et al. implemented a modeling tool called DescribeML [GMGC22], which is a dataset description tool for machine learning developed with Langium. The implemented tool uses a Langium-based grammar to define the created DSL's syntax and provides custom services showcasing Langium's language engineering capabilities.

2.3.1 Langium workflow

The Langium framework is also built based on dependency injection. All its default services and other components that the framework provides out of the box can be arbitrarily customized, completely replaced or extended.

The most important element of a Langium project is the grammar file that describes the grammar of the DSL for which the language server should be created. Langium has its own Langium grammar language⁴, which is based on EBNF that Xtext's grammar language is also based on. The grammar defines the structure of the AST which is created after Langium parses a document written in the specified language.

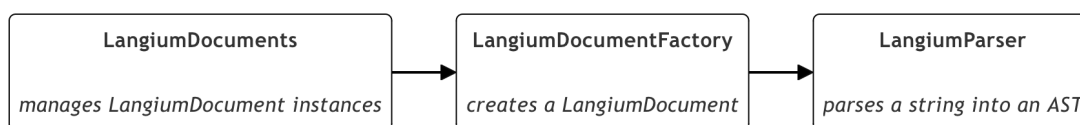


Figure 2.3: Creation of LangiumDocuments [Typb]

The LangiumDocument is the main data structure of the language server that represents a text document written in the specified DSL. The LangiumDocumentFactory creates a LangiumDocument utilizing the LangiumParser, that parses the text document based on the created grammar. The LangiumDocuments service manages the LangiumDocument instances that are loaded by the language server. This process of creating LangiumDocuments is shown in Figure 2.3.

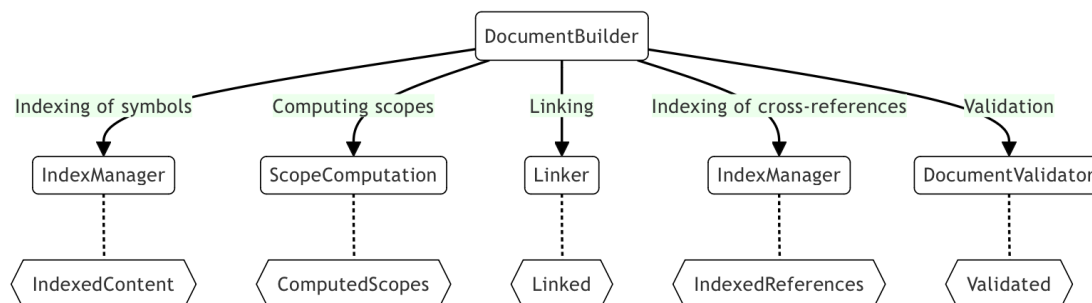


Figure 2.4: Stages of a LangiumDocument [Typb]

After the LangiumDocument was parsed and created, it has to be built by the DocumentBuilder service. During the build process, the LangiumDocument goes through following stages [Typb], which are also illustrated in Figure 2.4:

1. Parsed: after the LangiumDocument has been created and parsed by the LangiumParser. At this point the AST of the document has already been

⁴<https://langium.org/docs/grammar-language/>

created. The AST of a `LangiumDocument` is TypeScript-based and the types of the abstract syntax are generated based on the grammar.

2. `IndexedContent`: the `IndexManager` service collects all the symbols of the documents that could be cross-referenced from another document or from the document itself e.g., an ID attribute of an element. The symbols from all the indexed documents are collected together building the global scope.
3. `ComputedScopes`: after the global scope has been computed, the `ScopeComputation` service computes the local scope of every `LangiumDocument`, to define which named symbols belong to which `LangiumDocument`, and which symbols are accessible from which `LangiumDocument`.
4. `Linked`: in this stage the `Linker` services tries to resolve all the cross-references of a `LangiumDocument`. It queries the computed scope that is accessible to the given `LangiumDocument` and loads the AST node of the cross-reference if available within the computed scope.
5. `IndexedReferences`: during this stage an index for the previously obtained cross-references is created, which defines which `LangiumDocument` has dependencies to another document.
6. `Validated`: during validation, all the errors occurred in the previous stages are collected and organized by the stage in which the error occurred, and the severity of the error.
7. `Changed`: after a document has changed in the text editor, the `LangiumDocument` gets invalidated and either completely removed (if the file was deleted) or gets re-parsed and rebuilt again. If the indexed references of other `LangiumDocuments` contain references to the invalidated document, they also run through the linking phase again.

This document workflow allows Langium to act as a language server that is communicating with a language client based on the LSP. Furthermore, as Langium is based on node.js and TypeScript, it makes Langium a great choice as a language server for the aims of this thesis.

2.4 Summary

This chapter provided an overview on the basic terminology used in this thesis, and an introduction to Langium and GLSP. These frameworks are used for the artifact implementation. The following chapter provides an overview on the state of the art blended modeling tools.

State of the Art

This chapter gives an overview on the state of the art to combine textual and graphical modeling. Approaches using conventional frameworks and next-generation frameworks are listed. Furthermore, this chapter also elaborates what drawbacks the listed approaches have and which observed problems this thesis is intended to solve.

3.1 Combining Textual and Graphical Modeling

Most of the popular modeling tools are focusing on only one type of modeling: either textual or graphical but not both at the same time [AG13]. Examples for graphical only modeling tools are Eclipse Sirius [Fouc], Modeling SDK for Visual Studio [Micc] and Eclipse GMF Runtime [Ecla]. These frameworks enable users to create graphical DSLs for an arbitrary domain. These graphical DSLs are tailored to hold graphical information relevant for the specific framework which the creators of the DSLs can specify i.e., how each diagram element should look like, which type of elements the DSL should consist of and so on. Examples for textual modeling frameworks are Xtext [Foue], and Langium [Typa]. These frameworks provide out of the box solutions for creating textual DSLs, and provide default implementations for all types of functionalities a textual model editor should be capable of e.g., syntax highlighting, reference resolving, or tooltips and hints.

Several different approaches were already implemented for combining textual and graphical modeling. These approaches mostly differ in the way how the textual and graphical models are synchronized with each other, in which way the graphical and textual editors are combined, and how the users are allowed to interact with each of the models or representations of the model.

When combining textual and graphical editors, the two editors have to synchronize the textual representation, the graphical representation and the underlying models [vRWS⁺13]. The most straightforward approach is to only synchronize the graphical and

textual representations of the model in one way e.g., to generate a graphical representation of the textually defined model. In this scenario, the users can only interact with the textual model and the graphical representation has to be generated when the textual model changes. The underlying AST of the textual model is converted into a graphical model based on predefined rules, e.g., how each element of the textual model should be represented in the graphical model. The graphical model is then most presumably rendered using an auto layout, as users cannot directly modify the elements on the graphical view, and some predefined rules have to decide where each element should be displayed on the view.

Another approach of combining textual and graphical modeling is to synchronize the graphical and textual representations of the model not only in one way but both ways. This means that changes made in the textual representation must be synchronized to the graphical representation and vice versa. This synchronization is the main reason where most problems of combining textual and graphical modeling arise [AG13]. The two representations must either use the same underlying model and AST or they have to be synchronized with each other every time a change occurs in either of the representations. Even if the graphical and textual views represent and modify the same AST, other challenges arise as the textual and graphical views need to display different aspects of the model. For example, it does not make sense to render a comment in the graphical view written in the model's text that only contains relevant information for the textual view. At the same time it does not make sense to store information which is only relevant for the graphical representation, i.e., a color of an element type of a model - that should not be modified by the user - in the model.

3.2 Textual-graphical Modeling Frameworks

The following sections present several different tools that combine textual and graphical modeling, summarize their benefits and drawbacks, and compare the introduced tools.

3.2.1 Excalibur workbench [RCG18]

The Excalibur workbench is an Xtext and Sirius based framework designed to model requirements engineering processes. The authors created their own suitable DSL called Messir and used Xtext to create the grammar of the language and generate the textual model editor. They call their approach a text-first approach meaning that the users are only allowed to edit the models in the text editor. Using Sirius, the authors created a graphical DSL that only allows to graphically display certain important aspects of the textual models. The displayed diagrams cannot be further modified by the users and are re-created each time the textual model changes. Therefore, this system is an example for one way synchronization of the textual and graphical representations.

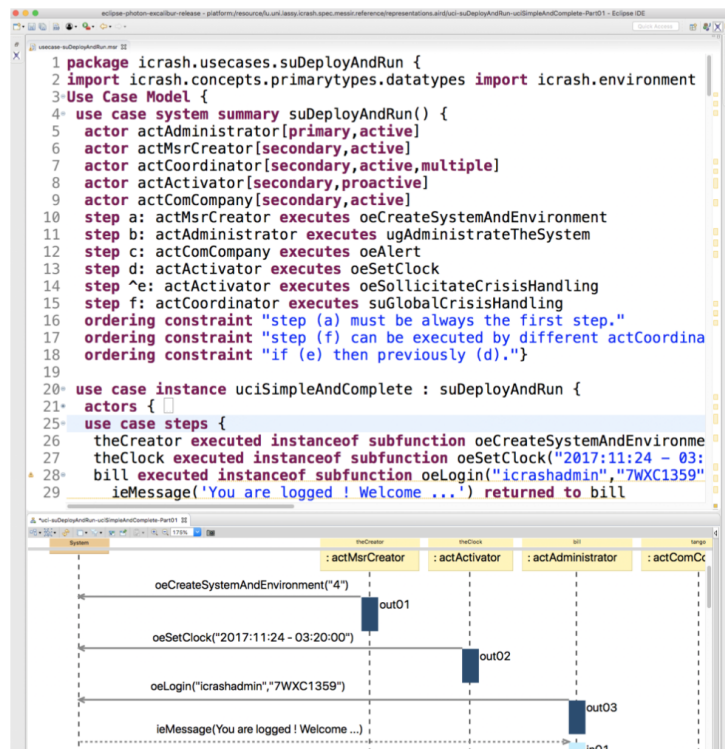


Figure 3.1: Excalibur: an Xtext-Sirius framework for read-only graphical representation. [RCG18]

3.2.2 Langium meets Sprotty [Pet22]

Petzold [Pet22] provides an example implementation of automatic diagram generation combining Langium with Sprotty, an open-source web-based diagramming framework. In previous work Sprotty was mainly used to display visualizations of models created with an Xtext-based language server. As Xtext is Java based and Sprotty is TypeScript-based, these solutions had to combine these two programming languages and maintain the connection between them, which meant increased development and maintenance overhead. Using Langium as a language server instead of Xtext provides a TypeScript-only based solution, which simplifies development and maintenance.

The proposed architecture creates a DSL using Langium and utilizes Langium as a language server for the textual model editor in VS Code. Furthermore, Petzold makes use of the `langium-sprotty`¹ extension that generates the diagram from the model's Langium-based AST, and also the `sprotty-vscode`² package that embeds Sprotty diagrams in VS Code webviews.

¹<https://github.com/langium/langium/tree/main/packages/langium-sprotty>

²<https://github.com/eclipse-sprotty/sprotty-vscode>

3. STATE OF THE ART

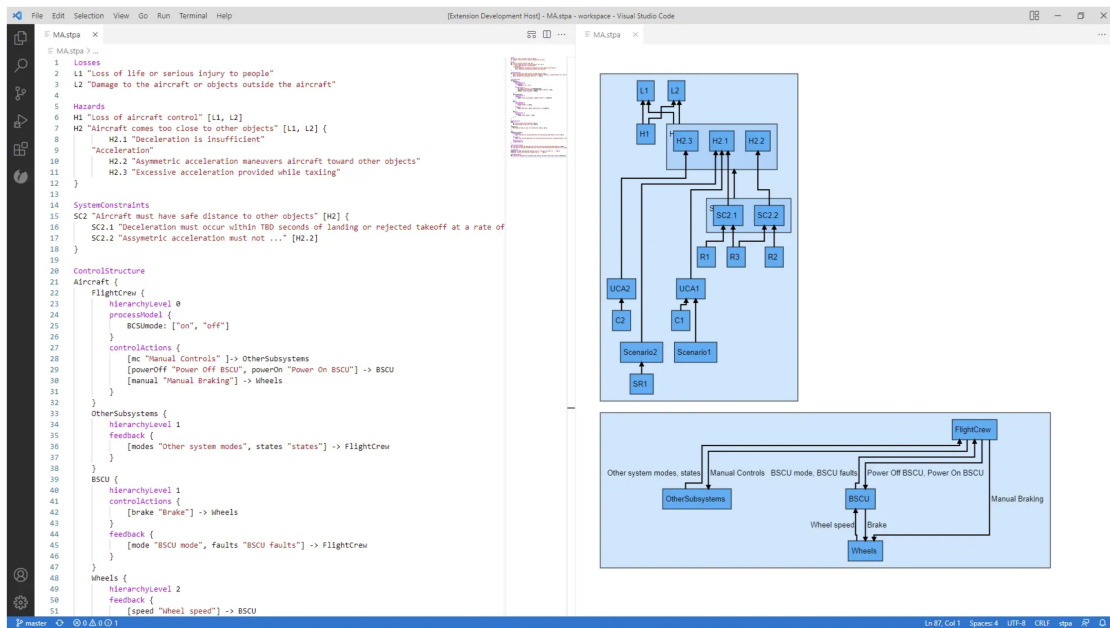


Figure 3.2: Langium meets Sprotty: graphical representation of a textual model using next-generation frameworks [Pet22]

The users of this framework can have the textual editor and the diagram representation open side-by-side as shown in Figure 3.2, and the graphical representation is automatically updated when the textual model gets modified. The diagram is read-only, however the framework provides some customization features in terms of color and visualizing sub-components in the model.

3.2.3 Xtext / Sirius - Integration [Obe17]

Obeo and Typefox present a framework utilizing Xtext and Sirius, a framework that allows to specify a graphical representation for an Ecore-based metamodel, created by Xtext. Three different approaches are presented by Obeo and Typefox: editing the same model both textually and graphically simultaneously; editing different aspects of a model textually and graphically, with references between the two representations and embedding an Xtext Editor into Sirius, to only allow editing smaller details of the model textually and the remaining details only graphically.

Figure 3.3 shows an example of an integrated Xtext editor in Sirius. The model editors are able to modify the core elements of the model graphically and to establish relationships between them, and can define rules for the elements in the integrated Xtext editor, which is easier to interact with for this use case than in a graphical representation for these rules. The text editor still provides syntax-aware text editing e.g., syntax highlighting and auto-completion.

Cooper et al. elaborate the problems recognized with the proposed approaches [CK19].

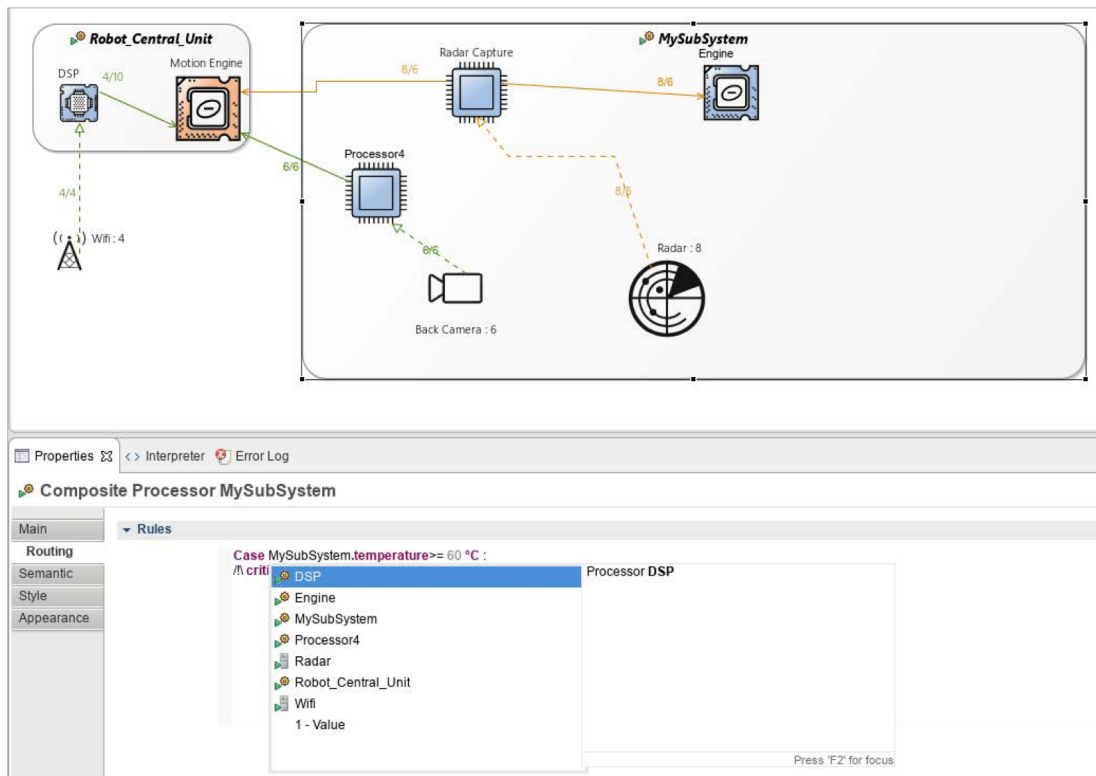


Figure 3.3: Xtext / Sirius - Integration: Embedding an Xtext Editor into Sirius [Obe17]

Firstly, editing the textual and graphical representations of the same model only synchronizes the views on save. This thesis tries to propose a solution to the simultaneous editing problem stated in Research Question 2 that allows editing in both of the representations in real-time. Secondly, this way of model representation can cause the diagram to get very complicated, as it tries to represent every element and every attribute of each element of the model. As for integrating the Xtext editor into Sirius, Cooper et al. see the biggest problem in the way the data for the Xtext editor is stored in the model. The textual data is stored as its string representation, which e.g., can cause problems when finding cross-references between models or renaming attributes.

3.2.4 Langium + Sirius Web = heart [Gir22]

Giraudet introduces an approach for a blended textual and graphical modeling tool combining Langium with Sirius Web, a web-based implementation of Sirius [Ecl4]. The proposed framework uses two different metamodels for the diagram and textual representation, and propagate the changes to each other when a modification on the model occurs. A Monaco editor is integrated into Sirius to make editing in the textual representation utilizing Langium as a language server possible, as shown in Figure 3.4.

Sirius displays a default representation of the model, the users are not able to change

3. STATE OF THE ART

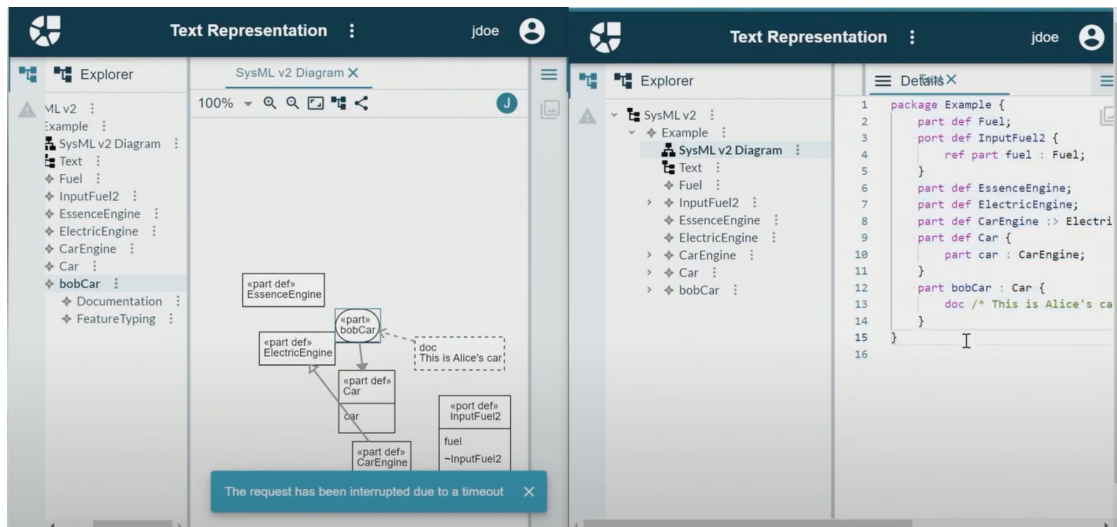


Figure 3.4: Langium + Sirius Web: simultaneously editing the same model graphically and textually [Gir22]

the visual properties of the diagram elements, only to edit the attributes and create and delete elements of the diagram that are also part of the textual representation. Sirius and Langium exchange information via a webhook in both directions, making on-the-fly changes possible on the same model in both representations simultaneously.

3.2.5 BIGER modeling tool [GB21]

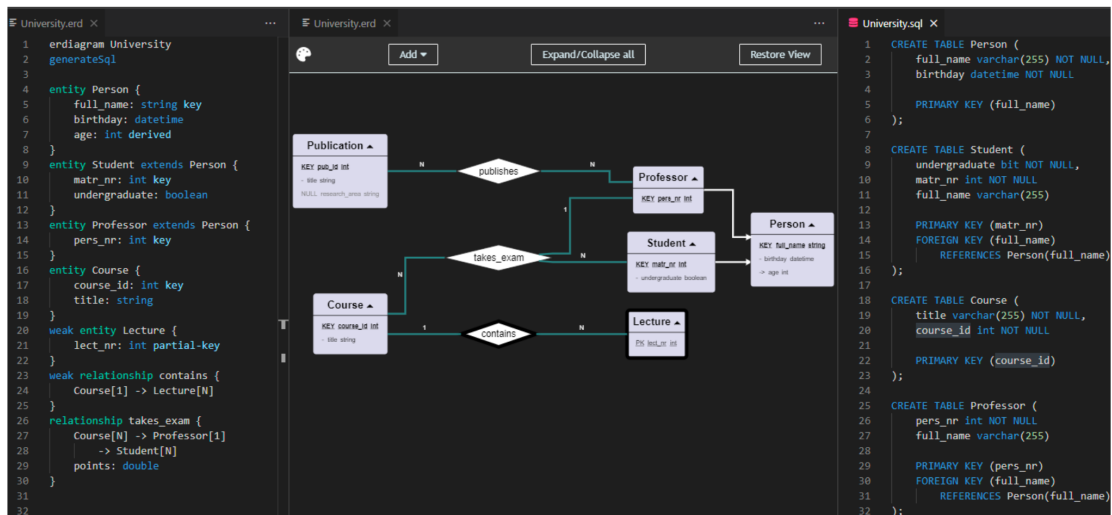


Figure 3.5: The BIGER modeling tool based on Xtext and Sprotty [GB21]

The BIGER modeling tool is a blended textual-graphical modeling framework based on

Xtext and Sprotty for creating Entity-Relationship (ER) models. The proposed tool is distributed as a VS Code extension³.

