JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Accepted for the ECMFA 2025 conference and publication in the JOT journal.
This is the camera-ready author version of the paper, the final version will be accessible via JOT.

A Model Management Framework for Next-Generation Web-based Modeling Tools

David Jaeger[*], **Adam Lencses**[*], **Martin Fleck**[‡], **Philip Langer**[‡], **and Dominik Bork**[*]
[*]Business Informatics Group, TU Wien, Austria
[‡]EclipseSource Services GmbH, Vienna, Austria

ABSTRACT The move towards web technologies has arrived at modeling and especially modeling tools. Several frameworks and platforms have been proposed recently, aiming at the efficient realization of web-based modeling tools, i.e., tools developed with web technologies that natively allow for cloud-based deployment and use in web browsers. Most of these frameworks separate modeling tool functionality like editing, rendering, and management of the model into separate specialized components, often realized in different technologies. For the model management of Ecore-based modeling languages like the UML, the Eclipse Modeling Framework (EMF) is still prevalent. While this may maximize reuse, it comes at a cost: EMF is Java-based, entailing a polyglot technology stack, which complicates the development and hinders an entirely browser-based deployment without any backend. In this paper, we address this problem by introducing a Typescript-only approach for generating Langium-based modeling language grammars and model management servers. Using our approach, developers can build next-generation web-based modeling tools with a homogeneous technology stack, which can be executed as cloud applications or plain browser applications without any backend, if needed. We evaluate our approach by applying it to two distinct modeling tools: a Workflow and a UML modeling tool.

KEYWORDS Modeling tools, Web modeling, Langium, LSP, GLSP, UML.

In recent years, we have observed an increasing interest in the modeling community in bringing some of their powerful platforms and frameworks (Kelly et al. 1996; Steinberg 2009; Jarke et al. 1995) into the web, or to re-invent them using web technologies (Bainczyk et al. 2022)–exemplary endeavors are Sirius Web and EMF Cloud. This movement is ongoing and motivated by the expected increase of flexibility and usability such web-based modeling tools would unlock, compared to their full-fledged, powerful, and currently still widely used fat-client alternatives (Bork et al. 2023; Rodríguez-Echeverría et al. 2018a,b).

Many of the existing approaches managed to shift the frontend onto web technologies, including the rendering of diagrams and user interfaces for editing, with novel technologies, such as Sprotty (Eclipse Foundation 2024; Petzold 2022) and GLSP (EclipseSource 2024). However, those approaches typically still rely on the prevalent Java-based frameworks, such as EMF (Eclipse Foundation 2024a; Steinberg 2009), in the backend for specifying the modeling language and for the model management, including the (de-)serialization of models, handling of cross-references, and validation. While this maximizes the reuse of existing modeling framework capabilities and language implementations, the dependency on Java increasingly becomes an issue, as it *(i)* leads to a heterogeneous developer environment complicating the development, *(ii)* entails additional runtime requirements if, for instance, shipped as a VS Code extension, and, most importantly, *(iii)* prevents a deployment as plain browser application, in which the modeling tool is hosted as a static website without the need for a backend. However, this form of deployment becomes increasingly important given the cost and IP considerations with a cloud infrastructure.

With the same motivation, Langium (langium 2024) has

JOT reference format:
David Jaeger, Adam Lencses, Martin Fleck, Philip Langer, and Dominik Bork. *A Model Management Framework for Next-Generation Web-based Modeling Tools.* Journal of Object Technology. Vol. 24, No. 2, 2025.

http://dx.doi.org/10.5381/jot.2025.24.2.a1

been recently released as a Typescript-native textual language development framework alternative to Xtext (Eclipse Foundation 2024). Besides enabling the efficient development of new textual languages, Langium also provides solutions for many complex problems that are relevant for model management, including parsing language files into semantic models represented as Typescript interfaces (i.e., the abstract syntax tree, AST), validating these models against language rules, and managing cross-references within a workspace. However, as Langium is a textual language framework strongly aligned with the Language Server Protocol (Bork & Langer 2023), it only focuses on textual editing, i.e., changing text documents in a two-dimensional space defined by row and character position. Therefore, the question arises whether the features of Langium can be made available for model-oriented use cases, enabling modeling tools to benefit from Langium's rich capabilities in model serialization, managing cross-references, and validating models.