The synchronization of the two representations are based on the modification of the underlying textual model. Both the changes made in the text editor and in the graphical editor are done on the textual model. After the textual model is changed, the Sprotty diagram is re-generated using an auto layout. However, as this is a blended modeling tool, it is also possible to edit the model in the graphical view using the predefined actions e.g., adding a new node, or directly modifying the attributes of a node on the diagram. It is not possible to change the layout of the diagram.

3.2.6 A blended modeling framework based on Xtext and Papyrus [ACLP17] [AC21]

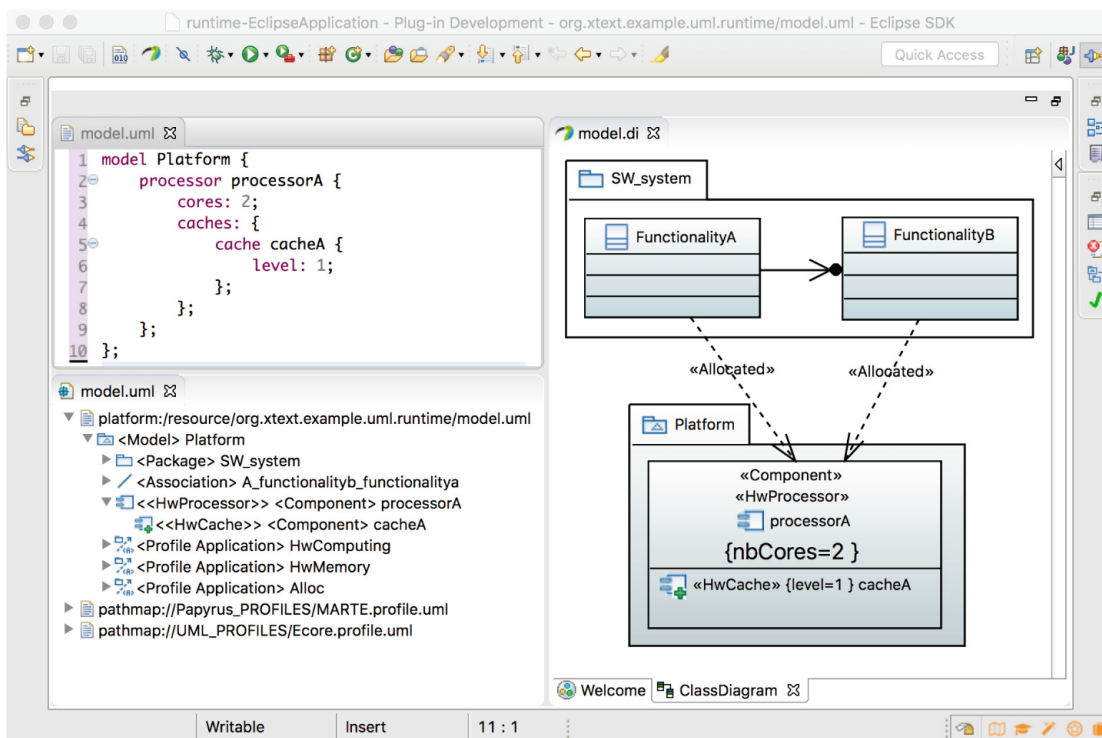


Figure 3.6: Blended modeling framework based on Xtext and Papyrus: simultaneously editing the same resource graphically and textually [ACLP17]

Adazzi et al. propose a blended textual-graphical UML modeling framework combining Xtext with Papyrus for UML [Eclb]. Their solution differs from previous approaches having separate abstract syntaxes for the graphical and textual representations, as Adazzi et al. use only one abstract syntax with two concrete syntaxes for one model.

³<https://marketplace.visualstudio.com/items?itemName=BIGModelingTools.erdiagram>

The proposed framework uses an in-place model transformation to solve the simultaneous editing problem. The changes on one concrete syntax are propagated in real-time to the other concrete syntax, which enables real-time simultaneous editing both textually and graphically.

3.3 Drawbacks Observed in the Listed Frameworks

The following section provides examples in the listed frameworks for the problems stated in Section 1.2 that this thesis aims to provide solutions for.

Problem 1: model service

None of the listed approaches utilize an external model service to handle the communication between the graphical and textual editors. For example, the Langium + Sirius Web framework [Gir22] explicitly states the use of a webhook in Langium and Sirius that gets called when a change in one of the representations occurs. The framework proposed in this thesis aims to provide a model service API that allows textual and graphical editors to manage and manipulate the underlying abstract syntax of the model jointly.

Problem 2: modification model

The proposed Xtext-Sirius framework in [Obe17] explicitly states that the modifications made in one representation are only delegated to the other representation when the model gets saved to the file system. This thesis aims to provide a solution to this problem by implementing a modification model that synchronizes the textual and graphical representations in real-time when a change occurs.

Problem 3: cross-references

Cooper et al. states [CK19] that implementing the Xtext-Sirius editor integration proposed by [Obe17] with enabling Xtext for editing small parts of the model in the graphical view, can cause problems with cross-referencing attributes as the text is only persisted as a string attribute in the model. This way the cross-reference resolution mechanism might not notice elements of the model that are only defined in this string attribute. This thesis aims to solve the problem of resolving cross-references correctly in both representations with the proposed Langium-GLSP framework.

Problem 4: non-semantic information

The video demonstration of the Langium + Sirius Web framework [Gir22] shows that a modification in the diagram triggers a re-serialization of the model as the format of the textual representation changes after editing in the graphical representation. This references the non-semantic information problem of the expected results that this thesis tries to provide a solution for.

	Used technology for textual modeling	Used technology for graphical modeling	Editing possible	Synchronization mechanism
Excalibur workbench [RCG18]	Xtext	Sirius	textual only	one way: Xtext -> Sirius only selected elements displayed
Langium meets Sprotty [Pet22]	Langium	Sprotty	textual only	one way: Langium -> Sprotty
Xtext / Sirius - Integration [Obe17]	Xtext	Sirius	textual + graphical	both ways changes only synchronized on save
Langium + Sirius Web = heart [Gir22]	Langium	Sirius Web	textual + graphical	both ways changes synchronized instantly
BIGER [GB21]	Xtext	Sprotty	textual + graphical	both ways changes synchronized instantly
Xtext / Papyrus framework [ACLP17]	Xtext	Papyrus	textual + graphical + tree view	both ways changes synchronized instantly

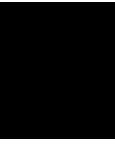
Table 3.1: Comparison of the listed frameworks.

The expected results of this thesis go beyond the state of the art as previous work does not comprehend combining textual and graphical modeling using Langium and GLSP. The proposed framework in this thesis also tries to provide solutions to all of the problems listed above.

3.4 Summary and Comparison of the Listed Frameworks

This chapter provided an overview on the state of the art blended modeling tools. Table 3.1 compares the most important aspects of the introduced frameworks. The listed frameworks differ in the technologies they use to combine textual and graphical modeling,

the modifications they allow in the model and how these modifications are propagated between the two frameworks they combine. It is important to state that even though some of the frameworks provide blended textual-graphical modeling, none of the frameworks allow to modify the structure of the diagram i.e., the size of the diagram elements or how they are arranged. The implemented prototype for this thesis aims to provide this functionality, too, which the following chapter further elaborates.



Concept

In this chapter, the concept for creating the prototype of the novel graphical-textual modeling framework is discussed. It outlines the prototype's requirements and details its planned architecture. Furthermore, this chapter also provides an overview of the model service and provides solution concepts for the stated requirements of the prototype.

4.1 Main Idea and General Approach

The main goal of this thesis is to create a prototype of a graphical-textual modeling framework that allows users to interact with the same model both graphically and textually simultaneously. Simultaneously means, that the same model can be opened in both the textual editor and the graphical editor, and changes made in one editor are automatically synchronized to the other. The goal of this thesis is to use "next generation frameworks" to implement the prototype. At the time of writing this thesis Langium and GLSP are considered as next generation frameworks, and a combination of these frameworks for blended graphical-textual modeling does not yet exist, therefore these two frameworks will be used to develop the prototype.

As introduced in Chapter 2 Langium provides out of the box features to be used as a language server for textual modeling, and GLSP is a framework for graphical modeling, hence, Langium will be utilized for textual modeling and GLSP for graphical modeling in the proposed framework. To provide a connection between these two frameworks, and to ensure that the underlying modification models of both frameworks are synchronized with each other, a third component, a model server is to be implemented.

An example implementation of a model service [Crob] that connects a GLSP server with Langium already exists, developed by CrossBreeze [Cra]. To facilitate the conceptualization and development process of this thesis, this model service will be used as a starting

point for the proposed framework's model server, and will be further implemented and extended to suit the needs of blended textual-graphical modeling.

4.1.1 General approach of the prototype implementation

To implement a blended textual-graphical modeling framework that meets all the requirements described below and to provide a tool that is ready-to-use and to demonstrate the required functionalities, an example modeling language must be chosen. The GLSP framework provides an example implementation of a TypeScript-based server and client for graphical modeling in the Workflow language [Foud]. Therefore - as at the moment no other TypeScript-based server-client example projects are available - the Workflow language based GLSP framework is chosen as a base for the prototype. This enables to concentrate on the core of the framework, which is the synchronization of graphical and textual modeling, to enable simultaneous editing in both of the representations.

To match the base GLSP project, the language server will be also tailored to the Workflow language. A suitable grammar must be created and the language server services e.g., syntax highlighting, model validation etc. will be adjusted to provide the required textual modeling features listed below.

The model service will also be implemented to suit the needs of the Workflow language based prototype, however, the general concepts and features of the model service should be implemented so that it can be used with arbitrary textual and graphical clients. This is feasible, as synchronization of the models between the different services should not be as language-specific as e.g., a diagram representation of a model or explicit validation rules on the language server are.

To validate the implemented prototype, the general concepts and features of blended modeling will be implemented and adjusted to provide blended graphical-textual modeling on two BIGUML [Bor] [MB23a] use cases: the class diagram and the package diagram. The GLSP server, the language server, and the model server will be tailored to match the required needs of the framework listed below. language-specific features that were only viable for the Workflow language will not be implemented, features that need to be adjusted will be documented and adopted and features that work as is will be inherited.

4.2 Requirements for the Blended Textual-Graphical Modeling Framework

This section elaborates the general requirements of a blended textual-graphical modeling framework. These requirements are based on the advantages and drawbacks of the modeling frameworks introduced in Chapter 3, and try to list all the necessary features that a blended textual-graphical framework should have, to make simultaneous graphical and textual modeling possible. These requirements are to be realized by the proposed Langium-GLSP prototype framework.

4.2.1 Requirements for the language server and text editor

This section explains the requirements for the language server, which must be met to enable textual modeling capabilities for the blended framework. The requirements only consider features of the Langium framework that need to be explicitly adapted for the graphical-textual modeling framework.

- **Grammar:** the grammar of the modeling language must be implemented in the Langium grammar language [Typec]. The grammar defines the textual syntax rules of the language and infers the types and interfaces of the TypeScript-based AST, that Langium auto-generates. For the prototype, the Workflow language grammar must be defined and implemented.
- **Syntax highlighting:** the textual editor must feature syntax highlighting, meaning that different grammar components are highlighted with different colors in the text editor.
- **Validation:** the textual model must be validated to ensure that it corresponds to the defined grammar and also to the specialities of the given modeling language. These language-specific requirements on the model must be explicitly defined as validation rules e.g., that a given name must not be used twice in a model or that a given element can only have edges to a specific type of element.
- **Error handling:** if there are any grammatical or semantic errors in the textual model, the editor must display them after unsuccessful validation. It must be clear for the user which component causes the error and why. For some language-specific errors that can be automatically fixed, a mechanism should also be implemented that fixes this error on the model.
- **Serialization:** the language server must provide a serializer, that converts the AST of the model to the text that is displayed in the text editor i.e., the actual model. The serializer must generate a syntactically correct textual model. It must also pay attention to the existing white spaces and comment of the model, that must be preserved during serialization.
- **Cross-references:** the language server must be able to resolve cross-references in the model, and correctly reference the element in question or display an error if the referenced element is not found.
- **Workspace and scope management:** the language server must define the workspace of the model, i.e., where the models should be loaded from for the current project. The cross-references are only resolved within the defined workspace. The server must also provide scope management i.e., to define which symbols are accessible from which models for cross-references.

- **Model synchronization:** the language server must be able to rebuild the underlying modification model i.e., the `LangiumDocument` after a change occurs in the graphical modeling editor and the change is delegated to the language server via the model server. The language server must also be able to delegate changes made in the textual model to the model server that will trigger the update of the graphical view.

4.2.2 Requirements for the graphical language server and client

This section states the requirements for the graphical modeling part of the Langium-GLSP framework. These features must be implemented in order to provide a blended textual-graphical modeling environment, where simultaneous editing of the same model is possible both textually and graphically.

- **Graphical model:** a graphical model must be created that represents the underlying AST generated by the language server from the textual model. The graphical model must be displayed in the web view on the client.
- **Model editing:** the GLSP framework must allow editing of the graphical model, including editing attributes of the elements, creating new elements, creating cross-references (e.g., by creating edges) between the elements, and also to modify the graphical representation of the elements i.e., the size and position of each element.
- **Validation:** the GLSP framework must be able to validate the model and to display errors on the graphical model if present.
- **Error handling:** the framework must allow users to fix errors and provide ways to modify the underlying model of the diagram to remove the error, if the error can be fixed in the graphical representation.
- **Model synchronization:** the GLSP framework must provide a solution for model synchronization. The framework must be able to send the modifications done in the graphical editor to the model service, that will delegate these changes to the language server. The GLSP framework must also be able to apply the changes coming from the language server delegated by the model server, and re-generate the graphical model according to these changes.

4.2.3 Requirements for the model server

The following requirements must be satisfied by the model server to ensure proper synchronization of the language server and GLSP server, and to make sure that the currently edited model is always up-to-date on both ends.

- **Access:** the model server must provide access to the models in the current workspace. An arbitrary client that wants to interact with the model must be able

to maintain connection with the model server, to open, request, update, and save a model.

- **Model provision:** the model server must be able to provide the current version of the requested model including every change that a client might have made to it during the current life-cycle. It means, that a client can have already made changes to a model without saving it to disk, when another client is requesting this model. Then, the model server must provide the current version of this model.
- **Model synchronization:** the model server must update its internal state of a model when a client requests an update on this model. If another client also has this model opened, the model server must notify this client and send over the new version of the model to make sure that all the clients are synchronized to the newest version of the model.

4.3 Framework Architecture

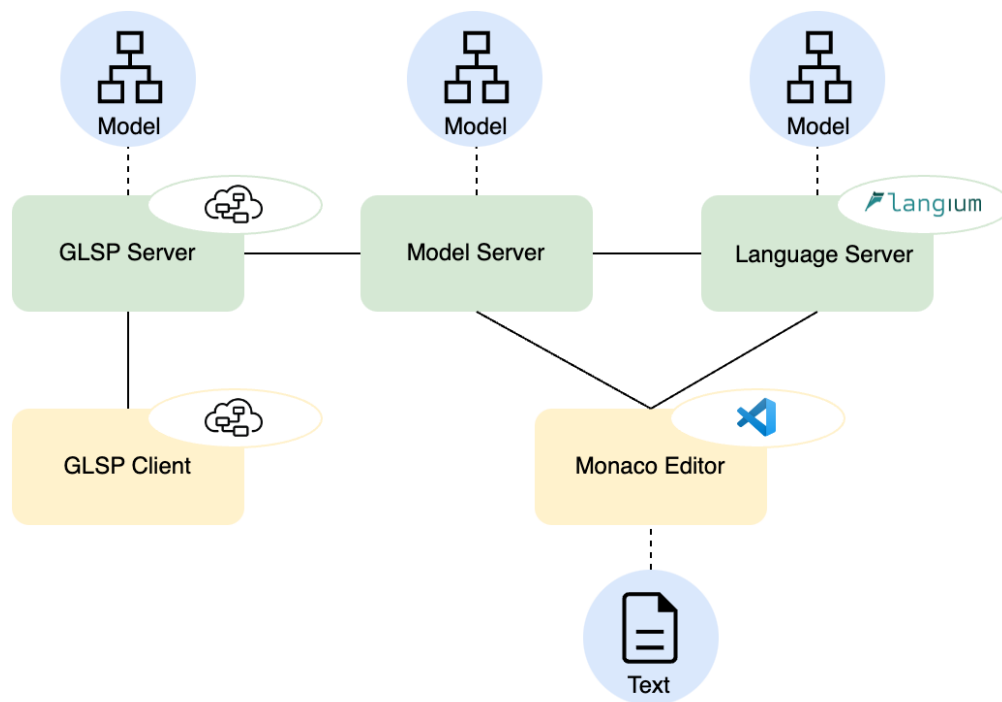


Figure 4.1: Architecture of the Langium-GLSP blended modeling framework

This section outlines an architecture that the implemented framework should be based on. The proposed architecture is displayed in Figure 4.1.

The framework will be implemented as a VS Code extension [Mica]. The Langium framework and the GLSP framework already provide glue code on how to integrate them as VS Code extensions, therefore the two frameworks must yet to be combined with each other utilizing the model server. This way, the Langium-GLSP framework will be able to run integrated into the VS Code editor, utilizing Monaco [Micd], the default text editor of VS Code as the framework's text editor, and the GLSP client's VS Code web view as the framework's graphical model editor.

The model server is the central connecting component of the framework. Both the GLSP server and the Langium language server must communicate with it to ensure model synchronization between them. The model server must also be able to communicate with the text editor itself, to directly delegate changes to it coming from the GLSP server and to listen to changes to delegate to the GLSP server.

Starting from the *text editor*, its internal state is the textual model i.e., the concrete syntax of the model. The Langium language server acts as a language server for this text editor, providing and enabling all the defined requirements for it e.g., syntax highlighting, validation, error handling. The users of the framework can directly interact with the model making changes to the textual concrete syntax of the model in this editor.

The *language server* makes textual modeling available for the Monaco editor. It listens to changes in the opened text document and rebuilds the corresponding LangiumDocument including the AST of the model as described in Section 2.3 every time a change in the text file occurs.

The *model server* listens to changes on the text editor, the language server, and the GLSP server. This is necessary as the changes made in the text editor are delegated in the form of the AST and textual concrete syntax to the GLSP server, and the GLSP server delegates the changes to the model server in textual form. Therefore, the model server has to maintain both the textual concrete syntax and the AST in its internal state.

The *GLSP server* delegates the changes both to the *GLSP client* and the model server. The GLSP client is responsible for rendering the graphical model and providing the graphical model editor in a VS Code web view.

4.4 Model Server Concept

The model server must be designed and developed in accordance with the aims and expected results of this thesis as stated in Section 1.2 and also with the requirements stated in this chapter's Section 4.2.

The initial model server provided by CrossBreeze [Crob] provides two possibilities for operation: to work as a server receiving and sending requests to and from the clients via JSON-RPC and to be integrated with the language server as a custom service. In the prototype the model server will be used as a shared language server service, that both the Langium language server and the GLSP server have access to.

4.4.1 Basic model operations

The model server has to implement the following unambiguous features to provide a solution for Research Question 1 of the expected results.

- **open:** the model server has to load the model from disk if it has not yet been opened with another client or it has to provide the current version of the model to the requesting client, even if it has already been open and modified by another client.
- **request:** the model server has to provide the current version of the model from its internal state if it has already been opened either from the requesting client or another client, otherwise it has to open the model and load it from the disk.
- **save:** the model server has to be able to save the current version of the model to disk on client request.
- **close:** the model server has to close the connection with a client on request and remove the model from its internal state if none of the other clients have it open.
- **update:** the model server has to provide a way for clients to actively request the server to update a model, and it also has to listen to updates on clients which do not provide methods to actively send updates to the model server.

4.4.2 Model server internal state and update mechanism

To provide an answer to the Research Questions 1, 2, and 4 i.e., to allow for simultaneous modifications on the model coming from the language server and the GLSP server preserving non-semantic information in the model's textual concrete syntax, the following possible concepts for the model server are established.

1. Storing only the AST in the model server's internal state with incremental updates

The model server must keep track of the current version of the model. Therefore, it is necessary that after every model update coming from the client, the internal state is updated. The best way to keep track of the model updates would be to hold the AST of the model in the model server's internal state and successively update it on every update request. This way every client would send a request on which operation they would like to perform on the AST, e.g., to add or delete a node or to modify an attribute on a node. This way, the AST would not be replaced on each update with a newer version, but would be updated incrementally according to the client's requests.

After every update, the GLSP server would receive the newer updated AST, to regenerate the graphical model from it. As the Langium framework does not foresee direct updates on the AST, the model server would have to directly replace the textual concrete syntax in the Monaco editor. For this, the model server would have to know which elements of the

semantic model have changed, and must update only these exact elements in the concrete textual syntax. Then, this would trigger Langium to re-build the `LangiumDocument` and re-generate its AST. This concept would provide solutions for Research Questions 1, 2, and 4. However, this concept also foresees purely incremental updates on the AST, and at this moment this is not possible using Langium. After making changes in the text editor, Langium always re-builds the whole `LangiumDocument`, and only provides the complete AST afterwards, without explicitly stating which changes occurred in the AST. Also, incrementally updating the AST on the model service would mean that either it has to provide operations on the AST and the textual syntax that are language-specific. As the model service should provide a generic implementation, this concept is not appropriate to solve the problem of the model service.

2. Storing only the AST in the model server's internal state with full updates

Instead of incremental AST updates, the whole AST could also be replaced on every update. This way, the clients would have to deal with updating the semantic model, and sending it to the model server as a newer version. After a model server received an update, it would have to either serialize the AST and replace the textual concrete syntax of the model in the text editor with the newly serialized version if the update was coming from the GLSP server, or, it would have to forward the AST to the GLSP server if it was coming from the language server.

This concept would provide solutions to Research Questions 1 and 2, but not to the problem of non-semantic information stated in Research Question 4. The previous concept would allow for a solution for updating the textual concrete syntax to only update elements that were actively changed by the clients and preserving the other parts of the textual model including all the white spaces and comments. However, if the whole AST was updated, it would either mean that the model server would have to re-serialize the whole model, which would update the previous version of the textual concrete syntax and delete all comments and extra white spaces, or the new and old versions of the AST would need to be compared to find out which exact elements changed and only update these in the textual concrete syntax as in the previous concept. As this task is proven to be non-trivial [CLG⁺10], this concept is also not feasible for the scope of this thesis.

3. Hybrid AST-text model server internal state

To overcome the hurdles discussed in Points 1. and 2. the following blended concept is developed for simultaneous updates on the model server's internal state, which involves both the textual concrete syntax of the model and the AST.

- **Update from the GLSP server:** the update request for the model server could either be in form of an incremental AST update or an update on the textual concrete syntax. With the first option, the problem of the model server having to provide updates on the textual concrete syntax would arise, as the model server can only send updates in textual form to the text editor as discussed before. Therefore, it is evident that the model server should store the model as its textual concrete

syntax in its internal state, and the GLSP server would only send updates on this textual syntax. To avoid the model server having to re-serialize the model and compare the changes of the old and new textual syntax to maintain non-semantic information, the re-serialization is sourced out to be done on the GLSP server. Hence, an update coming from the GLSP server is a full update on the textual concrete syntax. The model server then updates its internal state with the new textual syntax and sends this update to the text editor, which triggers the language server to re-build the document of the language server's internal state and update the AST.

- **Update from the language server:** as mentioned before, Langium re-generates the `LangiumDocument` including the model's AST on every update made in the text editor. As the language server is not able to directly send updates to the model server, the model server must listen to the `DocumentBuilder` service of Langium and update its internal state if a new `LangiumDocument` has been built. The `LangiumDocument` contains both the textual concrete syntax of the model and its AST. Hence, the model server's internal state is updated with the textual concrete syntax, and the AST (and the new textual concrete syntax) are sent to the GLSP server, which then re-generates the graphical model and triggers the client to re-render the web view.

With this concept all of Research Questions 1, 2 and 4 can be solved. With model updates enabled in both directions, joint modifications of the model's AST with both textual and graphical editors are possible, providing a solution concept for Research Question 1. The concept also attempts to provide a solution for Research Question 2, as it clearly defines that the underlying modification model of both the GLSP server and the language server is the textual concrete syntax. The concept also considers Research Question 4, as maintaining non-semantic information in the textual concrete syntax should be provided as explained above.

4.5 Solution Concepts for the Requirements

This section provides solution concepts for the requirements of the textual modeling, graphical modeling and model server for the blended textual-graphical modeling framework listed in Section 4.2. It also elaborates, whether the concepts are language-specific i.e., developed specifically for the Workflow language, or generic i.e., could also be used for an arbitrary language.

Textual modeling

- **Grammar** - language-specific
The grammar will be specifically developed to match the Workflow language syntax and semantics of the provided GLSP server and client projects. This must be

re-written for every other language that the framework will be used on. The grammar will be designed so that the model can be split into two different files: one containing all the elements of the model and another containing the attributes of these elements that are only needed for the graphical representation, in case of the Workflow language these are size and position.

- **Syntax highlighting** - generic
The auto generated syntax highlighting rules will be used by Langium.
- **Validation** - language-specific
Workflow language-specific validation rules must be developed, which are not likely to be reused for other languages.
- **Error handling** - generic/language-specific
The generic solution concept for error handling foresees that all the errors are labeled and provide an understandable explanation to the user. If an error is auto-solvable i.e., if a cross-reference can not be resolved, but the missing node could already be created with the information that is already provided in the model, than the language server should provide a code action to do so. The actual error messages, and implementation of the code actions are Workflow language-specific.
- **Serialization** - generic/language-specific
The generic solution concept for the AST serialization is to not re-serialize the whole AST, but only the changed elements, and replace them in the current serialized text (the textual concrete syntax). The serializer service therefore has to provide functions that only serialize one element of the AST. This is necessary to maintain the non-semantic elements of the textual concrete syntax. If the whole AST would be re-serialized and the textual concrete syntax replaced on every model update, the non-semantic information would be lost. The actual implementation of the serialized elements are Workflow language-specific.
- **Cross-references** - generic
The generic cross-reference resolution mechanism of Langium will be used.
- **Workspace and scope management** - generic
The scope will be reduced to only allow cross-references within files belonging to the same model, as it is not common that elements of a graphical model reference other elements in another graphical model. If the same model consists of multiple files, cross-references can still be declared in each of them referencing to an element declared in another file belonging to the model.
- **Model synchronization** - generic
As explained in the solution concept for the model server, the model server will be set up to listen to changes in the `DocumentBuilder` service of Langium. On each change in the textual model, the `DocumentBuilder` service rebuilds the `LangiumDocument`, and sends an update to the model server. The model server

then updates its internal state with the received textual concrete syntax and sends the new AST to the GLSP server or any other arbitrary client listening for changes on the model.

Graphical modeling

- **Graphical model** - language-specific
The graphical model is specifically tailored to match the GLSP client's graphical elements. Each element has its own type and graphical properties that is specific to the Workflow language.
- **Model editing** - generic/language-specific
The generic solution concept for model editing is that every possible element of the language must be able to be created, updated, and deleted via the GLSP framework, according to the semantics of the language. The elements have a predefined language-specific layout, that is defined by the graphical model, but the elements must be resizeable and the users must be able to re-position them.
- **Validation** - language-specific
The GLSP server must be able to validate the model according to the language's semantics.
- **Error handling** - generic/language-specific
The generic concept for error handling in the graphical model editor is to not allow any actions to be made on the graphical editor which would contradict the language's semantic, e.g., only allow to draw an edge between specific types of elements. If an erroneous modification on the model was made in the text editor, the model server would still send the error prone AST to the GLSP server in predefined language-specific scenarios, to make the GLSP client render the error to the user and to provide actions to solve the error. In the case of the Workflow language, missing references of nodes will be added to the graphical model as missing nodes, and the GLSP client will render them accordingly. The GLSP client will then provide possible actions to the user to create the missing node according to the language's specification on which type of nodes are allowed in which scenarios. If other types of errors occur in the text editor, the model server will not delegate the updated model to the GLSP server, only after the error was resolved in the text editor. To prevent overwriting the erroneous textual syntax in the text editor with updates from the GLSP editor, the model server will not update its internal state until the current `LangiumDocument` of internal state contains errors.
- **Model synchronization** - generic/language-specific
The generic concept for model synchronization is the following: on the one hand, the GLSP server will be listening to updates on the model server, and update its internal state i.e., the AST and the textual syntax of the model. After the internal state has been updated, the graphical model will be re-generated from the new AST

and the GLSP client will be triggered to update its graphical view. On the other hand, if the update operation is coming from the GLSP client, it will trigger the GLSP server to update the textual syntax of the model directly. This will be done by utilizing the serializer of the language server, to only re-serialize the elements of the model that were updated in this operation. The GLSP server then updates only the elements in the textual syntax that were re-serialized during the operation. This is possible as the AST contains the exact positions of each element in the textual syntax. This way, the non-semantic information in the textual syntax will be maintained. After the GLSP server is done updating the textual syntax it sends the updated textual syntax to the model server which provides the updated AST as a response to the update request and with this the GLSP server also updates its own graphical model, which again, triggers the GLSP client to update the graphical view.

Model server

- **Access** - generic
Access to the model will be provided by the basic model operations as described in Section 4.4.1.
- **Model provision** - generic
The model server will receive all updates on the model coming from either the language server or the GLSP server (or another arbitrary client, that is connected to the model server similarly to the GLSP server) and store it in its internal state. Therefore, if a new client requests the model, always the newest version will be provided by the model server, even if the model has not yet been saved to the disk by the other clients.
- **Model synchronization** - generic
The model synchronization will be implemented as discussed in Section 4.4.2 and this section's textual modeling and graphical modeling synchronization concepts. To avoid infinite update loops i.e., a GLSP server update triggering a text editor update and language server update, which normally would also again trigger a GLSP server update, and vice versa, the models will be versioned on the model server. The model server will track which version of the model each client is currently operating on and increase the client's model version if an update is coming from this client. This way, the model server can avoid sending the same update to the client that was originating from this exact client and only send the updates to the other clients if their version of the model is behind the newest updated one.

4.6 Summary

This chapter provided the basic requirements of a blended textual-graphical modeling framework that utilizes a model server to jointly update the underlying model. Furthermore, concept solutions were developed for the language server, the GLSP server, and the model server that fulfil the stated requirements. The concept solutions were tailored to provide a generic solution that can be applied for an arbitrary DSL where possible, and were fine-tuned for the Workflow language where necessary. The following chapter discusses how these concepts were implemented to provide a blended textual-graphical modeling framework for the Workflow language.

Prototype Implementation

This chapter elaborates the approach of the prototype implementation for the blended textual-graphical modeling framework. The implemented prototype has three major components, corresponding to the planned architecture of the framework, as introduced in the previous chapter: the language server, the model server, and the GLSP server. These components are implemented simultaneously as they all depend on each other to achieve the goal of blended textual-graphical modeling. Where applicable, the implementation aims for a general solution in terms of compatibility with arbitrary DSLs, however the prototype is tailored to support the Workflow language.

5.1 Implementation of the Language Server

The core of the textual modeling in the blended framework is the Langium [Typa] language server. Langium provides the possibility to create domain-specific languages together with an out of the box TypeScript-based language server with generic features for textual modeling, such as syntax highlighting, model validation, hints, serialization, cross-reference resolution, and scope management.

Langium provides a Yeoman generator [Yeo], that automatically generates a Langium language server project based on their Hello World example language, creating a general project structure with example implementations of some custom services. This example project will be modified and expanded to act as a language server for the Workflow language.

5.1.1 Grammar

The core of the Langium language server is the grammar of the given DSL the language server should be built for. The grammar must be defined as a `.langium` file according to the Langium grammar language [Typec]. Langium then generates the TypeScript

interfaces of the AST based on this grammar, which is also the base for the custom services, as they interact with these generated TypeScript interfaces. Therefore, the first step of the language server implementation is to define the grammar of the Workflow language.

The semantics of the Workflow language are gathered from the example `node.js` and TypeScript-based implementations of the GLSP client and server projects, and can be defined as follows.

Workflow semantics

A model written in the Workflow language represents a workflow that can be split into smaller tasks and sequential dependencies between these tasks, and can contain the following elements: nodes as a `TaskNode`, `ActivityNode` or `Category`, and edges.

A `TaskNode` can either be a manual task or an automated task. Both types of tasks have an ID, a label as their names, and a duration.

An `ActivityNode` can either be a decision node, a fork node, a join node, or a merge node. All activity nodes have an ID. A decision node implies that it can be decided which ones from its outgoing tasks might be done, with a merge node as its counterpart. A fork on the other hand implies that all of its outgoing tasks need to be done, before the result can be joined again with a join node.

A `Category` can have all types of nodes and edges as their children and acts as a group for them. It also has an ID and a label parameter.

An edge can either be a plain directed `Edge` connecting two nodes, meaning that a task has to be finished before the next task can be started, or a `WeightedEdge`, which can have a decision node as its source, and has a `probability` parameter, which indicates the probability that the task on the outgoing end of the edge will be executed.

Workflow syntax

```
1 grammar Workflow
2
3 entry Model:
4     (nodes+=Node | edges+=Edge | metaInfos+=MetaInfo)*;
5
6 Node:
7     (TaskNode | Category | ActivityNode);
8
9 MetaInfo:
10    (Size | Position);
11
12 TaskNode:
13    'TaskNode' name=ID label=STRING? (expanded?='expanded')?
14    duration=NUMBER? taskType=TaskType?';';
15
16 Category:
17    'Category' name=ID label=STRING? ('{' children=Model '}')? ';';
18
```

```

19 ActivityNode:
20     'ActivityNode' name=ID (nodeType=NodeType)?';';
21
22 TaskType returns string:
23     'automated' | 'manual';
24
25 NodeType returns string:
26     'decision' | 'fork' | 'join' | 'merge';
27
28 Weight returns string:
29     'low' | 'medium' | 'high';
30
31 Edge:
32     (source=[Node:ID] '->' (target=[Node:ID])? ';') |
33     ((source=[Node:ID])? '->' target=[Node:ID] ';') |
34     WeightedEdge;
35
36 WeightedEdge:
37     (source=[Node:ID] '->' probability=Weight (target=[Node:ID])? ';') |
38     ((source=[Node:ID])? '->' probability=Weight target=[Node:ID] ';');
39
40 Size:
41     'Size' node=[Node:ID] width=NUMBER height=NUMBER ';';
42
43 Position:
44     'Position' node=[Node:ID] x=NUMBER y=NUMBER ';';
45
46 hidden terminal WS: /\s+//;
47 hidden terminal ID: /[_a-zA-Z@][\w_\-@\#]*/;
48 hidden terminal STRING: /"[^"]*"|'[^']*'/;
49 hidden terminal NUMBER returns number: /(-)?[0-9]+(\.[0-9]*)?//;
50
51 hidden terminal ML_COMMENT: /\/*[\s\S]*?\*\///;
52 hidden terminal SL_COMMENT: /\//\[^\n\r]*///;

```

Listing 5.1: Workflow grammar

Listing 5.1 shows the implementation of the Workflow grammar using the Langium grammar language corresponding to the language’s semantics. The grammar is built on the following simple pattern: `"typeOfNode ID attribute1 attribute2;"` for nodes and `"node1 -> node2;"` for edges. An example of a model written corresponding to this grammar is shown in Figure 1.1.

The grammar also has additional elements not disclosed in the semantics, that are needed to enable blended modeling. As the solution concept states, the model will be split into two files: a `.wf` file for the core of the model including the definitions for all nodes and edges, and a `.wfd` file for these additional elements, which are only needed for the graphical representation of the model. For this reason, a model can also have `MetaInfo` elements, which can either be of type `Size` or `Position`. These elements must reference an already declared node, and store either the node’s `width` and `height` or its `x` and `y` coordinates, to define the node’s size and position on the graphical model.

The solution concept for graphical modeling foresees that missing nodes are also displayed on the diagram, and the framework must provide a possibility for the users to create

these missing nodes. To satisfy this requirement, the developed grammar also allows edges to either not have a source node reference or a target node reference provided. This means that edges are allowed to only be connected to one node, and on the other side of the edge a missing node will be displayed in the graphical model. Lines 32, 33 and 37, 38 of Listing 5.1 correspond to this concept in the grammar definition.

The concept of the Langium grammar language foresees that the defined nodes are referenced by their name attributes per default. Hence, in the Workflow grammar, the name attributes of the nodes are also used as their unique identifiers, and the additional `label` attribute is used as the actual name of the node.

The grammar also provides definitions for the type of each attribute as regular expression, which can either be of type `ID`, `STRING` or `NUMBER` as listed in Lines 47, 48, and 49 of Listing 5.1. Furthermore, the grammar also provides a possibility for comments that can either be single-line or multi-line comments, as defined in Lines 51 and 52 of Listing 5.1. The comments defined in the textual concrete syntax are part of the model's non-semantic information, which need to be preserved on updating the model.

Auto-generated resources

As mentioned before, Langium automatically generates the TypeScript interfaces of the AST based on this grammar, the grammar rules used for parsing and validating the model, and the syntax file of the defined language which is used by the language server for syntax highlighting. These files should not be manually adjusted and are re-generated if any change occurs in the grammar definition.

The implementation of the grammar therefore satisfies the *Grammar* and *Syntax highlighting* requirements of textual modeling.

5.1.2 Custom services

Langium provides general implementations for its language server services that handle the LSP requests coming from the text editor, such as code hints, code formatting, scope providing, or code validation. To implement the presented solution concepts for textual modeling in Section 4.5, some of these default implementations must be customized, and will therefore be overwritten or extended by custom implementations based on the Workflow language's syntax and semantics. These services are the following:

- **Validator:** this service registers custom validators, that are needed to ensure the Workflow language's semantics, and also to ensure that the correct element declarations are in the correct files.
- **Code action provider:** this service implements automatic code actions, that are available if an error occurs, to resolve this error. In the scope of the prototype, code actions will be used to generate missing nodes as an example for error handling.

- **Serializer:** this service serializes the model's AST, i.e., it generates the concrete textual syntax of the model given the AST. The solution concept also foresees, that this service provides serialization for the individual components of the Workflow model, in order to maintain non-semantic information in the textual concrete syntax of the model.
- **Workspace manager:** this service defines which files are included in the workspace that the language server is currently operating on. The default implementation - which foresees that every file of the open directory is part of the workspace - will only be extended by an event emitter, that tells the modeling framework when the language server is ready and the workspace is initialized, as the model server and GLSP server should only be started afterwards.
- **Scope provider:** this service defines which symbols are accessible from which models for cross-references. The scope will be reduced to only allow cross-references between files belonging to the same model.
- **Scope computation:** this service defines which elements of the document and how these elements are exported to be available for cross-references. The default scope computation will be extended to provide exports for the nested elements of the Workflow language.
- **Name provider:** this service provides methods that generate unique names for the model's components, based on their type, attributes and parents, if they have any. The default implementation is rewritten to match the Workflow language's syntax.

The custom implementations of the listed services are all registered in the language server's service module, to instruct the dependency injection component, that these custom implementations should be used by the language server per default. The implementation details of each of the custom services are discussed in the following sections.

5.1.3 Validator

Custom validators are needed to ensure that the model corresponds to the semantics of the DSL. There are several types of errors, that Langium will report:

- **Lexer and parser errors:** they occur during the parsing of the textual syntax and creation of the AST. These errors mean that the textual syntax does not correspond to the given grammar, and therefore, the AST cannot be generated correctly.
- **Linking errors:** they occur after the textual model has already been parsed and the AST has been generated, during cross-reference resolution, if a referenced node was not found by the language server.

- **Custom validation errors:** they occur when some of the custom validator rules were not satisfied by the model. Hence, a syntactically correct and parsed model (i.e., without any lexer and parser errors), might also have custom validation errors, if they do not satisfy the semantics of the language.

As the points stated above, a model that satisfies the grammar rules alone is not enough assurance, that the model also corresponds to the language's semantics. This observation also applies to the Workflow language, as some custom language semantic rules can only be ensured by custom validators. Therefore, the following custom validators are implemented on the language server.

- **No duplicate names:** the name property of each node (as they act as a unique identifier for this node) must be unique across all nodes of the model.
- **No duplicate edges:** an edge must not appear twice with the same source and target nodes. If the same edge could be declared twice in the model, they would still only be displayed as one edge on the diagram, as they would appear in the same position. This would cause uncertainty for the users as they would not know which edge they are currently modifying.
- **No duplicate position for a node:** a position of a node must only be declared once. If a node's position would be declared twice, the graphical model would either have to consider the first one or a random one, which would cause uncertainty in the graphical representation.
- **No duplicate size for a node:** similar to the position, the size of a node must also only be declared once, for the previous uncertainty reasons.
- **AST node in correct file:** the model must be split into two files, one containing the declarations for all the nodes and edges of the model and the other one containing the sizes and positions of the nodes declared in the previous file. As both files correspond to the same grammar, a custom validator is needed to check whether the elements are declared in the correct files.
- **Weighted edge starts from decision node:** according to the semantics of the Workflow language a weighted edge must only have a decision node as its source node.

A custom validator always has to be registered on a type of node of the AST, as this will be the scope of the validation. The duplicate checks and the AST node in correct file check are registered on the `Model` node type of the AST. This is the type of the root node of the AST and this node contains all the nodes of the model as their children. This is needed, as these validators must either iterate through all the nodes of the AST or all the nodes of a specific type of the AST, and only the `Model` root node contains all of these

nodes. The validator for the weighted edges is only registered on the `WeightedEdge` type, as it only needs to check one `WeightedEdge` at a time and validate its source node.

```

1 checkNoDuplicateNames(model: Model, accept: ValidationAcceptor): void {
2     const reported = new Set();
3     model.nodes.forEach((node) => {
4         if (reported.has(node.name)) {
5             accept("error",
6                 `Node has non-unique name '${node.name}'.`,
7                 { node: node, property: "name" });
8         }
9         reported.add(node.name);
10    });
11 }

```

Listing 5.2: Custom validator: no duplicate names

Listing 5.2 shows an example implementation of the *no duplicate names* custom validator. The input variable shows which node type the validator is operating on (in this case `Model`), and the function has to call the `ValidationAcceptor` input method if an error is detected on the input node. The validator iterates through all the nodes that must be checked for duplicates - in this case, all the nodes of the model must be checked - and collects every name into a set. If a currently expected node's name is already in the set, then the validator found a duplicate name and calls the `accept` method providing an adequate error message, which the text editor will display to the user, and the node and its property which caused the error to specify where the error message should be displayed.

The other custom validators listed above are implemented likewise. The implementations of the custom validators together with Langium's generic validators for lexer, parser and linking errors satisfy the *Validation* requirement for textual modeling.

5.1.4 Code action provider

A code action provides automated processes executed on the model's code i.e., in the textual concrete syntax. It can be declared, in which types of context the code action should be available for the user to execute, and for which part of the model the code action should be prompted for. Usually it is used if an error is detected on the model, and an automatic fix can be carried out to eliminate the error, but it could also be used in any other types of scenarios e.g., to delete unused code or to refactor code based on automated rules.

For the Workflow language, code actions will be used to provide automated error handling where possible. Corresponding to the requirements for error handling in textual modeling, code actions will be used to generate missing nodes, i.e., where a cross-reference to a node is already declared in the model - in the case of the Workflow language either as part

of an edge's source or target node, or as part of a size or position declaration - but the node itself is not yet declared. If this scenario arises, there is already enough information available for this undeclared node to be displayed on the graphical model as a missing node. As the graphical editor will also provide a possibility to create and declare new nodes from missing nodes that are already displayed as missing nodes, the textual model editor should also do so, and for this reason, a code action provider will be implemented.