When looking at existing solutions like GLSP and Langium, we can observe, that partial solutions for fully web-based modeling tools exist. GLSP supports the development of Typescript-based modeling editors but lacks full model management support using web technologies (i.e., currently only offering a Java-based model server). On the other hand, Langium is rich in functionality to automatically generate model management for textual languages. A gap thus remains in leveraging the model management functionality of Langium for graphical modeling languages and editors. This research consequently explores the possibilities of expanding the functionalities of Langium to provide model-oriented clients access to its AST model and use it as a Typescript-native model management framework for modeling tools, replacing crucial functionalities that would otherwise require traditional Java-based frameworks, such as EMF. This paper makes conceptual and technical contributions to the further development of new and the improvement of existing modeling tools, a very important pillar of modeling research (Paige & Cabot 2024; Michael et al. 2023).

We enhance Langium with a dedicated model management service and a corresponding API to which model-oriented clients can connect and manipulate the model state. As Langium is written in Typescript and can be packaged for the browser without a dependency on NodeJS, complete model management can be handled inside the browser. Access to the local file system can be implemented to load model files by using the Browser File System API[1] or other external services.

We further introduce a Typescript-native language to simplify the definition of the metamodel. This metamodel definition language is based on Typescript interfaces which are extended to capture necessary meta-information of metamodel elements by custom annotations (e.g., to signify cross-references). Taking such a Typescript-native metamodel definition language as input, we generate a generic JSON grammar from these type definitions, and, ultimately, a Langium-based model management server.

In the remainder of this paper, we first present the relevant background in Section 1. Our Typescript-based grammar specification approach is introduced in Section 2. The generation of the Langium-based model management service is described in Section 3 and evaluated and discussed in Section 4. Section 5 then sheds light on the implications of our work toward the development of next-generation web-based modeling tools. We close this paper with a conclusion in Section 6.

[1] https://developer.mozilla.org/en-US/docs/Web/API/FileSystem

1. Background

1.1. Web-based Modeling

Several approaches exist to develop modeling language grammars and model management servers using web technologies (Maróti et al. 2014; Syriani et al. 2013; Rocco et al. 2023; Lafontant & Syriani 2020). The majority of these approaches focus on either the frontend (i.e., model representation and user interaction like in Sprotty (Eclipse Foundation 2024; Petzold 2022), EMF.cloud (Eclipse Foundation 2024b), Sirius Web (Eclipse Foundation 2024c), ReactFlow (xyflow 2024), ReactDiagrams (Storm 2024), JointJS (client.IO 2024)), require a Java-based backend (like in the EMF ecosystem (Steinberg 2009)), exclude model management (like in GLSP (EclipseSource 2024)), or are constrained to textual languages (like Langium (langium 2024)). Emerging tools like Gentleman (Lafontant & Syriani 2020) and jjodel (Rocco et al. 2023) offer the promise of providing fully web-technology-based solutions to the development of web modeling tools.

With respect to the modeling language definition, also various approaches exist that have been adopted over time. Approaches exist to define DSLs by creating a grammar (Jézéquel et al. 2011) or by defining metamodels in a textual, graphical, or dialogue-driven manner (Steinberg 2009; Bork et al. 2020). Additionally, EMF can be integrated with Xtext (Eclipse Foundation 2024), which enables the creation of DSLs using a Java technology stack. Within the Eclipse ecosystem, models can also be managed within the browser through EMF.cloud (Eclipse Foundation 2024b). However, this solution only brings the frontend into the cloud, still requiring a Java server for the model management; therefore, the tool can still not run in a browser-based environment without a backend. In (Giner-Miguelez et al. 2022), a DSL and a supporting tool to describe machine learning datasets is introduced. The authors extend Langium by a few services to realize initial model management.

1.2. GLSP

The Graphical Language Server Platform (GLSP) (Metin & Bork 2023b) is an extensible open-source framework for building custom diagram editors with web technologies (eclipseglsp-website 2024). GLSP is based on an extensible client-server architecture and comes with four major components (Metin & Bork 2023b): *GLSP Server*, *GLSP Client*, *Platform Integration*, and *Model Management*. The client-server communication is based on a variant of the Language Server Protocol (lsp 2024; Bork & Langer 2023), which is extended to provide features required for graphical modeling. This way, the client only has to cope with the rendering of the graphical model and providing possibilities for the users to interact with the model, while the server handles the more computationally heavy tasks like modifying the underlying AST of the diagram, loading the diagram, and handling user actions performed on the client.

The GLSP server can be written in any programming language because there is a clearly defined protocol between the server and the client–available servers are programmed in Java and Typescript. The GLSP server loads the source model, which can be in an arbitrary format, e.g., in JSON or a parsed AST from Langium, and stores it in the model state. It then creates the graphical model from the source model. The graphical model contains all the elements that the client will display and all their necessary attributes, e.g., size, position, element type, and label. This graphical model is then serialized by the server and sent to the client. The server also provides action handler implementations for actions that were dispatched from either the client or the server itself. The actions can modify the model state directly. After every modification to 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 realized as a lightweight web-based editor.

The client focuses on and is responsible for rendering the diagram and providing editing operations for the diagram, e.g., CRUD model operations, manipulating the size or position of the elements, or renaming labels. The client requests information from the server regarding the possible operations on each distinct model element. Using the information provided by the server, the client can then provide the editing tools for the different kinds of model elements.

To provide customizability and extensibility, the GLSP server and client both use an inversion of control pattern based on dependency injection, DI (eclipseglspserverdigraphics 2024). On the server, all of the provided services and components are placed in a global DI container and can be either extended or completely overwritten. While the GLSP Client is realized with web technologies like CSS and SVG, the GLSP Server, with open-source implementations available in Java and Typescript, lacks proper model management capabilities.

1.3. 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." (langium 2024) Langium provides the possibility to create DSLs together with an out-of-the-box Typescript-based language server that can be integrated into VS Code as an extension or other web applications and can be arbitrarily customized to meet the language creators' needs. With its pre-built implementations, Langium simplifies language tasks such as parsing, AST generation, validation, scoping, cross-referencing, and more. The Langium framework is built on DI. All its default services and other framework components can be arbitrarily customized, completely replaced, or extended.

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

The *LangiumDocument* is the main data structure of the language server that represents a text document written in the specified language. The *LangiumDocumentFactory* creates a *LangiumDocument* utilizing the *LangiumParser*, which parses the text document based on the created grammar. After a *LangiumDocument* was parsed and created, it needs to be built by the *DocumentBuilder* service, resulting in the AST of the model including resolved cross-references and validation errors. This workflow allows Langium to act as a language server that communicates with a client using the LSP. We refer the reader to the Langium documentation for a comprehensive overview[2]

1.4. Synopsis

To the best of our knowledge, no adequate mature solution for efficiently realizing model management in the browser using exclusively web technologies exists yet. Primarily, we aim for a solution that is extensible, interoperable, and built with community-driven frameworks like Langium and standardized programming languages like Typescript. The reason for this constraining is that other approaches often do not excel from academic proof-of-concept prototypes. Proper browser-based model management is desired, as it would enable the shift of the entire modeling tool into the browser, omitting any installation steps. The usage of Java is one of the main obstacles on the path toward browser-based modeling tools. To address these shortcomings, we propose a novel grammar specification (Section 2) and model management generation (Section 3) approach that is based on Typescript and Langium.

2. Typescript-based Grammar Specification

Next, we first introduce the requirements for the Typescript-based grammar specification before presenting the realized concept. The requirements were derived from analyzing existing metamodeling languages like Ecore and the structure of Langium's grammar definition as this will be the target platform.

2.1. Requirements

The grammar definition language has to provide a notation to define ...

- **Root/Entry element definition**, what shall be the root/entry element of a model;
- **Element definition** for model elements;
- **Attribute definition** for model elements;
- **Inheritance definition** between model elements;
- **Multiplicity definition** for attributes (i.e., *exactly one*, *zero or one*, *zero or more*, and *one or more*);
- **Reference definition** for *containment references* (i.e., references to elements contained within the referencing element) and *cross-references* (i.e., references to other, independent elements within the model); and
- **Type alias definition**;
- **Grammar validation**: to ensure a valid representation of the grammar and a transformation into a corresponding Langium grammar definition.