To implement a code action, the `getCodeActions` method of the `CodeActionProvider` interface must be implemented by a custom code action service as shown in Listing 5.3. This method is called every time when an LSP request to show the Quick Fix menu [Mice] is triggered by the text editor. The method must return a list of code actions that are available from the current position of the code, where the method was triggered from. The document i.e., the built `LangiumDocument` containing the textual syntax and AST of the model and the params of the request containing the request context are handed over as input parameters.

```
1  getCodeActions(  
2    document: LangiumDocument,  
3    params: CodeActionParams,  
4    cancelToken?: CancellationToken  
5  ): MaybePromise<Array<Command | CodeAction> | undefined> {  
6    const codeActions: Array<CodeAction> = [];  
7    // only handle linking-errors  
8    if (  
9      params.context.diagnostics.length > 0 &&  
10     params.context.diagnostics.filter(  
11       (value) => value.data?.code === "linking-error"  
12     ).length === params.context.diagnostics.length  
13   ) {  
14     const missingNode = params.context.diagnostics[0];  
15     let uri = document.uri.toString();  
16  
17     // if size or position has missing reference,  
18     // the created node must be inserted  
19     // to the corresponding .wf file  
20     // and not the current .wfd file  
21     if (  
22       missingNode.data.containerType === Size ||  
23       missingNode.data.containerType === Position  
24     ) {  
25       uri = uri.slice(0, -1);  
26     }  
27  
28     // create decision node  
29     const createDecisionNode = this.createCodeAction(  
30       "Decision Node",  
31       this.createActivityNode("decision", document, missingNode),  
32       uri  
33     );  
34     codeActions.push(createDecisionNode);  
35  
36     // if error is caused by WeightedEdge 'source' property only allow  
37     // the creation of a decision node  
38     if (  
39       missingNode.data.containerType === WeightedEdge &&
```

```

40     missingNode.data.property === "source"
41   ) {
42     return codeActions;
43   }
44
45   [...]
46
47   // create category
48   const createCategory = this.createCodeAction(
49     "Category",
50     this.createCategory(document, missingNode),
51     uri
52   );
53   codeActions.push(createCategory);
54 }
55
56 return codeActions;
57 }
58
59 /**
60  * Creates an ActivityNode object with the given properties
61  * @returns The serialized ActivityNode
62  */
63 private createActivityNode(
64   type: NodeType,
65   document: LangiumDocument,
66   missingNode: Diagnostic
67 ): string {
68   const activityNode: ActivityNode = {
69     $container: document.parseResult.value as Model,
70     $type: "ActivityNode",
71     name:
72       missingNode.data.refText ??
73       findAvailableNodeName(document.parseResult.value as Model, "_an"),
74     nodeType: type,
75   };
76   return this.services.serializer.Serializer.serializeAstNode(activityNode);
77 }
78
79 [...]
80
81 /**
82  * Creates a CodeAction that attaches a serialized
83  * node to the end of the document with the given uri.
84  * @returns The serialized Category
85  */
86 private createCodeAction(
87   nodeType: string,
88   serializedNode: string,
89   uri: string
90 ): CodeAction {
91   const action: CodeAction = {
92     title: `Create missing ${nodeType}`,
93     kind: "quickfix",
94     edit: {
95       changes: {
96         [uri]: [
97           {
98             range: {
99               start: {
100                 character: Number.MAX_SAFE_INTEGER,
101                 line: Number.MAX_SAFE_INTEGER,

```

```
102         },
103         end: {
104             character: Number.MAX_SAFE_INTEGER,
105             line: Number.MAX_SAFE_INTEGER,
106         },
107     },
108     newText: `\n${serializedNode}\n`,
109 },
110 ],
111 },
112 },
113 };
114 return action;
115 }
```

Listing 5.3: Code action: create missing nodes

The listed implementation executes the following steps:

1. Check if there are errors in the requesting context and check if these errors are linking errors, which means that a referenced node cannot be found i.e., the declaration of this node is missing.
2. If there are errors in the requesting context and all of these errors are linking errors, find the name of the missing node that must be created. This is possible, as the requesting context only contains errors for the node which was selected when requesting the quick fix menu, and not for the whole document, therefore, the missing node name that is reported, will be the one that needs to be created to fix the given error. If any other errors are also present on the selected node (e.g., syntax errors), these must be fixed before the code action can be executed, and the `getCodeActions` method will return an empty list.
3. If the missing node was referenced from a size or position element, the node must not be created in the current `LangiumDocument`, but the corresponding `.wf` file of the model, therefore, the Uniform Resource Identifier (URI) of the document that will be edited by the code action must be modified.
4. Create the code actions for every possible node type. The code action does the following: it creates a node of the required type with the corresponding missing name, and serializes this node as shown in the `createActivityNode` function of Listing 5.3. The other types of nodes are created likewise. This serialized node then gets appended to the end of the textual syntax as shown in the `createNodeAction` function of Listing 5.3. This way, the previous textual syntax of the model will not be refactored, only extended, which provides a solution for the problem of maintaining non-semantic information in the textual syntax.
5. To assure correct semantics of the Workflow language, if the missing node is the source node of a `WeightedEdge`, the code action only allows the creation of decision nodes. Otherwise, the creation of every node type is allowed.

The implementation of the custom validator and the code action provider solves the *Error handling* requirement of textual modeling stated in Section 4.2 and also provides an answer for Research Question 4 of the problem statement of this thesis, as it maintains non-semantic information of the textual syntax on automated editing of the model.

5.1.5 Serializer

As Langium does not yet provide an out-of-the box serializer at the time of writing this thesis, a custom serializer that converts the AST of the model to its textual concrete syntax must be implemented. Corresponding to the *Serializer* requirement for textual modeling, the implementation of the custom serializer must fulfill following criteria.

- The serializer must generate a *syntactically correct output*, that can be validated and parsed using the implemented grammar for the Workflow language.
- The serializer must provide a *method to serialize the whole model* given its AST. The serializer must implement the `Serializer` interface of Langium, and it foresees to implement its `serialize` method, that serializes the model, even if this method will not be used directly from the implemented textual-graphical modeling framework, as serializing the whole model when an update on the AST happens and replacing the textual syntax with the newly serialized textual syntax would remove the non-semantic information of the textual model.
- The serializer must also provide methods that make it possible to only *serialize one concrete element of the model* i.e., one concrete node of the AST. This is necessary to maintain the non-semantic information of the textual syntax of the model, and these methods will be used by the blended modeling framework to update the textual syntax when necessary. The `serializeAstNode` method in Listing 5.4 serializes one concrete node of the AST. This method will be used instead of the `serialize` method in the implemented textual-graphical modeling framework.

```

1  serialize(root: Model): string {
2    const nodes = this.serializeNodes(root.nodes);
3    const edges = this.serializeEdges(root.edges);
4    const metaInfos = this.serializeMetaInfos(root.metaInfos);
5    const serializedModel = [nodes, edges, metaInfos]
6      .filter((part) => part.length > 0)
7      .join("\n");
8    return serializedModel;
9  }
10
11 serializeAstNode(astNode: AstNode): string {
12   if (isNode(astNode)) {
13     return this.serializeNode(astNode);
14   } else if (isEdge(astNode)) {
15     return this.serializeEdge(astNode);
16   } else if (isMetaInfo(astNode)) {
17     return this.serializeMetaInfo(astNode);

```

```
18     }
19     return "";
20 }
21
22 protected serializeNodes(nodes: Node[]) {
23     return `${nodes.map((node) => this.serializeNode(node)).join("\n")} `;
24 }
25
26 protected serializeNode(node: Node): string {
27     if (isTaskNode(node)) {
28         return this.serializeTaskNode(node);
29     } else if (isCategory(node)) {
30         return this.serializeCategory(node);
31     } else if (isActivityNode(node)) {
32         return this.serializeActivityNode(node);
33     }
34     return "";
35 }
36
37 [...]
38
39 protected serializeTaskNode(node: TaskNode): string {
40     let serializedNode = `TaskNode ${node.name}`;
41     if (node.label) serializedNode += ` "${node.label}"`;
42     if (node.expanded) serializedNode += ` "expanded"`;
43     if (node.duration) serializedNode += ` ${node.duration}`;
44     if (node.taskType) serializedNode += ` ${node.taskType}`;
45     serializedNode += ";";
46     return serializedNode;
47 }
48
49 [...]
```

Listing 5.4: Serializer

Listing 5.4 shows methods of the implemented custom serializer. The `serialize` method serializes the whole model provided as its `root` input parameter, and the `serializeAstNode` method serializes only the provided `astNode` input node, which could also have children e.g., a category node. Both of the methods are based on the actual implementations of the serialization functions for each types of nodes e.g., the `serializeTaskNode` function, which serializes a task node. These functions are implemented to convert a given type of node to its textual representation corresponding to the grammar rules of the serialized node's type.

The `serialize` and `serializeAstNode` methods break down their input parameters and iterate through every node in the provided abstract syntax tree. After reaching the actual leaf nodes, for which concrete serialization methods are provided i.e., `TaskNode`, `ActivityNode`, `TaskNode`, `Category`, `Edge`, `WeightedEdge`, `Size`, and `Position`, these nodes are serialized, and concatenated with a line break between each node. This way, the textual concrete syntax of either the given model or the given AST node is created.

The implementation of the custom serializer provides solution for the *Serializer* requirement of textual modeling.

5.1.6 Workspace manager

Langium's `DefaultWorkspaceManager` is implemented to find source files in the workspace, which are the files of the workspace that have the grammar specific predefined extension, for the Workflow language `.wf` and `.wfd`. Its `initializeWorkspace` function indexes the workspace folders, and collects information about all referenced and exported AST nodes and stores it locally. This information will be passed to and used by the scope management service later.

```

1 export class WorkflowWorkspaceManager extends DefaultWorkspaceManager {
2   protected onWorkspaceInitializedEmitter = new Emitter<URI[]>();
3
4   constructor(protected services: WorkflowSharedServices) {
5     super(services);
6     this.initialBuildOptions = { validation: true };
7   }
8
9   override async initializeWorkspace(
10    folders: WorkspaceFolder[],
11    cancelToken?: CancellationToken | undefined
12 ): Promise<void> {
13   await super.initializeWorkspace(folders, cancelToken);
14   console.info("Workspace Initialized");
15   const uris =
16     this.folders?.map((folder) => this.getRootFolder(folder)) || [];
17   this.onWorkspaceInitializedEmitter.fire(uris);
18 }
19
20 get onWorkspaceInitialized(): Event<URI[]> {
21   return this.onWorkspaceInitializedEmitter.event;
22 }
23 }

```

Listing 5.5: Workspace Manager

Listing 5.5 shows the extended `WorkflowWorkspaceManager` class. The only necessary modification to the default service is to include an event emitter, the `onWorkspaceInitialized` emitter, which is fired after the workspace was successfully initialized. This way, other servers that depend on the language server can be notified that the language server is up and running, and therefore can be started safely. In the blended modeling framework the GLSP server and the model server depend on the language server and will only be started after this event was emitted.

5.1.7 Scope provider

A scope provider service is a language-specific service that determines the visible target elements for cross-references in a given context. The context determines for which AST node the available cross-references should be provided and which type of elements should be available for cross-references from the given node. An example for the Workflow language is, that when creating an edge, the language server should provide suggestions for all declared nodes in the given model for the source and target nodes, but not other

edges. Another example is, when declaring a size or position element of a node in the `.wfd` file of the model, the language server should provide suggestions for all the nodes declared in the model's `.wf` file.

Langium provides two types of scopes: the local and the global scope. The local scope determines which elements are available for cross-references within the given document, and the global scope contains all the possible cross-referable elements of the workspace by default. As per the *Workspace and scope management* requirement, a model should only be able to reference elements declared in the same model, i.e., a size and position element declared in a model's `.wfd` file should only be allowed to reference elements declared in the model's corresponding `.wf` file and not elements from a different model. Therefore, the global scope must be overwritten accordingly. The local scope provider will not be modified.

```
1  protected override getGlobalScope(  
2      referenceType: string,  
3      context: ReferenceInfo  
4  ): Scope {  
5      // define uris from which references should be included for the global scope  
6      const source = getDocument(context.container);  
7      const uri = source.uri.toString();  
8  
9      const uris: Set<string> = new Set();  
10     uris.add(uri);  
11     if (uri.endsWith("d")) {  
12         // add .wf document  
13         uris.add(uri.slice(0, -1));  
14     } else {  
15         // add .wfd document  
16         uris.add(uri.concat("d"));  
17     }  
18  
19     // the global scope contains all elements known to the language server  
20     // from the documents included in uris  
21     const globalScope = this.globalScopeCache.get(  
22         referenceType,  
23         () => new MapScope(this.indexManager.allElements(referenceType, uris))  
24     );  
25  
26     return globalScope;  
27 }
```

Listing 5.6: Scope provider

Listing 5.6 shows the implementation of the global scope provider function. The only difference between the default implementation and the customized implementation is, that the `allElements` method of the index manager is called with an array of URIs. Without the URIs specified, this method returns every element reference from all the files of the workspace. To be in accordance with the requirements, the global scope should only contain elements of the same model, therefore, an array of both URIs of the current model where the scope request is coming from - one with the model's `.wf` file and one

with the model's `.wfd` file - is passed to the `allElements` method. This way, only the required elements will be provided in the global scope.

5.1.8 Scope computation

To enable scope providing, the available elements for the global and local scopes must first be defined, and the descriptions for these elements must be computed. A description of an element for scope providing contains the element itself, the name of the element and the document it is located in. Afterwards, the scope is computed from these descriptions creating a map object, which maps the name of the elements to the document it was found in.

The default scope computation exports descriptions for the model itself if it is named (i.e., has a unique name identifier) and every child of the model. However, the default service does not consider nested elements i.e., elements that also contain other elements that should be exported for the scope. For the Workflow language, the default scope provider would not consider elements that are declared in a category node, therefore, the service must be extended accordingly.

```

1  export class WorkflowScopeComputation extends DefaultScopeComputation {
2    override async computeExportsForNode(
3      parentNode: AstNode,
4      document: LangiumDocument<AstNode>,
5      children: (root: AstNode) => Iterable<AstNode> = streamContents,
6      cancelToken: CancellationToken = CancellationToken.None
7    ): Promise<AstNodeDescription[]> {
8      const exports: AstNodeDescription[] = [];
9
10     this.exportNode(parentNode, exports, document);
11     for (const node of children(parentNode)) {
12       await interruptAndCheck(cancelToken);
13       this.exportNode(node, exports, document);
14       if (isCategory(node) && node.children) {
15         // recursively include elements of the child model of a category node
16         exports.push(
17           ...(await this.computeExportsForNode(
18             node.children,
19             document,
20             children,
21             cancelToken
22           ))
23         );
24       }
25     }
26     return exports;
27   }
28 }

```

Listing 5.7: Scope computation

Listing 5.7 shows the extended scope computation service. The `computeExportsForNode` method is called by the language server on the root model to compute all the references

that can be exported for the given model. The method iterates through the children of the model and creates descriptions for them where possible. If a given child is a category node, then the function recursively calls itself again on the child model of this category node if available, and therefore also computes the descriptions for the given category's children. This ensures that every node of a model written in the Workflow language can be referenced.

The implementations of the custom workspace manager, scope provider and scope computation services provide solution for the *Workspace and scope management* requirement of textual modeling.

5.1.9 Name provider

Langium's default name provider retrieves the unique identifier name attribute of an arbitrary AST node if exists. The name provider is called while computing the scope of the document, and the computed description of the exported nodes uses this retrieved name as an identifier. Furthermore, in the blended modeling framework, the provided name of an AST node will also be used as a unique identifier of the node in the model index service of the GLSP server. As this index service must index every single element of the model, the language server's name provider must be extended. The default implementation only considers the name attribute of the element, and does not provide a name if this attribute does not exist. As only the model's nodes: the tasks and categories have a name attribute, the custom name provider must also define names for the edges and the size and position elements. The implementation of the name provider is listed in Listing 5.8.

```
1 export class WorkflowNameProvider implements NameProvider {
2   constructor(protected services: WorkflowServices) {}
3
4   /**
5    * Returns the direct name of the node if it has one.
6    * Creates unique name for edges and MetaInfo nodes.
7    *
8    * @param node node
9    * @returns direct, local name of the node if available
10   */
11  getLocalName(node?: AstNode): string | undefined {
12    if (isNode(node)) {
13      return node && isNamed(node) ? node.name : undefined;
14    } else if (isEdge(node)) {
15      return `${node.$type}_${node.source?.$refText}${node.target?.$refText}`;
16    } else if (isMetaInfo(node)) {
17      return `${node.$type}_${node.node?.$refText}`;
18    } else {
19      return undefined;
20    }
21  }
22
23  getName(node?: AstNode): string | undefined {
24    return node ? this.getLocalName(node) : undefined;
```

```
25     }  
26 }
```

Listing 5.8: Name provider

The name provider class provides the direct name of the node or creates a unique name if it does not have one for elements that are already declared, however, names for the elements that are yet to be created either by the language server's code action or by the GLSP server must also be provided. For this purpose, a utility function called `findAvailableNodeName` is created, which provides a unique default name for nodes on creation. The implementation of this function is listed in Listing 5.9. The following prefixes are used by the framework by default on creating new elements: `_tn` for task nodes, `_an` for activity nodes, and `_cat` for categories. A consecutive number is attached to the given prefix by the function, creating a unique name for the model.

```
1  /**  
2   * Provides a name for newly created nodes  
3   * @param container the model root  
4   * @param name the prefix of the name e.g., '_tn'  
5   * @returns a new name that does not yet exists in the model  
6   */  
7  export function findAvailableNodeName(container: Model, name: string): string {  
8    let counter = 1;  
9    let availableName = name + counter;  
10   while (  
11     streamAst(container).find(  
12       (node) => isNode(node) && node.name === availableName  
13     )  
14   ) {  
15     availableName = name + counter++;  
16   }  
17   return availableName;  
18 }
```

Listing 5.9: Name utility

5.2 Implementation of the GLSP Server

The graphical modeling component of the blended textual-graphical modeling framework is based on the GLSP framework. The GLSP framework consists of two main components: the client and the server. The server is responsible for loading, modifying and providing the underlying model and possible actions on it for the client, and the client copes with rendering the diagram and providing editing possibilities for the users.

As the GLSP framework provides an example implementation of a TypeScript-based server and client for graphical modeling on the Workflow language [Foud], the implementation of the blended framework will be based on these provided components. The client already provides every aspect for graphical modeling on the Workflow language, therefore, it will be reused as is. The server will be modified to operate on models originating from the model server instead of the file system, and to match and expand the generation of the graphical model and every available action on it to the requirements of blended modeling.

5.2.1 Graphical model

To provide graphical modeling for the blended modeling framework, the GLSP server must create a graphical model from the input model sourcing from the model server, corresponding to the elements that are foreseen to be rendered on the Workflow language's GLSP client. For this purpose, the following core components of the server must be adapted.

- `Source model storage`: to request the model from the model server instead of loading it from the file system.
- `Model state`: to store the current model state for both the `.wf` and `.wfd` parts of the model, including both the AST and the textual concrete syntax.
- `Model index`: to utilize the language server's name provider for indexing every element of the model.
- `GModel factory`: to generate the graphical model based on the underlying AST sourcing from the model server, and to also incorporate the representation of missing nodes on the diagram.

Source model storage

The default implementation of the source model storage service loads a requested file from disk when the user opens the file on the GLSP client. To enable model synchronization between the language server and the GLSP server, the model must be requested from the model server instead of loading it from the disk, to provide the current status of it for the GLSP server. The custom implementation of the `loadSourceModel` method of the custom `WorkflowModelStorage` serve is provided in Listing 5.10.

```

1  async loadSourceModel(action: RequestModelAction): Promise<void> {
2    // load semantic model from document in language model service
3    const sourceUri = this.getSourceUri(action);
4    const rootUri = sourceUri;
5    const rootUriDetails = `${sourceUri}d`;
6    const root = await this.state.modelService.request(
7      rootUri,
8      isModel,
9      "glsp"
10   );
11   const rootDetails = await this.state.modelService.request(
12     rootUriDetails,
13     isModel,
14     "glsp"
15   );
16   if (!root) {
17     throw new GLSPServerError("Expected Workflow Diagram Root");
18   }
19   if (!rootDetails) {
20     throw new GLSPServerError("Expected Workflow Diagram Details Root");
21   }
22   this.state.setSemanticRoot(rootUri, root, rootUriDetails, rootDetails);
23   this.state.modelService.onUpdate(
24     this.state.semanticUri,
25     async (newModel: Model) => {
26       this.state.replaceSemanticRoot(newModel);
27       this.actionDispatcher.dispatch(
28         UpdateClientOperation.create(false, true)
29       );
30     }
31   );
32   this.state.modelService.onUpdate(
33     this.state.semanticUriDetails,
34     async (newModel: Model) => {
35       this.state.replaceSemanticRootDetails(newModel);
36       this.actionDispatcher.dispatch(
37         UpdateClientOperation.create(true, false)
38       );
39     }
40   );
41 }

```

Listing 5.10: Workflow model storage: loadSourceModel

The client is customized to only operate on `.wf` files, however, as the corresponding `.wfd` file providing the size and position details of the model's elements is also necessary for the graphical model, it must also be requested from the model server as seen in Lines 6 and 11 of Listing 5.10. The function then sets the semantic root of the model state in Line 22 of Listing 5.10, as the further services of the GLSP server operate on this model state.

To provide continuous synchronization with the model server, the model storage service also subscribes to updates coming from the model server when the model is opened initially. The model server's `onUpdate` listener gets fired every time an update on the model happens. Subscribing to this listener for both of the model's URIs provides the

possibility to synchronize the model with other clients of the model server. When an update happens, the GLSP server replaces the model state with the newer one and triggers the re-generation of the graphical model, which afterwards triggers the re-rendering of the model on the client, keeping the model up-to-date.

The further methods of the model storage `saveSourceModel` and `sessionDisposed` are also extended to call the model server's appropriate methods for saving and closing the model.

Model state

The model state is the service that holds the current version of the model i.e., the internal state of the GLSP server and provides access for external services of the language server, making them available for the GLSP server. The model loaded by source model storage service is stored here, and every other service of the GLSP server operates on the model provided by this service. The graphical model is generated from the model provided by the model state service, every action executed on the model in the GLSP client modifies the underlying model of the model state and every update on the model originating from outside of the GLSP server, e.g., the model server, must trigger an update on the underlying model of the model state in order to keep the graphical model up to date.

As explained in the solution concepts for model synchronization in graphical modeling in Section 4.5, the model state holds and maintains two representations of the model: the AST and the textual concrete syntax. This is necessary to provide continuous updates on the textual syntax when an update occurs on the model, maintaining the non-semantic information of the textual syntax. To maintain currentness of the model, the model state must provide methods for the following scenarios of model updates.

- **Update request from the GLSP framework:** if the user edits the model in the diagram editor, the model state must provide update possibilities for the textual syntax. After sending the update to the model server, the model server generates an updated AST for the new textual syntax and sends it back to the GLSP server for re-rendering, updates its internal state and notifies the other clients that the model was updated. If the model server's internal state currently contains arbitrary errors (i.e., there are errors in the text editor, which are not linking errors), it won't generate the updated AST but return an empty model, expressing that updates cannot be currently delegated.
- **Update request from the model server:** if an update outside the GLSP framework happens, the model server notifies the GLSP server to update its internal state. Therefore, the model state must provide methods to replace the AST and the textual syntax of the model with the new, updates ones.

```

1  export interface WorkflowSourceModel {
2      text: string | undefined;
3      textDetails: string | undefined;
4  }
5
6  /**
7   * Custom model state that does not only keep track of the
8   * GModel root but also the semantic root.
9   * It also provides convenience methods for accessing specific language services.
10  */
11  @injectable()
12  export class WorkflowModelState
13      extends DefaultModelState
14      implements JsonModelState<WorkflowSourceModel>
15  {
16      [...]
17
18      setSemanticRoot(
19          uri: string,
20          semanticRoot: Model,
21          uriDetails?: string,
22          semanticRootDetails?: Model
23      ): void {
24          this._semanticUri = uri;
25          this._semanticRoot = semanticRoot;
26          if (uriDetails) {
27              this._semanticUriDetails = uriDetails;
28          }
29          if (semanticRootDetails) {
30              this._semanticRootDetails = semanticRootDetails;
31          }
32          this._semanticText = semanticRoot.$document?.textDocument.getText() ?? "";
33          this._semanticTextDetails =
34              semanticRootDetails?.$document?.textDocument.getText() ?? "";
35          this.index.indexSemanticRoot(this.semanticRoot, this.semanticRootDetails);
36      }
37
38      replaceSemanticRoot(model: Model) {
39          this._semanticRoot = model;
40          this._semanticText = model.$document?.textDocument.getText() ?? "";
41          this.index.indexSemanticRoot(this.semanticRoot, this.semanticRootDetails);
42      }
43
44      async updateSourceModel(
45          sourceModel: WorkflowSourceModel,
46          doNotUpdateSemanticRoot?: boolean,
47          doNotUpdateSemanticRootDetails?: boolean
48      ): Promise<void> {
49          if (!doNotUpdateSemanticRoot) {
50              const model = await this.modelService.update<Model>(
51                  this.semanticUri,
52                  sourceModel.text ?? this.semanticRoot,
53                  "glsp"
54              );
55              if (Object.keys(model).length > 0) {
56                  // only replace semantic root if model is not empty
57                  this._semanticRoot = model;
58              }
59          }
60      }

```

```
60     if (!doNotUpdateSemanticRootDetails) {
61         const model = await this.modelService.update<Model>(
62             this.semanticUriDetails,
63             sourceModel.textDetails ?? this.semanticRootDetails,
64             "glsp"
65         );
66         if (Object.keys(model).length > 0) {
67             // only replace semantic root if model is not empty
68             this._semanticRootDetails = model;
69         }
70     }
71     this.index.indexSemanticRoot(this.semanticRoot, this.semanticRootDetails);
72 }
73
74 insertToSemanticText(node: AstNode, container?: string) {
75     let serializedNode =
76         this.services.language.serializer.Serializer.serializeAstNode(node);
77     if (container) {
78         // new node was inserted as a child node of a category
79         let insertPosition = container.lastIndexOf(";");
80         let newContainer = container;
81         if (insertPosition < 0) {
82             // no childre yet, create model container
83             serializedNode = `{\n${serializedNode}\n}`;
84             insertPosition = container.lastIndexOf(";");
85         } else {
86             // append child to model
87             serializedNode = `${serializedNode}\n`;
88         }
89         newContainer =
90             newContainer.slice(0, insertPosition) +
91             serializedNode +
92             newContainer.slice(insertPosition);
93         this._semanticText = this._semanticText.replace(container, newContainer);
94     } else {
95         this._semanticText += `{\n${serializedNode}\n}`;
96     }
97 }
98
99 deleteFromSemanticText(range: Range) {
100     this._semanticText = this._semanticText.replace(
101         getRangeFromText(range, this._semanticText),
102         ""
103     );
104 }
105
106 updateInSemanticTextDetails(
107     oldText: string,
108     oldAttributeValue: string,
109     newAttributeValue: string
110 ): string {
111     let newText = oldText;
112     const cleanedText = removeAllComments(newText);
113     // find the position of the attribute in the cleaned text
114     const attributeIndex = cleanedText.indexOf(oldAttributeValue);
115     newText = replaceStartingFrom(
116         newText,
117         oldAttributeValue,
118         newAttributeValue,
119         attributeIndex
120     );
121     this._semanticTextDetails = this._semanticTextDetails.replace(oldText, newText);

```

```
122     // return updated text to ensure multiple updates in the same comment
123     return newText;
124 }
125
126 [...]
127 }
```

Listing 5.11: Workflow model state

Listing 5.11 shows selected functions of the custom Workflow model state service. To allow update request from the GLSP framework itself, the model state provides the following methods to enable direct modifications on the textual concrete syntax of the model, which are called by the action handlers of the server.

- `insertToSemanticText`: for creating new elements. The method serializes the new node to be added to the model and appends it to the current textual syntax. If the new node is a child node, then it appends it to the list of the parent’s children.
- `deleteFromSemanticText`: to remove elements from the model. The method deletes the provided range from the textual syntax. The AST of the model contains the range for every element in the textual syntax i.e., the first and last characters associated with the element in the textual syntax, therefore, an element can’t be precisely deleted from the textual syntax.
- `updateInSemanticText`: for updating elements in the textual syntax. The old textual syntax of the element and the old and new values of the attributes must be provided. The old attribute is then replaced by the function with the new one, and the old version of the element’s textual syntax is then replaced by the new one. This method ensures, that non-semantic information within the element’s textual syntax e.g., an in-line comment, will be maintained and will be ignored when searching for the current value of the attribute (i.e., to not interfere with the update, if a comment within the element’s textual syntax contains the same value). A drawback of this method is, that it would only replace the first attribute that matches the old value, so if there are two attributes with the same value, the second one would not be replaced. To solve this problem, either replacing the whole textual syntax with a re-serialized one would provide a solution, meaning, that the non-semantic information would be lost, or Langium would need to provide the ranges of each attribute in the textual syntax, which is not yet the case.

After the action handlers updated the textual syntax, they must also call the provided `updateSourceModel` method. This method triggers the update of the model on the model service, therefore it is necessary to synchronize the model between the textual and graphical client.

To incorporate update requests from the model server the model state provides the `setSematicRoot`, `replaceSemanticRoot` and `replaceSemanticRootDetails`

methods. The `setSematicRoot` must be called when the model is initially loaded to set the URIs, ASTs and textual syntaxis for both the `.wf` and `.wfd` files of the model. The `replaceSemanticRoot` and `replaceSemanticRootDetails` are called by the source model service when the model server triggers an update event. Either the model components of the `.wf` or the the `.wfd` file get updated at a time, depending on which part of the model got updated by another client.

Model index

The model index service is used to index all elements of a model by their ID and to index all references of the model. The model index also offers a set of query methods to retrieve indexed elements and to retrieve unresolvable cross-references to display them as missing nodes. This service is essentially used by the action handler services or any other arbitrary service that performs updates on the model state. The index provides the exact node of the AST for these services and can pursue the updates on this node.

```
1  protected idToSemanticNode = new Map<string, AstNode>();
2  protected references = new Set<string>();
3
4  createId(node?: AstNode): string | undefined {
5      return this.services.language.references.NameProvider.getLocalName(node);
6  }
7
8  indexSemanticRoot(root: Model, rootDetails?: Model): void {
9      this.idToSemanticNode.clear();
10     this.references.clear();
11     streamAst(root).forEach((node) => {
12         this.indexAstNode(node);
13         streamReferences(node).forEach((reference) => {
14             this.references.add(reference.reference.$refText);
15         });
16     });
17     if (rootDetails) {
18         streamAst(rootDetails).forEach((node) => {
19             this.indexAstNode(node);
20             streamReferences(node).forEach((reference) => {
21                 this.references.add(reference.reference.$refText);
22             });
23         });
24     }
25 }
26
27 protected indexAstNode(node: AstNode): void {
28     const id = this.createId(node);
29     if (id) {
30         this.idToSemanticNode.set(id, node);
31     }
32 }
33
34 getAllInvalidReferences(): string[] {
35     return Array.from(this.references).filter(
36         (referenceId) => !this.findSemanticElement(referenceId, isAstNode)
37     );
38 }
39
```

```

40  /**
41   * Creates an index with a different ID for the provided node.
42   * This is necessary to index edges with missing nodes.
43   * @param idNode the node with the ID that should be used for the index
44   * @param node the node that should be indexed by the given ID
45   */
46  addNodeToIndexWithDifferentId(idNode: AstNode, node: AstNode): void {
47      const id = this.createId(idNode);
48      if (id) {
49          this.idToSemanticNode.set(id, node);
50      }
51  }
52
53  findNode(id: string): Node | undefined {
54      return this.findSemanticElement(id, isNode);
55  }
56
57  findSemanticElement<T extends AstNode>(
58      id: string,
59      guard: (item: unknown) => item is T
60  ): T | undefined {
61      const semanticNode = this.idToSemanticNode.get(id);
62      return guard(semanticNode) ? semanticNode : undefined;
63  }
64
65  [...]

```

Listing 5.12: Workflow model index

Listing 5.12 shows important methods of the index service. The index itself is a map, that maps the ID of an element to the corresponding AST node. The `indexSemanticRoot` method creates a reference for every element of the model, using the language server’s name provider as the element’s ID. The `addNodeToIndexWithDifferentId` indexes edges with undefined source or target nodes. These edges will be displayed on the diagram as edges to a node with a random generated ID, and therefore, the edge must be indexed with this ID, to allow the action handlers to provide delete and reconnect operations for these edges.

The `findNode` method shows an example for a method that retrieves an element from the index with a given type and ID. The methods retrieving the other types of elements of the model are implemented similarly.

GModel factory

The GModel factory service is responsible for generating the graphical model from the model state. The generated graphical model is serialized by the GLSP framework and sent to the GLSP client for rendering. The factory gets triggered every time a change in the model state occurs, resulting in a re-generation and re-serialization of the graphical model and re-rendering on the client side.

The factory’s `createModel` method must generate a `GGraph` i.e., the graphical model containing all the corresponding graphical elements of the model in the form of GLSP

specific graphical elements e.g., GNode or GEdge. Corresponding to the *Error handling* requirement of graphical modeling, the generated graphical model must also contain missing nodes i.e., edges without source or target nodes or nodes with missing cross-references.

```
1 createModel(): void {
2   this.missingNodes = new Set();
3   const newRoot = this.createGraph();
4   if (newRoot) {
5     // update GLSP root element in state so it can be used in
6     // any follow-up actions/commands
7     this.modelState.updateRoot(newRoot);
8   }
9 }
10
11 protected createGraph(): GGraph | undefined {
12   const model = this.modelState.semanticRoot;
13   const modelDetails = this.modelState.semanticRootDetails;
14   this.graphBuilder = GGraph.builder().id(this.modelState.semanticUri);
15   model.nodes
16     .map((node) => this.createNode(node))
17     .forEach((node) => this.graphBuilder.add(node));
18   this.createEdgesAndMissingNodes(model, modelDetails).forEach((element) =>
19     this.graphBuilder.add(element)
20   );
21   return this.graphBuilder.build();
22 }
23
24 protected createNode(node: Node): GNode {
25   if (isActivityNode(node)) {
26     return this.createActivityNode(node);
27   } else if (isCategory(node)) {
28     return this.createCategory(node);
29   } else if (isTaskNode(node)) {
30     return this.createTaskNode(node);
31   }
32   return GNode.builder().build();
33 }
34
35 protected createTaskNode(taskNode: TaskNode): GNode {
36   const node = GTaskNode.builder().id(taskNode.name);
37   if (taskNode.taskType) {
38     node.taskType(taskNode.taskType);
39     switch (taskNode.taskType) {
40       case "automated":
41         node.type(ModelTypes.AUTOMATED_TASK);
42         break;
43       case "manual":
44         node.type(ModelTypes.MANUAL_TASK);
45         break;
46     }
47   }
48   if (taskNode.duration) {
49     node.duration(taskNode.duration);
50   }
51   if (taskNode.label) {
52     node.name(taskNode.label);
53   } else {
54     node.name(taskNode.name);
```

```

55     }
56     const size = this.modelIndex.findSize(taskNode.name);
57     if (size) {
58         node.addLayoutOption("prefWidth", size.width);
59         node.addLayoutOption("prefHeight", size.height);
60         node.size(size.width, size.height);
61     }
62     const position = this.modelIndex.findPosition(taskNode.name);
63     if (position) {
64         node.position(position.x, position.y);
65     }
66     return node.build();
67 }

```

Listing 5.13: Workflow GModel factory - creating a task node

Listing 5.13 shows how the factory constructs the GGraph out of smaller graphical nodes corresponding to each element of the diagram. The code shows an example for creating graphical nodes based on the creation of task nodes. The GTaskNode.builder() creates a specific type of GNode which contains the corresponding CSS classes and other task node style properties. The GTaskNode is then further customized with the type of the task and the attributes of the element, which the builder will generate GNode attributes for. Furthermore, the size and position of the element is retrieved using the model index service and are also added to the GNode's properties.

The methods creating GNodes for activity nodes and categories, and GEdges for edges and weighted edges are implemented similarly. To also visualize missing nodes on the diagram, the createEdgesAndMissingNodes method must be implemented. Missing nodes that should be displayed on the diagram can occur in the following scenarios.

- The cross-reference of the source or target node of an edge can not be resolved i.e., the source or target node of an edge is not defined in the model.
- The node's cross-reference of a size or position element can not be resolved i.e., a size or position element is defined for a non-existent node.
- The source or target of an edge is not defined i.e., the edge is only connected to one node.

```

1  protected createEdgesAndMissingNodes (
2      model: Model,
3      modelDetails?: Model
4  ): GModelElement[] {
5      const createdElements: GModelElement[] = [];
6      // non-existent cross-references
7      this.modelIndex
8          .getAllInvalidReferences()
9          .map((missingId) => this.createMissingNode(missingId))
10         .forEach((node) => {
11             createdElements.push(node);

```

```
12     });
13     // add missing nodes to graph
14     Array.from(this.missingNodes)
15       .map((id) => this.createMissingNode(id))
16       .forEach((node) => {
17         createdElements.push(node);
18       });
19
20     // add missing edge sources
21     model.edges
22       .filter((edge) => !edge.source)
23       .forEach((edge) => {
24         const missingNodeId = `${ModelTypes.MISSING_NODE}_${this.modelIndex.createId(
25           edge
26         )}`;
27         const newEdge: Edge = {
28           $container: edge.$container,
29           $type: edge.$type,
30           source: { ref: undefined, $refText: missingNodeId },
31           target: edge.target,
32         };
33         this.modelIndex.addNodeToIndexWithDifferentId(newEdge, edge);
34         createdElements.push(this.createMissingNode(missingNodeId));
35         createdElements.push(this.createEdge(newEdge));
36       });
37
38     // add missing edge targets
39     [...]
40
41     // create edges with valid target and source (remaining edges)
42     model.edges
43       .filter((edge) => edge.target && edge.source)
44       .map((edge) => this.createEdge(edge))
45       .forEach((edge) => {
46         createdElements.push(edge);
47       });
48
49     return createdElements;
50 }
```

Listing 5.14: Workflow GModel factory - creating missing nodes

To display missing nodes from the scenarios described above the `createEdgesAndMissingNodes` function is implemented as displayed in Listing 5.14. The function iterates through every listed scenario and creates `GNodes` and `GEEdges` connected to either the currently declared missing node or to an already existing one. If an ID of a missing node is already declared, then this ID will be used as the missing node's ID. This way, if the user decides to define this node, the reference to this newly defined node will automatically be correct and the missing node will disappear from the diagram. If the missing node does not yet have a known ID, then a random ID is generated for it. The edge referencing to this missing node will also be re-indexed with the newly generated random ID of the missing node, to enable the GLSP framework to interact with it from other services.

The implementation of the source model storage, the model state, the model index and the GModel factory provide solution for the *Graphical model* requirement of graphical

modeling. Furthermore, the implementation of displaying missing nodes on the graphical model contributes to the *Error handling* requirement of graphical modeling, and the custom implementations of the source model storage and the model state contributes to the *Model synchronization* requirement of graphical modeling.

5.2.2 Model editing

The implementation of the source model storage, the model state, the model index and the GModel factory services already provide a graphical representation of the model for the GLSP client. However, to also be able to edit the model in the graphical view, action handlers must be implemented.

An action handler responds to a specific action coming from the GLSP client, such as dragging an element on the diagram or changing the label of an element. An operation handler is a specific form of an action that performs a specific update on the model state. The client provides all the necessary attributes for the server to be able to modify the model state, e.g., the ID of the element that was modified and the new values of the attributes that should be updated. The server's corresponding operation handler then verifies the action if necessary, and changes the textual syntax of the model state accordingly. The update of the model state then triggers sending the update to the model server, which provides the updated AST to the model storage and model state. After the updated AST was provided to the model state, the re-generation of the graphical model also gets triggered.

The designated operation handlers responding to predefined actions on the model and their implementations are bound to the semantics of the DSL that the GLSP framework is operating on. Different DSLs may require different actions, however, how an operation handler is built up, and how it modifies the underlying model state is similar in all of the operation handlers of the blended framework. The following actions are available for the Workflow language in the blended textual-graphical modeling framework, and have designated operation handlers.

- **Change bounds:** changing either the position or the size of a selected element.
- **Create new element:** the creation of following types of nodes and edges are available.
 - Automated task
 - Manual task
 - Decision node
 - Fork node
 - Join node
 - Merge node

- Edge
- Weighted edge - only allowing a decision node as a source node.
- Category
- **Reconnect edge:** changing either the source or target node of the selected edge. If the selected edge is a weighted edge, the source node can only be changed to another decision node.
- **Delete:** deletes the selected element. If the deleted element has connecting edges, the edges will also be deleted.
- **Edit label:** changing the label of a task node.
- **Edit task:** updating an attribute of a task node. Currently, only updating the task type is supported. If the missing node is connected to a simple edge or is not connected to an edge, the creation of an arbitrary type of node is possible. If the missing node has an outgoing weighted edge, only the creation of a decision node is permitted.
- **Create missing node:** the creation of the selected missing node.
- **Update GLSP client:** an empty operation handler to trigger re-generating the graphical model and the update of the GLSP client. The corresponding action is fired by the GLSP server when the model gets updated by the model server.

The listed operations can be triggered in various forms, depending on where the action is fired on the client side. The following possibilities exist in the Workflow GLSP framework.

- **Direct model editing:** the change bounds, the label edit, the reconnect edge and the delete operations can be directly performed on the diagram. The change bounds operation by selecting a node on the diagram and either dragging and dropping it to a new position or resizing it by dragging one of the edges of the node. The label edit operation by double clicking on the designated label and typing in the new name for the label. The reconnect edge operation by selecting an edge on the diagram and dragging one of the ends of the edge to a new node. The delete operation by clicking on an element and hitting the delete key.
- **Tool palette:** each of the create new element operations are available on the tool palette. To create a new node the user must click on the desired node type and click on the diagram in the position where the new element should be created. The tool palette with the available actions is shown in Figure 5.2.
- **Command palette:** the edit task type, the create new element, the delete and the create missing node operations are available on the command palette, as shown in Figure 5.1. The command palette is context dependent, meaning that the available

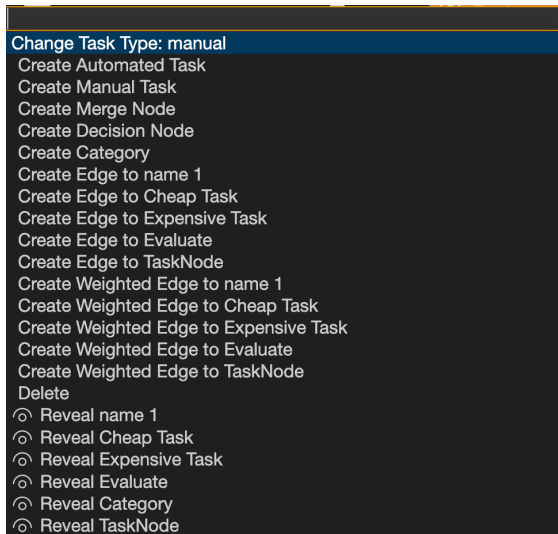


Figure 5.1: Command palette

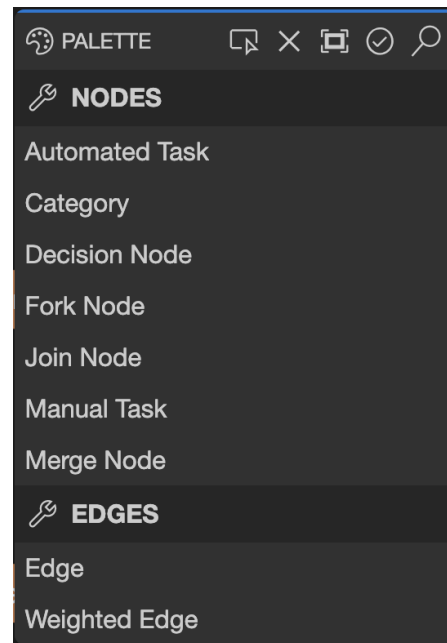


Figure 5.2: Tool palette

actions displayed depend on whether an element was selected and which element was selected. The create new element and delete operations are available generally, the change task type operation is available only when called on a selected task node and the create missing node operations are available only when called on a selected missing node.

Operation handler structure

As mentioned before, even though every operation handler modifies or interacts with the model in a different way, the executed steps in the implementation are similar. The core of each operation handler is the `Command`. A command implements a specific modification of the model state, which can be applied by invoking its `execute()` function. The blended framework's `WorkflowCommand` also provides implementations to the `undo` and `redo` methods, which respectively undo or redo the change they made to the model state.

```

1 @injectable()
2 export class WorkflowDeleteOperationHandler extends OperationHandler {
3   operationType = DeleteElementOperation.KIND;
4
5   @inject(WorkflowModelState) protected state: WorkflowModelState;
6
7   createCommand(operation: DeleteElementOperation): Command | undefined {
8     if (!operation.elementIds || operation.elementIds.length === 0) {
9       return;

```

```
10     }
11     return new WorkflowCommand(this.state, () => this.deleteElements(operation));
12 }
13
14 protected deleteElements(operation: DeleteElementOperation): void {
15     for (const elementId of operation.elementIds) {
16         const element = this.state.index.findSemanticElement(elementId, isDiagramElement);
17         if (element?.$cstNode) {
18             this.state.deleteFromSemanticText(element.$cstNode.range);
19         }
20         if (isNode(element)) {
21             this.deleteIncomingAndOutgoingEdges(element);
22             this.deleteSizeAndPosition(element);
23         }
24     }
25 }
26
27 private deleteIncomingAndOutgoingEdges(node: Node): void {
28     this.state.semanticRoot.edges
29         .filter(
30             (edge) =>
31                 edge.source.$refText === node.name || edge.target?.$refText === node.name
32         )
33         .forEach((edge) => {
34             if (edge.$cstNode) {
35                 this.state.deleteFromSemanticText(edge.$cstNode.range);
36             }
37         });
38 }
39
40 private deleteSizeAndPosition(node: Node): void {
41     this.state.semanticRootDetails.metaInfos
42         .filter((metaInfo) => metaInfo.node.$refText === node.name)
43         .forEach((metaInfo) => {
44             if (metaInfo.$cstNode) {
45                 this.state.deleteFromSemanticTextDetails(metaInfo.$cstNode.range);
46             }
47         });
48 }
49 }
```

Listing 5.15: Delete operation handler

Listing 5.15 shows the implementation of the delete operation handler as an example for implementing operation handlers. An operation handler must always have a unique `operationType` attribute corresponding to the type of the action that should trigger this operation. The only function that the operation handler must implement is the `createCommand` function, which must create and return a `WorkflowCommand` to be executed by the GLSP server.

The command must define how the model state should be updated. This is done by providing a function that executes the necessary updates on the model state, which in this case is the `deleteElements` function. The `WorkflowCommand` records the current state and executes the provided function on the current state updating it. The previous state is recorded in case the user triggers an undo or redo action.

In the provided example, the model state is updated as follows. Firstly, the textual syntax is updated, and the elements to be deleted are cut out from the textual syntax. This is the general approach of updating the model state in every implemented operation handler. The command then triggers the model update on the model server providing an updated AST for the GLSP server's model state. Afterwards, this triggers the re-generation of the graphical model on the GLSP server.

In the specific operation handler shown in Listing 5.15, deleting an element might also trigger the deletion of other elements from the model too. If the deleted element is a node and has connected edges, the `deleteIncomingAndOutgoingEdges` method is also executed for this node deleting the connected edges for it. Furthermore, if the deleted element is a node, meaning that it also has size and position elements persisted on the model, these will also be deleted by executing the `deleteSizeAndPosition` method.

Action providers

To make the implemented operation handlers available for execution on the client, the framework must implement action providers. Action providers define which actions or operations are available for execution, in which form and on which elements of the model. In the previous section three ways of providing user actions for the graphical model were defined: executing actions via directly editing the model, executing actions on the tool palette and executing actions on the command palette.

The implementation of the tool palette and direct model editing action calls are done on the client project, as these do not require dynamic modification of available actions for the Workflow language. Every element on the diagram is re-sizeable and might be re-positioned and also every type of element might be created via the tool palette continuously. However, the action provider for the command palette must be implemented on the server side.

The available actions on the command palette depend on the context it was called from i.e., which element was selected when opening it. The create new element and delete operations are available generally, the change task type operation is available only when called on a selected task node and only to change its task either from manual to automated or vice versa and the create missing node operations are available only when called on a selected missing node.

```

1  export class WorkflowCommandPaletteActionProvider extends CommandPaletteActionProvider {
2    getPaletteActions(
3      selectedElementIds: string[],
4      selectedElements: GModelElement[],
5      position: Point,
6      args?: Args
7    ): LabeledAction[] {
8      const actions: LabeledAction[] = [];
9      if (this.modelState.isReadOnly) {
10         return actions;
11      }

```

```
12     [...]
13     // Create actions
14     const location = position ?? Point.ORIGIN;
15
16     // Create actions for missing nodes with IDs
17     if (selectedElements.length === 1) {
18         const element = selectedElements[0];
19         if (element instanceof GMissingNode) {
20             // if missing node has weighted outgoing edge it can only be a decision node
21             if (
22                 this.modelState.index
23                     .findOutgoingEdges(element.id)
24                     .filter((edge) => isWeightedEdge(edge)).length === 0
25             ) {
26                 actions.push({
27                     label: "Create Missing Node - Manual Task",
28                     actions: [
29                         CreateNodeOperation.create(ModelTypes.MANUAL_TASK, {
30                             location,
31                             args: { name: element.id },
32                         }),
33                     ],
34                     icon: "fa-plus-square",
35                 });
36                 [...]
37             }
38             actions.push({
39                 label: "Create Missing Node - Decision Node",
40                 actions: [
41                     CreateNodeOperation.create(ModelTypes.DECISION_NODE, {
42                         location,
43                         args: { name: element.id },
44                     }),
45                 ],
46                 icon: "fa-plus-square",
47             });
48         }
49     }
50 }
51 [...]
52 }
```

Listing 5.16: Command palette action provider

Listing 5.16 shows a segment of the implemented command palette action provider. The `getPaletteActions` function must return an array containing every available action for the given selected elements. In the example, actions for creating a manual task and a decision node out of a missing node are shown. The action provider first must make sure, that the selected element is a missing node to provide this action, and then define the possible actions including the operation that must be called on executing the action and the necessary attributes for the operation. To ensure a semantically correct model, the example shows how only a creation of a decision node out of a missing node is possible, if the missing node has an outgoing weighted edge connected to it.

The implementation of the custom operation handlers and the command palette action provider provides solution for the *Model editing* requirement of graphical modeling.

Furthermore, only allowing the creation of elements according to the Workflow language's semantics contributes to the *Error handling* requirement of graphical modeling, and the custom implementations of the `WorkflowCommand` contributes to the *Model synchronization* requirement of graphical modeling.

5.2.3 Validation

The GLSP framework must ensure that creating and editing a model results in a syntactically and semantically correct model in regards of the underlying DSL i.e., the Workflow language. In other words, creating, updating, or deleting elements of the model must result in a syntactically and semantically correct AST and textual syntax of the model. To ensure this criteria, the following solutions are implemented.

- **Creating new elements:** the corresponding operation handler ensures that the resulting AST and textual syntax of the model state are correct after execution. This is possible, as the actions do not require custom user input, the inserted elements and the whether creating a new element is possible are predefined according to the language's syntax and semantics.
- **Deleting elements:** similarly to creating new elements, it can also be ensured by the operation handler that the AST and textual syntax of the model are correct after deletion.
- **Updating elements:** the only update that requires custom user input is the label edit operation. The user is allowed to enter an arbitrary text as the new label of a task node, therefore it must be validated. For this purpose, the `LabelEditValidator` interface is implemented, which ensures, that the label is only valid and can be persisted if it is not empty, and warns the user if it does not begin with a capital letter. The correctness of other update operations are ensured by their corresponding action handlers.

To showcase validation on the graphical model a custom implementation of the `ModelValidator` service is provided on the GLSP server, validating the model for duplicate IDs and duplicate edges. However, this would not be necessary, as having duplicate IDs would cause a runtime exception on the GLSP client, as it is not capable of rendering two elements with the same ID. The implementation of the operation handlers, the model validator, and the label edit validator services provide solution for the *Validation* requirement of graphical modeling.

5.2.4 Error handling

Error handling in graphical modeling can be prevented in a large measure by defining default attribute values when creating or modifying an element, and only allowing updates on the model that result in both a semantically and syntactically correct model. For

example, on creating a new element, the element's attributes will be filled by default valid values. On updating an element, the updated attributed should be validated if it is a custom user input, or should be a predefined valid value, that a user can choose to update the value to. On deleting an element, other elements referencing to this element should also be updated or deleted.

As the blended textual-graphical modeling framework allows simultaneous editing of the textual and graphical representations of the model, errors might still arise while editing the model in the text editor, as updates to the model can not be limited there. To still make interactions with an erroneous model in the graphical editor available after an error occurs during textual model editing, the model service provides updates to the GLSP server in following cases on the Workflow language:

- **Size or position of element gets deleted.** As the resulting model is still syntactically correct, the model must still be rendered on the diagram. If the position of an element is missing, it will be displayed on the (0,0) position of the diagram, and if the size of an element is missing a default value will be applied for rendering. On moving or resizing the element, the missing attribute will be added to the model again.
- **Missing node in a size or position element.** If a size or position's element references to a node that is not defined on the model, a node will be rendered on the diagram with the corresponding size and/or position and ID on the diagram, displayed as a missing node. The user is then able to define a node to the missing attribute with the available command palette actions.
- **Missing node in a source or target edge reference.** If an edge has a source and/or target attribute referencing an undefined node, a missing node will be rendered on the diagram with the corresponding ID and the edge connecting to it. The user is able to define a node for the missing node with a command palette action. The created node will have the ID of the missing node, and therefore the missing cross-reference of the edge will be resolved. Another way of handling this error is to reconnect the edge to point to an existing node instead of the missing node. In this case, the missing node disappears from the diagram.
- **Missing source or target node of edge.** If an edge has an undefined source or target node, the edge will still be displayed on the graphical model connected to a missing node with a random unique ID. The user is able to create a new node from the missing node or reconnect the edge to an existing node, in which case, the new node's ID or the selected node's ID will be inserted into the edge's missing source or target reference.

In other error scenarios on the textual model, the model server does not provide updates to the GLSP server, and also does not accept updates from the GLSP server. This is to

prevent overwriting the erroneous textual syntax in the text editor, which would cause confusion to the user and go against the concept of incremental updates. After the error is fixed in the text editor, the model server accepts updates again from the GLSP server.

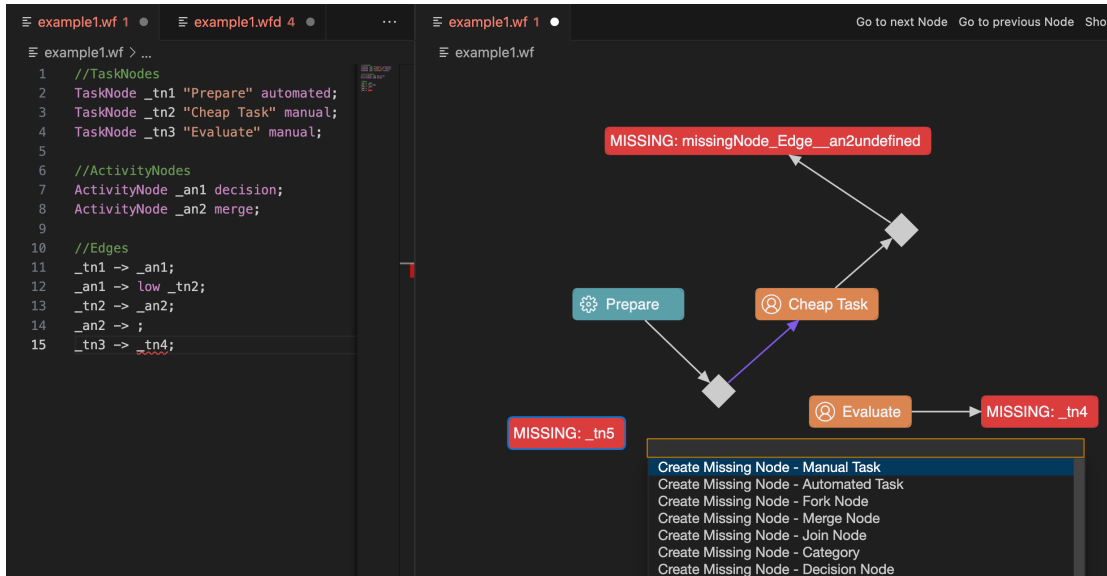


Figure 5.3: Error handling in the graphical editor

Figure 5.3 shows the different possibilities a missing node might appear on the graphical model. The left side shows the text editor and the right side the graphical editor, both operating on the `example1` model. `_tn5` represents a missing cross-reference for size and position elements, both of these elements are defined in the `example1.wfd` file. `_tn4` represents a missing cross-reference in an edge's target. The definition of the edge is shown in line 15 of the text editor, where the text editor also marks the missing cross-reference as an error. The last missing node represents the missing target node definition in the edge defined in line 14. `_tn5`'s open command palette shows the available actions for creating a node out of the missing `_tn5` node.

The implementation of the graphical model including the representation of the missing nodes and the implementation of the action handlers and action provides according to the scenarios listed above, provide a solution to the *Error handling* requirement of textual modeling.

5.3 Model Server

The model server is the central connecting artifact of the blended textual-graphical modeling framework. It allows the connected clients to access the requested model, to provide the newest version of the model for the clients and to synchronize model updates between them. Without the model server, simultaneous updates on the same model i.e., editing the same file would not be possible. Updates on a client would only be delegated

to other clients if the file got updated on the disk or the updates would explicitly be sent to a client.

The implementation of the model server must enable to edit the same model with different clients at the same time, while keeping the model in sync on the connected clients. For this, the model server stores the content of the file containing the model i.e., the textual concrete syntax, when the model was first requested by a connected client. From this point on, any other client that requests to open this model will be sent the model stored in the model server's internal state, making sure that the client receives the current version. Update requests coming from a client also only update the internal model state of the model service, and the model only gets persisted to disk when a client explicitly requests the server to save it. Additionally, the model server notifies the connected clients when an update on a model in its internal state happens, and provides them the updated version of the model on request.

Base project

The model server's implementation is based on the CrossModel model server from CrossBreeze [Crob], and is extended and modified to provide simultaneous model editing capabilities for the graphical and textual clients. The implementation on one hand provides a possibility for the clients to send request messages via an RPC connection. These messages are handled in the `ModelServer` class, with each request calling the corresponding method of the `ModelService` class, that further handles the execution. On the other hand, the server can also be integrated as a custom service of the language server and the GLSP server, in which case, the mentioned servers can directly call the available methods of the `ModelService` class. The latter approach is used in the blended modeling framework.

The main concept of the model server is to provide Langium's model provision and updating capabilities to clients which do not operate on the LSP. To integrate with the language server, the model server is registered as a custom model service as a part of the language servers added shared custom services. Furthermore, a wrapper class `OpenableTextDocuments` is implemented to extend the LSP's default `TextDocuments` class, which provides events that are fired when a text document is opened, changed, saved and closed by VS Code's text editor and keeps hold of the synchronized text documents, which are currently opened by the language server. The `OpenableTextDocuments` class extends this by providing methods to be able to invoke events from within the language server. These events are used to notify the model server's internal state in the `OpenTextDocumentManager` that a language server event is occurred, and therefore the state of the model on which the event occurred can be updated.

5.3.1 Model server internal state

The `OpenTextDocumentManager` is extended to hold the current state of each model per client that is opened by either the language server or another client. The state is a map with `uri` of the model and `client` as key, and the model's `version` and `text` as value. Storing the current version of a model per client is necessary to provide incremental updates synchronized between multiple clients. An update is only delegated to other clients if the client's version of the model where the update is coming from is greater than the client's model version where the update should be delegated to. This way, infinite updates where a delegated update would trigger another update on the client where the update was originating from can be prevented.

Updating the internal state from a model server client



Figure 5.4: Updating the internal state from a model server client

To update the internal state, the update can be originating from two directions: either from a connected client e.g., the GLSP server, or the language server. The former update process is displayed in Figure 5.4.

An update from a client can be either in form of the textual concrete syntax of the model or the AST of the model. In latter case, the update method of the `Model Service` serializes the whole AST, therefore, it should be only used if maintaining non-semantic information on the textual syntax is not necessary or on initial serialization of the model. The model service then retrieves the client's current version of the model, and sends an update request to the `OpenTextDocumentManager` with an updated version of it, which then updates its internal state. To provide an updated AST for the client that requested the update from the model service, the model service also retrieves the current `LangiumDocument` of the model, updates it with the new version of the model's textual syntax and triggers `Langium` to rebuild the document. This way, the model's AST is updated and can be sent back to the client that requested the update, e.g., the GLSP server, that will then rebuild its graphical model based on the received updated AST. Furthermore, as the `LangiumDocument` is up-to-date it can be provided to other clients on request.

Updating the internal state from the language server

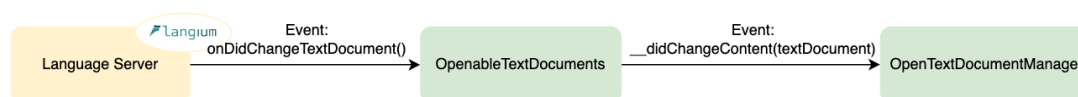


Figure 5.5: Updating the internal state from the language server

To detect updates coming from the text editor through the language server, the `OpenableTextDocuments` class must listen to the `onDidChangeTextDocument` event emitted by the underlying LSP connection, as shown in Figure 5.5. This event gets emitted every time a change occurs in the model’s text editor. The `OpenableTextDocuments` class then updates its synchronized documents and fires the `__didChangeContent` event, to which the `OpenTextDocumentManager` service subscribes to. This is necessary as the `OpenableTextDocuments` class is injected into the `OpenTextDocumentManager` class, therefore, communication in this direction is only possible via events and listeners. After the `__didChangeContent` event is emitted, the `OpenTextDocumentManager` updates its internal state of the text editor’s model version and text.

5.3.2 Model access and provision

To provide access and modification possibilities to the model server’s internal state the following methods are implemented in the `Model Service`.

- **open:** triggers the `OpenTextDocumentManager` to read the file from disk if it has not been opened yet. On initial open the model is loaded with version 1 to the model server’s internal state and the `OpenableTextDocuments` class is also notified that the document was opened. If the request was triggered by the `OpenableTextDocuments` i.e., the text editor, the model service replaces the text editor’s content with the up-to-date version of the model’s textual syntax, if it has already been updated by another client.
- **request:** opens the document, if it has not been opened yet, and retrieves the current `LangiumDocument` containing the AST and the textual syntax of the model. The `LangiumDocument` is up-to-date as it has either been rebuilt on a client model update request, or on a change made in the text editor, or it is built for the first time if the model has not been opened yet.
- **save:** firstly, the method serializes the model if it is called with an AST provided, or directly triggers the `OpenTextDocumentManager`’s `save` method with the provided textual syntax. The `OpenTextDocumentManager` then saves the textual syntax of the model to disk.
- **close:** triggers the `OpenTextDocumentManager`’s `close` method, that removes the model from the requesting client’s state and notifies the `OpenableTextDocuments` if no other clients have this model open to also remove the model from its synchronized documents.
- **update:** the model service’s `update` method is implemented as discussed in Section 5.3.1.

5.4 Model Synchronization

To provide synchronous graphical-textual modeling for the blended framework, the model server must notify its connected clients if another client performed an update on one of its opened models. As discussed in Section 5.3.1, an update from either a connected client or the language server already updates the model server's internal state for the given client, however, these updates still need to be synchronized with the other clients if they also have the updated model open. For this purpose, two possible directions must be implemented similarly to updating the model server's internal state.

Updating the text editor

Updates originating from a connected client e.g., the GLSP server, must be synchronized with the language server after the model's internal state was updated. Therefore, the `OpenTextDocumentManager`'s `update` method must provide the updated textual syntax of the model for the language server. For this, the method looks up whether the language server's version of the model is behind the current version of the model and notifies the `OpenableTextDocuments` to update its synchronized documents with the new textual syntax of the model. Furthermore, if the model is open in a text editor, the text editor's content is replaced with the new version of the model's textual syntax. The methods implementation is shown in Listing 5.17

```

1  /**
2   * Updates the semantic model stored in the document with the given model
3   * or textual representation of a model.
4   * Any previous content will be overridden.
5   * If the document was not already open for modification, it throws an error.
6   * If the request is coming from a non-LSP client, the content of an open text editor
7   * with the given document will be overwritten.
8   *
9   * @param uri document URI
10  * @param model semantic model or textual representation of it
11  * @returns the stored semantic model
12  */
13  async update(
14    uri: string,
15    version: number,
16    text: string,
17    client?: string
18  ): Promise<void> {
19    if (!this.isOpen(uri, client)) {
20      throw new Error(`Document ${uri} hasn't been opened for updating yet`);
21    }
22
23    this.openDocuments.set(
24      { uri: this.normalizedUri(uri), client: client ?? "text" },
25      { version, text }
26    );
27
28    if (client && client !== "text") {
29      let textDocument = this.openDocuments.get({
30        uri: this.normalizedUri(uri),
31        client: "text",

```

5. PROTOTYPE IMPLEMENTATION

```
32     });
33     let clientDocument = this.openDocuments.get({
34       uri: this.normalizedUri(uri),
35       client,
36     });
37
38     // update the version of the LangiumDocument if update was not coming
39     // from the LSP, otherwise it is already updated
40     if (
41       (!textDocument && clientDocument) ||
42       (textDocument && clientDocument && clientDocument.version > textDocument.version)
43     ) {
44       this.textDocuments.notifyDidChangeTextDocument (
45         {
46           textDocument: VersionedTextDocumentIdentifier.create (
47             this.normalizedUri(uri),
48             clientDocument.version
49           ),
50           contentChanges: [{ text }],
51         },
52         client
53       );
54       // replace the content of an open text editor with the given update
55       if (
56         textDocument &&
57         clientDocument &&
58         clientDocument.version > textDocument.version
59       ) {
60         await this.replaceTextEditorContent(uri, text);
61       }
62     }
63   }
64 }
```

Listing 5.17: Model server: OpenTextDocumentManager’s update method

If an update was triggered by the `OpenableTextDocuments` update event, the same method is executed on the `OpenTextDocumentManager`. Therefore, it is necessary to check whether the received version is greater than the one that the language server is already operating on, otherwise, this method would replace the text editor’s content in an infinite loop, as it would be triggered every time a change in the text editor occurs.

```
1  /**
2   * Generates a workspace edit and applies it to replace
3   * the content of an open text editor
4   * @param uri the uri of the open file
5   * @param text the text that the current content will be replaced with
6   */
7  private async replaceTextEditorContent(uri: string, text: string) {
8    let workspaceChange = new WorkspaceChange();
9    let textChange = workspaceChange.getTextEditChange(uri);
10   textChange.replace(
11     {
12       start: { line: 0, character: 0 },
13       end: { line: Number.MAX_SAFE_INTEGER, character: Number.MAX_SAFE_INTEGER },
14     },
15     text
16   )
```

```

16   );
17   await this.connection.workspace.applyEdit(workspaceChange.edit);
18 }

```

Listing 5.18: Model server: replaceTextEditorContent method

To replace the open text editor’s content, the method displayed in Listing 5.18 is implemented. This method sends a `WorkspaceChange` command to the open LSP connection, delegating the text editor with the opened document of the provided URI to replace its entire content with the provided new textual syntax.

Updating a connected client

To enable model update synchronization for arbitrary clients, the `OpenTextDocumentManager`’s update method does not explicitly send updates to the connected clients. On the contrary, it provides a listener, that subscribes to Langium’s `DocumentBuilder` service and triggers execution of the provided function every time a document reaches the `Validated` state. The implementation of this method is listed in Listing 5.19.

```

1  /**
2   * Subscribes to the 'Validated' state of Langium's document builder.
3   *
4   * @param uri Uri of the document to listen to. The callback only gets
5   * called when this URI and the URI of the saved document are equal.
6   * @param client the requesting client
7   * @param listener Callback to be called
8   * @returns Disposable object
9   */
10 onUpdate<T extends AstNode>(
11   uri: string,
12   client: string,
13   listener: (model: T) => void
14 ): Disposable {
15   return this.documentBuilder.onBuildPhase(
16     DocumentState.Validated,
17     (allChangedDocuments, _token) => {
18       const changedDocument = allChangedDocuments.find(
19         (document) => document.uri.toString() === uri
20       );
21       if (changedDocument) {
22         const textDocumentVersion = this.getClientDocumentVersion(
23           this.normalizedUri(uri),
24           "text"
25         );
26         const clientDocumentVersion = this.getClientDocumentVersion(
27           this.normalizedUri(uri),
28           client
29         );
30         // only trigger listener is update is coming from text editor
31         if (clientDocumentVersion && textDocumentVersion > clientDocumentVersion) {
32           if (
33             !changedDocument.diagnostics ||
34             (changedDocument.diagnostics.length === 0 &&

```

```
35         changedDocument.parseResult.parserErrors.length === 0) ||
36         changedDocument.diagnostics.filter(
37           (value) => value.data.code === "linking-error"
38         ).length === changedDocument.diagnostics.length
39       ) {
40         const root = changedDocument.parseResult.value;
41         return listener(root as T);
42       }
43     }
44   }
45 }
46 );
47 }
```

Listing 5.19: Model server: onUpdate listener

The calling client must subscribe to every model they want to listen to updates for with providing the model's URI. The `onUpdate` method then filter's the changed documents, to only trigger the listener function if the client has subscribed to the given model. Then, the method verifies whether the client's version is behind the language server's version, and if so, executes the listener function. This is again necessary to prevent infinite updates.

To provide the possibility of displaying missing nodes on the graphical model, the listener also gets executed if the updated model has linking errors, which means that some of the cross-references could not be resolved on the model. Otherwise, the listener would only get triggered if the built document does not have any errors, to prevent the GLSP server from displaying an invalid model.

5.4.1 Summary

The implementation of the model server provides model access, model provision and model synchronization satisfying the stated requirements for the model server from Chapter 4. Furthermore, the implemented solution concept also provides an answer for Research Question 1, providing a solution to how textual and graphical editors can manage and manipulate the model's underlying AST jointly.

5.5 VS Code Extension

To combine the GLSP framework, the Langium language server, and the model server together to form the blended modeling framework, a VS Code extension is developed [Mica]. The Workflow GLSP client already provides an extension [Eclif], which needs to be extended to make use of the blended backend.

To integrate the language server as an LSP provider, it is launched as a language client in the extension with the Workflow language registered as a DSL that the language client should operate on. This way, VS Code knows that the models written in the Workflow language with the extensions `.wf` and `.wfd` should be hooked up with the Langium

language server if opened in the default text editor. To also launch the GLSP server and the model server in the extension, a default language server starter script is developed. After the language server has successfully started and the workspace for the extension is initialized, the script also starts the GLSP server together with the model server. This way, all the necessary backends are started and connected for the extension.

To also make graphical modeling available, the GLSP client must be registered as a custom editor provider. The default editor for the `.wf` files is set to be the GLSP client, hence, on opening the model within the VS Code extension, it will be automatically loaded in the GLSP editor instead of the text editor. The operating extension is shown in Figure 5.6. The left panel shows the GLSP client connected to the Workflow GLSP server, and the middle and the right panel show the default text editor connected to the Workflow language server. Changes made in the GLSP editor are synchronized and automatically displayed in the corresponding text editor and changes made in either of the text editors automatically update the GLSP client's graphical model.

5.6 Summary

This chapter provided details about the implementation of the blended graphical-textual modeling framework. The chapter elaborated how each component of the framework: the language server, the GLSP server, and the model server were implemented to satisfy the requirements for each component listed in the previous Chapter 4 based on the introduced solution concepts. The implementation of the VS Code extension concludes the implementation of the framework's components and forms the blended modeling tool operating on the Workflow language.

Table 5.1 provides an overview of the implemented concepts for the blended textual-graphical modeling framework based on the Workflow language.

The following chapter describes how the implemented framework was evaluated.

	Textual modeling	Graphical modeling
Grammar	Workflow language split into two files: <code>.wf</code> for nodes and edges <code>.wfd</code> for size and position of the nodes (which are only needed for the graphical model)	
Modification model	textual syntax in text editor; LangiumDocument is rebuilt on every change in the textual syntax	the model's AST and textual syntax provided by the model server; incremental updates on the textual syntax, AST is regenerated by the model server after updated textual syntax is sent
Model-specific features	syntax highlighting, serialization, workspace and scope management	model editing on the diagram
Validation	Langium's default validators to detect lexer, parser and linking errors; custom validators to ensure semantics	custom validators to ensure semantics; textual syntax must be correct to ensure the generation of the graphical model
Error handling	custom validators provide hints and error messages; code actions to solve linking errors by creating new nodes for missing references	only models with linking errors are delegated to the GLSP server, otherwise updates on the diagram are not allowed; displays missing nodes for unresolvable cross-references; provides command palette actions to solve linking errors by creating new nodes for missing references
Cross-references	global scope is restricted to only provide node descriptions between the model's <code>.wf</code> and <code>.wfd</code> files	model's corresponding <code>.wfd</code> file is loaded on opening a <code>.wf</code> file, cross-references between them are resolved by the server
Model editing	unrestricted updates in text editor; language server rebuilds model on each update	only allow updates that result in a semantically and syntactically correct model; if update requires custom user input, input is validated
Model synchronization	updates on graphical model result in a new textual syntax delegated to the model server, model server replaces text editor's content	updates on textual model are emitted by listener on Langium's DocumentBuilder, updated LangiumDocument containing AST and textual syntax of the model is sent to GLSP server

Table 5.1: Implemented concepts of the blended modeling framework.

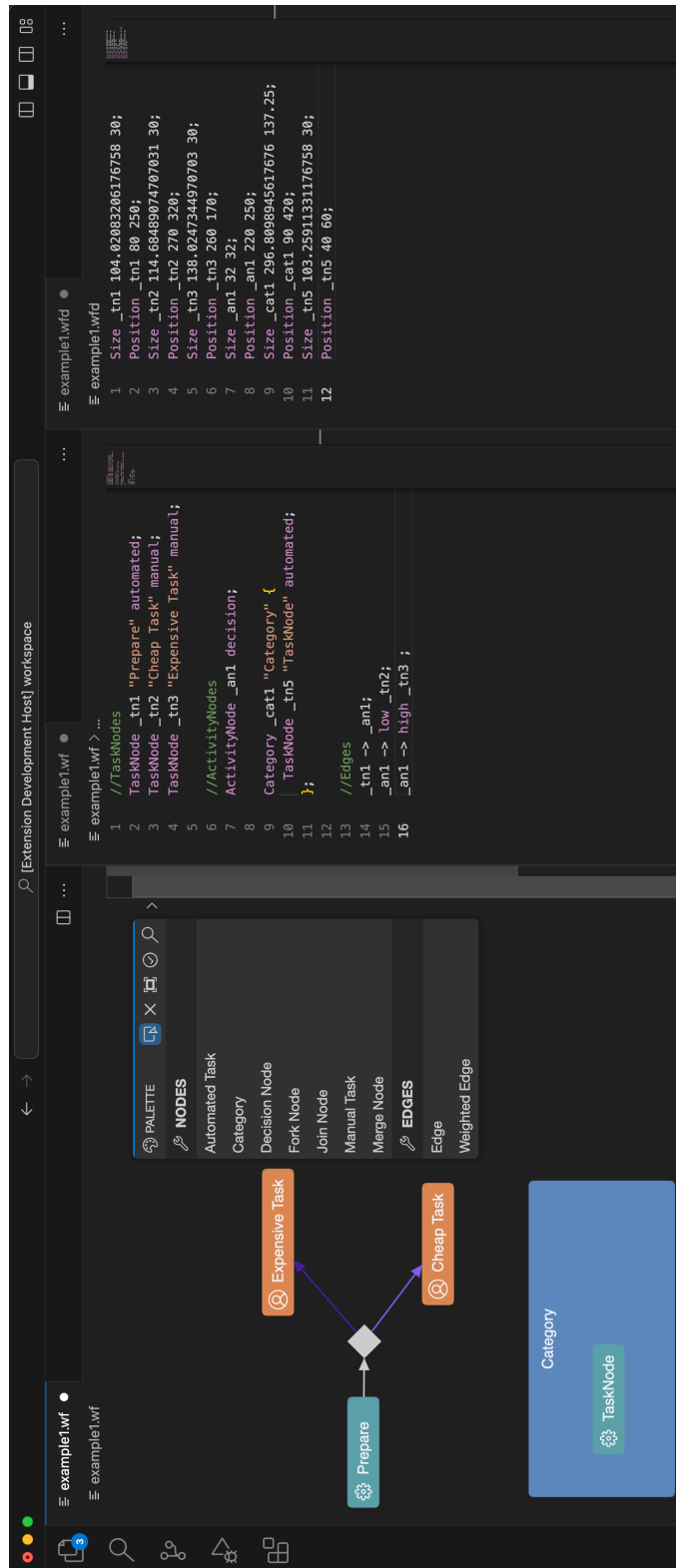


Figure 5.6: VS Code extension: the blended framework

Evaluation

This chapter describes the chosen procedure to evaluate the implemented blended textual-graphical modeling framework. Then, it introduces the artifacts and scenarios, on which the framework was evaluated and elaborates the implementation steps executed during the evaluation process. Finally, the chapter concludes the results of the evaluation procedure.

6.1 Evaluation Procedure

The evaluation procedure's main goal is to show the feasibility of the implemented generic concepts of the blended modeling framework - which was developed based on the Workflow language - for arbitrary modeling languages created by Langium and with a corresponding GLSP framework. For this purpose, the descriptive evaluation method [Hev07] is chosen: the implemented artifacts will be instantiated and evaluated based on two BIGUML [Bor] [MB23a] use cases: the Package Diagram and the Class Diagram. The generic components of the blended framework will be instantiated with BIGUML as a proof of concept, furthermore a description of which concepts and components of the blended framework could be successfully implemented on the two scenarios, which language-specific features might be implemented and which concepts could not be implemented will be provided.

6.2 BIGUML Artifacts

To instantiate the implemented concepts of the blended modeling framework on another modeling language the following resources are required: a language server for the modeling language implemented with Langium a TypeScript-based GLSP server and a VSCode-based GLSP client operating on the given modeling language. To acquire these resources for the package diagram and class diagram use-cases of UML [Obj], resources of a thesis

developed by David Jäger in parallel will be used. Jäger's thesis *Frontend-only browser-based modeling tools* [Jä24] develops a GLSP-Langium-based modeling tool utilizing a model server for AST access and updates, which can be browser-packaged, meaning, that the complete modeling tool can run in a browser. The implemented tool also provides a generator service that generates a complete Langium-based language server for a modeling language defined as a TypeScript-based meta-model. The generated language server is based on a generic JSON grammar, that is also generated by this tool. To evaluate the developed artifacts Jäger implemented a meta-model for UML from which the generator tool created a UML-based Langium language server. Jäger's evaluation procedure also includes the implementation of a corresponding TypeScript-based GLSP server utilizing the implemented model service, which is connected to an already existing VSCode-based GLSP client for UML¹ [Bor] [MB23a] .

To evaluate the blended textual-graphical modeling framework, the implemented UML use cases of Jäger's thesis will be used: the package diagram and the class diagram. Both of the use cases are based on the same generated UML grammar, and therefore the same language and model server. Solely, the GLSP framework is customized to operate on the two types of diagrams. The provided GLSP server and language server will be customized to utilize the model service of the blended framework and make simultaneous textual and graphical modeling available.

6.3 Implementation of the BIGUML Scenarios

The implementation concepts of the package diagram and class diagram use cases are shown in Table 6.1. As the evaluation's goal is to clarify the feasibility of the generic concepts of the blended modeling framework, the requirements that were tailored specifically to the Workflow language's semantics will not be considered and the provided implementations of the BIGUML artifacts will be used as is. Generic implementation means, that the implemented generic solutions for the Workflow blended framework are used, and the provided initial implementations are updated with them. Customized implementation means that the given service had to be rewritten to match the UML language as a generic solution was not possible.

- **Model server:** as the model server is the core component that enables simultaneous modeling, and its requirements foresee a generic implementation, it is used as is, and no modifications are performed on it.
- **Language server:** the generated UML language server already provides default implementations of the grammar, for syntax highlighting, validation, error handling, serialization, cross-references and workspace and scope management. To correspond to the requirements of the blended framework only the provided generic serializer must be updated to provide methods for serializing every element of the language as is and not just a part of a diagram.

¹<https://marketplace.visualstudio.com/items?itemName=BIGModelingTools.umlDiagram>

Framework component	Implemented functionality	Implementation method
Model server	Access	generic implementation
	Model provision	generic implementation
	Model synchronization	generic implementation
Textual modeling	Grammar	provided implementation
	Syntax highlighting	provided implementation
	Validation	provided implementation
	Error handling	provided implementation
	Serialization	provided implementation customized with generic node serialization
	Cross-references	provided implementation
	Workspace and scope management	provided implementation
Graphical modeling	Model synchronization	provided implementation
	Graphical model	provided implementation extended with generic missing nodes implementation
	Model editing	generic operation handlers; customized textual syntax modification methods
	Validation	provided implementation
	Error handling	generic implementation
Model synchronization	generic implementation	

Table 6.1: Implemented concepts of BIGUML blended modeling framework.

- **GLSP server:** the provided services must be customized to correspond to the concepts of blended modeling and enable model synchronization with the model server. Both the services of the package diagram and the class diagram have to be customized.
 - **Graphical model:** the provided `GModelFactory` is extended with the generic implementation of adding missing nodes to the graphical model which correspond to non-resolvable cross-references in the model.
 - **Model editing:** all of the provided action handlers have to be customized to use the methods of the model state for updating the textual syntax of the model. Furthermore, the methods that `insert` a new element, `update` an existing element and `delete` an existing element must be customized to fit the syntax of the generated UML grammar.

Method	Implementation details
insert	new element to root model: inserts element to the end of the textual model
	new child element: inserts element to the end of the list of the corresponding node's child elements
delete	replaces the textual syntax of the given node with an empty string in the textual model
update	replaces the old value of an attribute with the new one in the given node's textual syntax

Table 6.2: Implemented concepts for incremental textual syntax updates on the Workflow language.

- **Validation:** the provided generic implementation of the model validator is kept.
- **Error handling:** the generic concept for error handling in graphical modeling is applied: no modifications are allowed that would cause errors in the model, the non-resolvable cross-references are displayed on the diagram as missing nodes, and missing nodes are createable by generic actions.
- **Model synchronization:** the generic approach for model synchronization is applied, extending the model storage and model state with the generic methods for listening to changes coming from the model server and sending the updated textual syntax to the model server on diagram updates.

6.3.1 Graphical model editing

To enable incremental updates on the textual concrete syntax for operation handlers in the GLSP server, the `insertToSemanticText`, `deleteFromSemanticText`, and `updateInSemanticText` methods must be customized to match the syntax of the UML grammar.

The incremental updates for the Workflow language cover the cases shown in Table 6.2 and the incremental updates for the UML language must cover the cases listed in Table 6.3.

To showcase the implementations of the customized textual syntax update methods, Listing 6.1 shows the implementation of the `updateInSemanticText` method. The other update methods are based on the same idea.

Firstly, the comments are removed from the textual syntax, to not consider the text in the comments while searching for attributes or values in the textual syntax. Then, the attribute's position in the textual syntax is determined and the first element found starting from this position, which corresponds to the attribute's old value is replaced with the attribute's new value in the original text. It is necessary to determine the attribute's position in the text to make sure that the correct value is updated. Otherwise,

Method	Implementation details
insert node	every new element is inserted into an array, method must find the starting "[" of the correct array in the textual model and insert element after; array can also be nested into other elements/arrays
	first element of array: only insert serialized node to the found position
	array already contains elements: insert serialized node to the found position following with a "," to separate new node from other elements in the array
delete	delete from array with only one element: delete element
	first or n-th (but not last) element of array: delete element with a following ","
	last element of array: delete element with a prepending ","
update	replaces the old value of an attribute with the new one in the given node's textual syntax; must also consider nested attributes e.g., an element's id in a reference object.
insert attribute	insert a new attribute to an existing node, method must find the starting "{" of the correct node to add the attribute to; the attribute can also be nested into a child element
	first attribute of node: only insert serialized attribute to the found position
	node already contains attributes: insert serialized attribute to the found position following with a "," to separate new attribute from the other attributes of the node

Table 6.3: Implemented concepts for incremental textual syntax updates on the UML language.

a replace operation could replace another attribute's value if it is the same than the one that should be updated. Finally, the old textual syntax is replaced by the newly created one in the model's textual syntax.

```
1 updateInSemanticText (  
2   oldText: string,  
3   attributeName: string[],  
4   oldAttributeValue: string,  
5   newAttributeValue: string  
6 ): string {  
7   let newText = oldText;  
8   const cleanedText = removeAllComments(newText);  
9   if (typeof oldAttributeValue === "undefined") {  
10    // should also fill undefined string variables  
11    oldAttributeValue = "";  
12  }  
13  
14  // find the position of the attribute in the cleaned text  
15  let attributeIndex = regexIndexOf(  
16    cleanedText,  
17    new RegExp(`${attributeName.at(0)}\\s*:` , "g"),  
18    0  
19  );  
20  if (attributeName.length > 1) {  
21    // iterate through all the nested attributes of the element  
22    attributeName.slice(1).forEach((selector) => {  
23      attributeIndex = regexIndexOf(  
24        cleanedText,  
25        new RegExp(`${selector}\\s*:` , "g"),  
26        attributeIndex  
27      );  
28    });  
29  }  
30  
31  // replace old attribute value with new one  
32  newText = replaceStartingFrom(  
33    newText,  
34    oldAttributeValue,  
35    newAttributeValue,  
36    attributeIndex  
37  );  
38  this._semanticText = this._semanticText.replace(oldText, newText);  
39  // return updated text to ensure multiple updates in the same comment  
40  return newText;  
41 }
```

Listing 6.1: BIGUML GLSP server: updateInSemanticText method

The implemented BIGUML class diagram use case in the blended textual-graphical editor is shown in Figure 6.1. The left side shows the GLSP editor and the right side the corresponding textual editor. Both editors have the same model open, changes made in one editor are synchronized immediately with the other utilizing the model service.

6.4 Scenario Evaluation

After successful implementation of the package diagram and class diagram BIGUML scenarios, it must be proofed whether the generic concepts of the blended modeling

framework still hold on either of the scenarios. The following concepts are handled corresponding to the generic requirements of textual modeling, graphical model and the model server as listed in Section 4.2.

- Textual modeling
 - Model synchronization: whether the changes made in the textual editor are correctly synchronized with the graphical editor
- Graphical modeling
 - Graphical model: declared elements displayed correctly, missing nodes displayed for non-resolvable cross-references
 - Model editing:
 - * insert node: creating a new node in an empty container and in a non-empty container
 - * delete: deleting a node from an array only containing this node, deleting the first node of an array, deleting the last node of an array
 - * update: updating an attribute's value in the given node, updating an attribute's value in the given node's nested node
 - * insert attribute: insert new attribute to a node
 - Error handling: missing nodes are createable
 - Model synchronization: changes made in the graphical editor are correctly synchronized with the textual editor maintaining non-semantic information
- Model server
 - Access: whether the model is openable and can be accessed by both clients
 - Model provision: whether the model server provides the correct version of the model (including providing an empty model if there are non-linking errors)
 - Model synchronization: whether the model server synchronizes the model correctly between the GLSP server and the language server

The other requirements for textual modeling and graphical modeling stated in Section 4.2 are language-specific and either a default implementation was used in the Workflow framework or a Workflow language-specific solution was implemented. Therefore, for these requirements the generated default services are used in the BIGUML modeling tool, and these requirements will not be evaluated.

6.4.1 Evaluation results

Table 6.4 shows the results of the scenario evaluation. Both the package diagram and class diagram implementations were evaluated.

For textual modeling and the model server every requirement was met by the instantiated BIGUML use cases. Changes made in the text editor were instantly synchronized with the GLSP editor, the GLSP server was provided access to the model server's opened models, the GLSP server received the updates coming from the model server if the model did not have errors or only contained linking-errors and the updated textual syntax of the GLSP server got correctly sent to the model server updating the text editors content.

The graphical model, error handling and model synchronization requirements of graphical modeling were also met by the implemented use cases, and the insert node, delete, and update operations of the model editing requirement also met the predefined criteria. The insert attribute requirement however failed the evaluation for both use cases. The GLSP server inserted in both cases the new attribute on the predefined position of the provided note: directly after the node's opening bracket, as the first attribute of the node. However, the generated UML language foresees a strict order of attributes on the node, and in most of the cases the attribute to be inserted should not be on the first place.

To solve this problem without modifying the grammar, either the insert positions of each attribute of each type of node must be hard coded based on the grammar's rules, which is not feasible for the purposes of this evaluation, a validator rule for each type of node must be implemented for the language server to mark nodes as erroneous if not all of their attributes are defined or the update method must be rewritten to replace the complete node with a re-serialized updated node, which would remove the non-semantic information for this node.

6.5 Summary and Discussion

The evaluation of the blended textual-graphical modeling framework has shown that the implemented generic concepts for textual modeling, graphical modeling and the model server can be successfully applied for other modeling languages that are based on a structurally different grammar to enable simultaneous graphical and textual modeling on the same model.

The implemented generic services of the language server, the model server, and the GLSP server could be reused on a provided framework containing a Langium-based language server and a TypeScript-based GLSP server and client. On the GLSP server, solely the implemented methods for manipulating the textual syntax had to be customized to fit the syntax of the new grammar and the operation handlers of the GLSP server to make use of these methods to update the model's textual syntax. On the language server only the serializer had to be customized to not only provide a method for serializing the complete model, but also to provide a method for serializing individual elements.

Framework component	Functionality	Package Diagram	Class diagram	
Textual modeling	Model synchronization	✓	✓	
Graphical modeling	Graphical model	✓	✓	
	Model editing	insert node	✓	✓
		delete	✓	✓
		update	✓	✓
		insert attribute	✗	✗
	Error handling	✓	✓	
Model synchronization	✓	✓		
Model server	Access	✓	✓	
	Model provision	✓	✓	
	Model synchronization	✓	✓	

Table 6.4: Scenario evaluation: package and class diagram.

As the implementation of the evaluation scenarios are based on a Langium language server generated by Jäger's [Jä24] language server generator tool, the customized methods for manipulating the textual concrete syntax on the GLSP server could be reused for other languages generated by this tool, as they are based on the same JSON syntax.

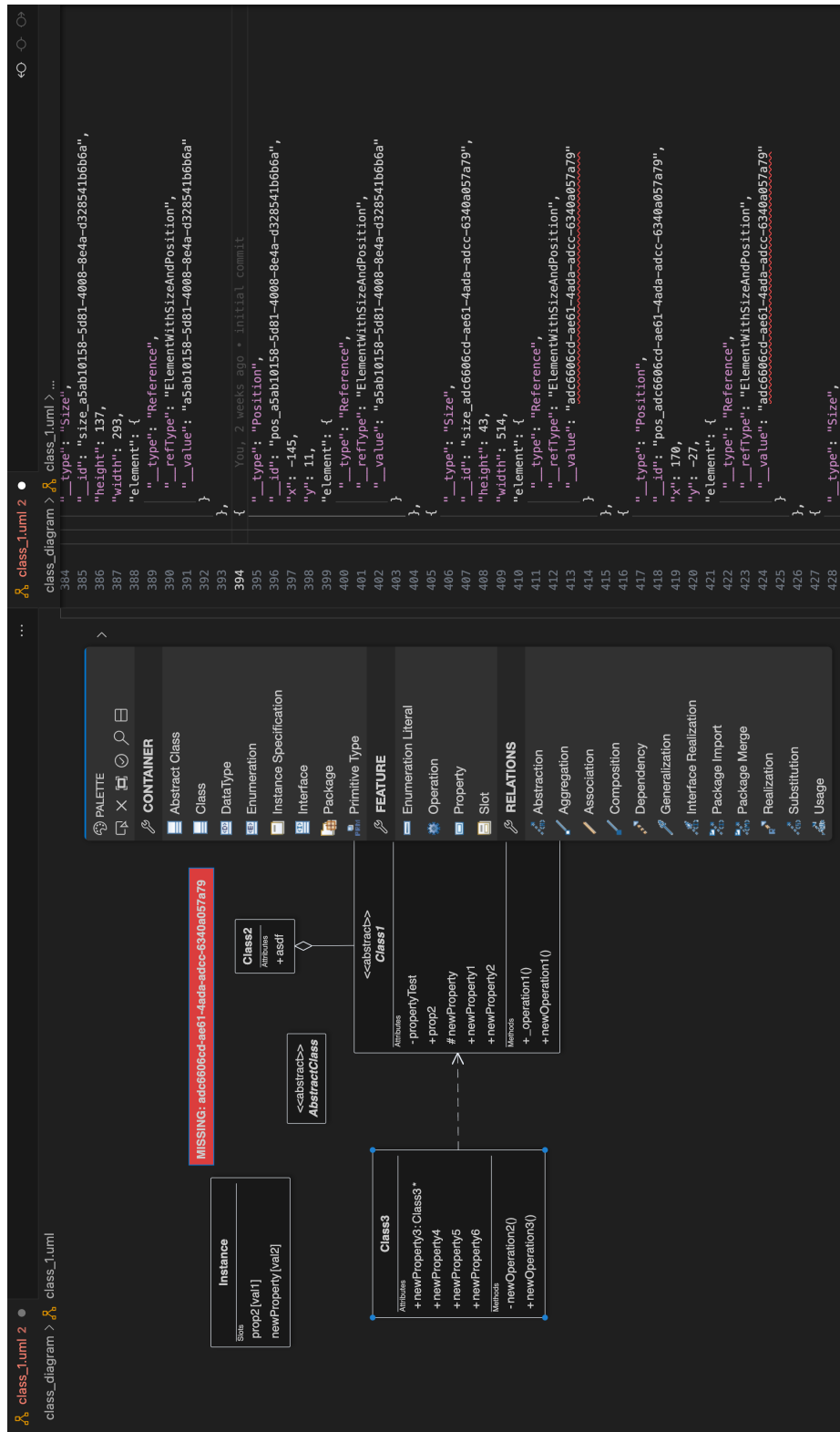


Figure 6.1: BIGUML scenario: an example class diagram model.

Conclusion

This final chapter summarizes the work done in this thesis, provides answers to the research questions stated in Section 1.2, and elaborates possibilities for future work.

7.1 Conclusion

The main goal of this thesis was to develop a concept on how blended textual-graphical modeling could be realized with the next generation frameworks Langium and GLSP utilizing a model service to jointly manage the underlying modification model, and to implement an artifact showcasing the established concept. The concept had to consider that the graphical and textual editor must operate on the same model, simultaneous updates must be possible between the two editors, and non-semantic information of the model must be maintained during updates on the model. Updates on the model must be delegated through the model service to other editors and the model service must keep track of the model's current state and be able to provide the current model to requesting clients.

After elaborating the framework's requirements and establishing a conceptual solution to them, an example artifact based on the requirements was established. The blended textual-graphical modeling artifact is based on the Workflow language, as for this language an example TypeScript-based GLSP framework already existed. This framework was extended by a Langium language server based on the Workflow language's grammar and a model server handling model access, provision and updates between the language server and the GLSP server. The GLSP server was also extended and customized to meet the defined requirements of blended modeling.

To evaluate the developed concept and artifact, the implemented solution concepts were instantiated with two UML use cases: the package diagram and the class diagram. The initial implementation of these use cases is based on Jäger's thesis' artifacts, which were

developed in parallel. The artifacts provide a UML-based Langium language server connected to a TypeScript-based GLSP framework called BIGUML. These artifacts were extended by the implemented artifacts for the blended textual-graphical framework and the generic requirements of the blended framework were verified on the extended BIGUML modeling tool. The evaluation of the blended textual-graphical modeling framework has shown that the implemented generic concepts for textual modeling, graphical modeling and the model server can be successfully applied for other modeling languages, needing only some services to be customized based on the used language's grammar.

With the implementation and evaluation of the concepts and artifacts, finally, the research questions stated in Section 1.2 can be answered.

1. Model service:

How can the model service allow textual and graphical editors to manage and manipulate the underlying abstract syntax tree (AST) jointly?

The implemented concept only allows textual and graphical editors to manipulate the underlying AST indirectly. Updates arriving from the graphical editor are in form of the textual concrete syntax of the model. The model server re-generates the underlying `LangiumDocument` based on the updated textual syntax of the model, and provides the updated AST from the `LangiumDocument` for the graphical editor. It also updates the content of a text editor if it has the given model open, with the updated textual syntax of the model.

To allow the textual editor to update the underlying AST of the model server, the model service listens to any updates coming from the language server. If the content of a textual editor was modified, the model service gets notified and again replaces the underlying `LangiumDocument` with the newly built one provided by the language server. The graphical editor listens to these updates and the model service provides the updated `LangiumDocument` containing the AST and the textual syntax of the model for the graphical editor if an update happens.

2. Modification model:

How to implement the modification model to allow for simultaneous modifications on the textual and graphical models?

The modification model of the GLSP server is the textual concrete syntax of the model, even though, the model index, the model state and the diagram elements are also based on the AST of the model. The updates on the model are solely executed on the textual syntax, from which the model service provides the GLSP server a re-generated updated AST.

The modification model of the text editor is also the textual concrete syntax as every update performed on the model in the text editor are direct edits on the model's textual

syntax. The language server generates a `LangiumDocument` from the textual syntax on every update, providing the updated AST and textual syntax for the model service and its connected clients.

3. Cross-references:

How can the Langium-GLSP framework resolve cross-references in the textual and graphical representations?

In the implemented Workflow framework and the evaluation scenarios the Langium language server resolves cross-references based on the framework's default cross-reference resolution mechanism with a customized scope to only resolve cross-references between files that belong to the same model, as it is unusual for graphical modeling editors to reference nodes in another model.

The GLSP server resolves cross-references based on the ID of the node provided in the reference's text in the AST. The GLSP server also only looks for nodes in the same diagram to resolve cross-references.

If a reference is not resolvable, the language server displays an error and the GLSP server displays a missing node with the given ID of the unresolvable cross-reference on the diagram. It is possible to create a node with this ID on both the textual and graphical model editors.

4. Non-semantic information:

How to handle non-semantic information with the Langium-GLSP framework for textual and graphical models?

Updating the textual syntax in the textual editor maintains non-semantic information as the user would explicitly have to delete comments or reformat the model to change the non-semantic information. The changes delegated via the model server to the GLSP server also contain the textual syntax with the maintained non-semantic information.

The updates on the diagram editor are explicitly performed on the textual syntax to maintain non-semantic information. The framework provides an example implementation of the language-specific `update`, `delete`, and `insert` methods, which manipulate the textual syntax of the model maintaining non-semantic information based on the Workflow language.

Even though inserting a new attribute on a node did not meet the requirements on the customized manipulation methods on the evaluation scenarios, possible solutions were provided to fix the arose problems.

7.2 Future Work

To conclude the thesis, this section presents possible future work and revisions that could improve the developed concepts and artifacts.

Firstly, the implemented model service is planned to be combined with David Jäger's implemented model service and generator tool [Jä24] to be released as an open-source framework within the EMF Cloud [Ecl]. The framework should also provide the implemented example artifacts for blended modeling including the Workflow modeling framework and the BIGUML modeling tool.

To improve the developed artifacts functionality-wise, several approaches could be investigated to also provide a generic implementation for model updates on the GLSP server. One idea could be to customize Langium's AST generation to not only provide the textual syntax for each node but also for attributes. This way, updating the textual syntax could be simplified significantly as the position of each node and their attributes would be known and could be updated directly or the position of where to insert or delete elements from would also be straightforward. Another idea to generalize model updates on the GLSP server would be to investigate updates in form of manipulating the node's AST instead of its textual syntax. This way, the model service would have to serialize the provided AST and re-inject the lost non-textual information after serialization.

Finally, another idea for improvement would be to combine the concepts developed by David Jäger and this thesis. The generator tool developed in his work generates a Langium language server based on a TypeScript-based interface declaration file corresponding to the language's grammar, and his version of the model server operates with JSON patches as model updates, as the generated grammar corresponds to the JSON format. As comments and other non-semantic information are not part of the JSON structure, the model service always re-serializes the current textual model to create a correct JSON structure and apply the patch on it. It would be interesting to investigate how the JSON patches could be extended to maintain non-semantic information of the textual model on GLSP updates, eventually utilizing the JSON5 [jso] format instead of the common JSON format.

List of Figures

1.1	VS Code's Monaco editor: An example of a textual model editor.	2
1.2	GLSP Client: An example of a graphical model editor.	3
2.1	Metamodeling: 'conformsTo' and 'instanceOf' relationships [BCW17]. (page 15)	11
2.2	GLSP: client-server architecture [Foua]	13
2.3	Creation of LangiumDocuments [Typb]	15
2.4	Stages of a LangiumDocument [Typb]	15
3.1	Excalibur: an Xtext-Sirius framework for read-only graphical representation. [RCG18]	19
3.2	Langium meets Sprotty: graphical representation of a textual model using next-generation frameworks [Pet22]	20
3.3	Xtext / Sirius - Integration: Embedding an Xtext Editor into Sirius [Obe17]	21
3.4	Langium + Sirius Web: simultaneously editing the same model graphically and textually [Gir22]	22
3.5	The BIGER modeling tool based on Xtext and Sprotty [GB21]	22
3.6	Blended modeling framework based on Xtext and Papyrus: simultaneously editing the same resource graphically and textually [ACLP17]	23
4.1	Architecture of the Langium-GLSP blended modeling framework	31
5.1	Command palette	71
5.2	Tool palette	71
5.3	Error handling in the graphical editor	77
5.4	Updating the internal state from a model server client	79
5.5	Updating the internal state from the language server	79
5.6	VS Code extension: the blended framework	87
6.1	BIGUML scenario: an example class diagram model.	98

List of Tables

3.1	Comparison of the listed frameworks.	25
5.1	Implemented concepts of the blended modeling framework.	86
6.1	Implemented concepts of BIGUML blended modeling framework.	91
6.2	Implemented concepts for incremental textual syntax updates on the Workflow language.	92
6.3	Implemented concepts for incremental textual syntax updates on the UML language.	93
6.4	Scenario evaluation: package and class diagram.	97

List of Listings

5.1	Workflow grammar	42
5.2	Custom validator: no duplicate names	47
5.3	Code action: create missing nodes	48
5.4	Serializer	51
5.5	Workspace Manager	53
5.6	Scope provider	54
5.7	Scope computation	55
5.8	Name provider	56
5.9	Name utility	57
5.10	Workflow model storage: loadSourceModel	59
5.11	Workflow model state	61
5.12	Workflow model index	64
5.13	Workflow GModel factory - creating a task node	66
5.14	Workflow GModel factory - creating missing nodes	67
5.15	Delete operation handler	71
5.16	Command palette action provider	73
5.17	Model server: OpenTextDocumentManager's update method	81
5.18	Model server: replaceTextEditorContent method	82
5.19	Model server: onUpdate listener	83
6.1	BIGUML GLSP server: updateInSemanticText method	94

Acronyms

AST abstract syntax tree. 4, 12, 15, 16, 18, 19, 29, 30, 32–38, 42, 44–46, 48, 51–53, 56, 58, 60, 63–65, 69, 73, 75, 79, 80, 84, 86, 90, 100–102

DSL domain-specific language. 6, 14, 15, 17–19, 39, 41, 45, 69, 75, 84

ER Entity-Relationship. 23

GLSP Graphical Language Server Platform. xii, 3–6, 9, 12, 13, 16, 24, 25, 27, 28, 30–35, 37–39, 41, 42, 45, 53, 56–61, 63, 65, 67–73, 75–79, 81, 84–86, 89–92, 94–97, 99–103, 107

LSP Language Server Protocol. 12, 16, 44, 48, 78, 80, 83, 84

UML Unified Modeling Language. 11, 23, 89–93, 96, 99, 100, 105

URI Uniform Resource Identifier. 50, 54, 59, 64, 83, 84

Bibliography

- [AC21] Lorenzo Addazi and Federico Ciccozzi. Blended graphical and textual modelling for uml profiles: A proof-of-concept implementation and experiment. *Journal of Systems and Software*, 175:110912, 2021.
- [ACLP17] Lorenzo Addazi, Federico Ciccozzi, Philip Langer, and Ernesto Posse. Towards seamless hybrid graphical–textual modelling for uml and profiles. In Anthony Anjorin and Huáscar Espinoza, editors, *Modelling Foundations and Applications*, pages 20–33, Cham, 2017. Springer International Publishing.
- [AG13] Colin Atkinson and Ralph Gerbig. Harmonizing textual and graphical visualizations of domain specific models. In *Proceedings of the Second Workshop on Graphical Modeling Language Development*, GMLD ’13, page 32–41, New York, NY, USA, 2013. Association for Computing Machinery.
- [BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice: Second Edition*. Morgan & Claypool Publishers, 2nd edition, 2017.
- [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.

- [Bor] Bork, Dominik. bigUML. <https://github.com/borkdominik/bigUML>. [Online; accessed 01-Feb-2024].
- [CGR09] María Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within modeling language definitions. pages 670–684, 10 2009.
- [CK19] Justin Cooper and Dimitris Kolovos. Engineering hybrid graphical-textual languages with sirius and xtext: Requirements and challenges. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 322–325, 2019.
- [CLG⁺10] Baojiang Cui, Jiansong Li, Tao Guo, Jianxin Wang, and Ding Ma. Code comparison system based on abstract syntax tree. In *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*, pages 668–673, 2010.
- [Croat] CrossBreeze. CrossBreeze. <https://crossbreeze.nl>. [Online; accessed 15-Feb-2024].
- [Croat] CrossBreeze. crossmodel. <https://github.com/CrossBreezeNL/crossmodel/tree/main/extensions/crossmodel-lang/src/model-server>. [Online; accessed 15-Feb-2024].
- [CTVW19] Federico Ciccozzi, Matthias Tichy, Hans Vangheluwe, and Danny Weyns. Blended modelling - what, why and how. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 425–430, 2019.
- [DLP⁺22] Istvan David, Malvina Latifaj, Jakob Pietron, Weixing Zhang, Federico Ciccozzi, Ivano Malavolta, Alexander Raschke, Jan-Philipp Steghöfer, and Regina Hebig. Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study. *Software and Systems Modeling*, 22, 06 2022.
- [Eclat] Eclipse Foundation. Eclipse GMF Runtime™. <https://projects.eclipse.org/projects/modeling.gmf-runtime>. [Online; accessed 08-Feb-2024].
- [Eclb] Eclipse Foundation. Eclipse Papyrus™. <https://eclipse.dev/papyrus/>. [Online; accessed 08-Feb-2024].
- [Eclc] Eclipse Foundation. Emf cloud. <https://eclipse.dev/emfcloud/>. [Online; accessed 18-Mar-2024].
- [Eclat] Eclipse Foundation. Sirius web. <https://eclipse.dev/sirius/sirius-web.html>. [Online; accessed 08-Feb-2024].

- [Ecle] Eclipse Foundation. Sprotty. <https://sprotty.org>. [Online; accessed 08-Feb-2024].
- [Eclf] EclipseSource. Workflow extension. <https://github.com/eclipse-glsp/glsp-vscode-integration/blob/master/example/workflow/extension/src/workflow-extension.ts>. [Online; accessed 08-Mar-2024].
- [Foua] Eclipse Foundation. Eclipse GLSP. <https://eclipse.dev/glsp/>. [Online; accessed 01-Feb-2024].
- [Foub] Eclipse Foundation. Eclipse modeling framework (EMF). <https://eclipse.dev/modeling/emf/>. [Online; accessed 01-Feb-2024].
- [Fouc] Eclipse Foundation. Eclipse sirius. <https://projects.eclipse.org/projects/modeling.sirius>. [Online; accessed 02-Feb-2024].
- [Foud] Eclipse Foundation. Workflow example overview. <https://eclipse.dev/glsp/examples/#workflowoverview>. [Online; accessed 01-Feb-2024].
- [Foue] Eclipse Foundation. Xtext. <https://eclipse.dev/xtext/>. [Online; accessed 01-Feb-2024].
- [GB21] Philipp-Lorenz Glaser and Dominik Bork. The bigger 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.
- [Gir22] Théo Giraudet. Langium + sirius web = heart. <https://blog.obeo.com/langium-sirius-web>, 2022. [Online; accessed 08-Feb-2024].
- [GMGC22] Joan Giner-Miguel, 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.
- [Hev07] Alan Hevner. A three cycle view of design science research. *Scandinavian Journal of Information Systems*, 19, 01 2007.
- [HR00] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical report, ISR, 2000.
- [HR04] D. Harel and B. Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72, 2004.
- [HRM⁺04] Alan Hevner, Alan R, Salvatore March, Salvatore T, Park, Jinsoo Park, Ram, and Sudha. Design science in information systems research. *Management Information Systems Quarterly*, 28:75–, 03 2004.

- [jso] json5. Json5 – json for humans. <https://json5.org>. [Online; accessed 18-Mar-2024].
- [Jä24] David Jäger. Frontend-only browser-based modeling tools. 2024.
- [Kü06] Thomas Kühne. Matters of (meta-) modeling. *Software Systems Modeling*, 5:369–385, 12 2006.
- [MB23a] Haydar Metin and Dominik Bork. Introducing BIGUML: A flexible open-source glsp-based web modeling tool for UML. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023 Companion, Västerås, Sweden, October 1-6, 2023*, pages 40–44. IEEE, 2023.
- [MB23b] 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 Extension API. <https://code.visualstudio.com/api>. [Online; accessed 15-Feb-2024].
- [Micb] Microsoft. Language Server Protocol. <https://microsoft.github.io/language-server-protocol/>. [Online; accessed 10-Feb-2024].
- [Micc] Microsoft. Modeling SDK for Visual Studio. <https://learn.microsoft.com/en-us/visualstudio/modeling/modeling-sdk-for-visual-studio-domain-specific-languages?view=vs-2022>. [Online; accessed 02-Feb-2024].
- [Midd] Microsoft. Monaco - The Editor of the Web. <https://microsoft.github.io/monaco-editor/>. [Online; accessed 16-Feb-2024].
- [Mice] Microsoft. Refactoring source code in Visual Studio Code. <https://code.visualstudio.com/docs/editor/refactoring>. [Online; accessed 23-Feb-2024].
- [Obe17] Xtext / Sirius Integration - White Paper. https://www.obeodesigner.com/resource/white-paper/WhitePaper_XtextSirius_EN.pdf, 2017. [Online; accessed 08-Feb-2024].
- [Obj] Object Management Group®. About the unified modeling language specification version 2.5.1. <https://www.omg.org/spec/UML/2.5.1/About-UML>. [Online; accessed 05-Feb-2024].

- [Pet22] Jette Petzold. Langium meets sprotty: Combining text and diagrams in vs code. <https://www.typefox.io/blog/langium-meets-sprotty-combining-text-and-diagrams-in-vs-code>, 2022. [Online; accessed 08-Feb-2024].
- [RCG18] Benoît Ries, Alfredo Capozucca, and Nicolas Guelfi. Messir: A text-first dsl-based approach for uml requirements engineering (tool demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018*, page 103–107, New York, NY, USA, 2018. Association for Computing Machinery.
- [Sch08] Markus Scheidgen. Textual modelling embedded into graphical modelling. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture – Foundations and Applications*, pages 153–168, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [Sel03] Bran Selic. Selic b.: The pragmatics of model-driven development. *iee softw.* 20(5), 19-25. *Software, IEEE*, 20:19 – 25, 10 2003.
- [SL] Miro Spönemann and Philip Langer. Combining textual graphical editors. <https://docs.google.com/presentation/d/1JLevi168Jmm03Wop4DVCCy2d-xUyCxURHBOVu3GMu10>. [Online; accessed 26-May-2023].
- [Tha22] Bernhard Thalheim. Models: the fourth dimension of computer science. *Software and Systems Modeling*, 21:1–10, 02 2022.
- [Typa] TypeFox GmbH. Langium. <https://langium.org>. [Online; accessed 01-Feb-2024].
- [Typb] TypeFox GmbH. Langium - Document Lifecycle. <https://langium.org/docs/document-lifecycle/>. [Online; accessed 10-Feb-2024].
- [Typc] TypeFox GmbH. Langium grammar language. <https://github.com/eclipse-langium/langium/blob/main/packages/langium/src/grammar/langium-grammar.langium>. [Online; accessed 05-Feb-2024].
- [vRWS⁺13] Oskar van Rest, Guido Wachsmuth, Jim R. H. Steel, Jörn Guy Süß, and Eelco Visser. Robust real-time synchronization between textual and graphical editors. In Keith Duddy and Gerti Kappel, editors, *Theory and Practice of Model Transformations*, pages 92–107, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [Wil97] David S. Wile. Abstract syntax from concrete syntax. In *Proceedings of the 19th International Conference on Software Engineering, ICSE '97*, page 472–480, New York, NY, USA, 1997. Association for Computing Machinery.

[Yeo] Yeoman. Yeoman | The Web's Scaffolding Tool For Modern Webapps.
<https://yeoman.io>. [Online; accessed 22-Feb-2024].