[2] Langium documentation: https://langium.org/docs/reference/grammar-language/

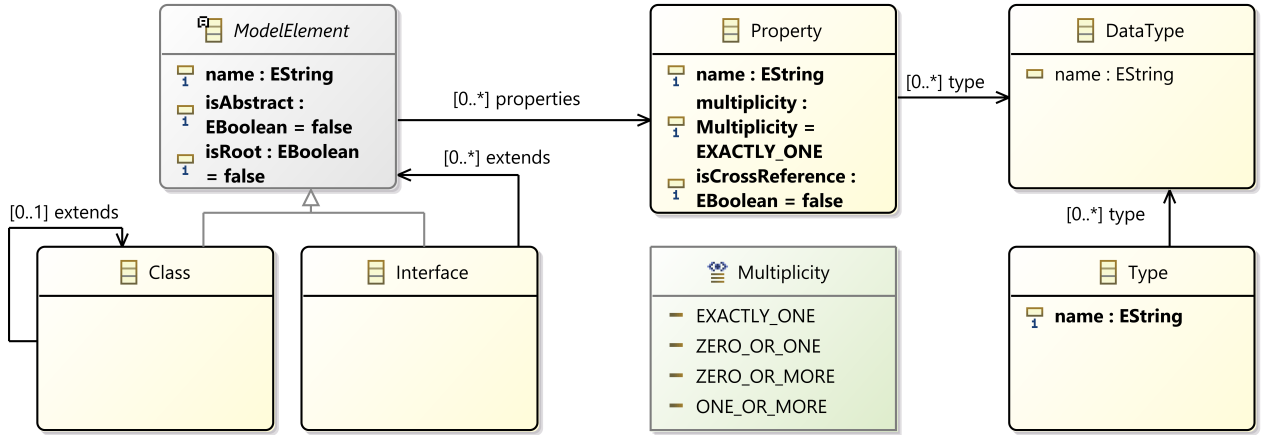

Figure 1 Visualization of the Typescript-based grammar language's metamodel in Ecore

2.2. Language Concepts

In the definition of the language concepts, it has to be ensured that all defined requirements are fulfilled while natively using Typescript. The metamodel of the Typescript-based grammar language is visualized in Figure 1. It consists of *Types* and *ModelElements*, which can be either of type *Class* or *Interface*[3]. *ModelElements* can have multiple *properties*, while *Types* can have multiple *DataTypes*. It has to be noted, that the *DataType* represents built-in data types (like *string* or *number*), constant data types (like instances of strings or numbers), and complex data types, including *ModelElements* and *Types*.

Next, we illustrate the concepts of our Typescript-based grammar specification language by iteratively rebuilding the metamodel in Figure 2. The metamodel relates to a university domain where a *University* is composed of *Rooms*, offers *Courses* which are delivered by a *Professor* and taken by *Students*. This example aims to be complex enough to show the expressivity of our Typescript-based grammar specification approach while remaining compact enough to fit the paper size.

Listing 1 shows how model elements can be defined in our new grammar language using the Typescript-native keywords `class` and `interface`. Attributes add semantics to model elements are defined in our grammar by creating properties and their multiplicity inside the model element definition. The interface *Person* shows how the four kinds of multiplicity are handled within the Typescript-based grammar language. The *name* property represents an element with multiplicity *exactly one*, as next to the property's name and the property's type no additional notations are used. The *title* property represents the multiplicity type *zero or one*, signaled by the Typescript native optional (*?*) operator after the property's name. The *firstNames* property defines an attribute with the multiplicity *one or more*, as its type is a container of type Array. Finally, the *nickNames* property represents an attribute with the multiplicity *zero or more* by combining the use of the optional operator (*?*) with a container of type Array.

[3] Interfaces and classes can be used interchangeably. As Typescript does not support multiple inheritance, using interfaces enables extending multiple other interfaces, adding support for multiple inheritance as is being used in Ecore.

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

Listing 1 Definition of model elements and their attributes

The grammar needs a notation to create containment references and cross-references to other model elements. Containment references can be created using the default Typescript notation for properties (see Listing 2). For cross-references in model elements with the *class*-keyword, the decorator *@crossReference* has been created. As *interfaces* do not support decorators, a custom container type (i.e., *CrossReference<T>*) has been created to enable the definition of cross-references for model elements defined as *interfaces*. Both solutions are shown in Listing 3.

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

Listing 2 Definition of containment reference

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

Listing 3 Definition of cross-reference

Inheritance is natively supported in Typescript by the *extends* keyword. The Typescript-based grammar definition uses this definition as can be seen in Listing 4.

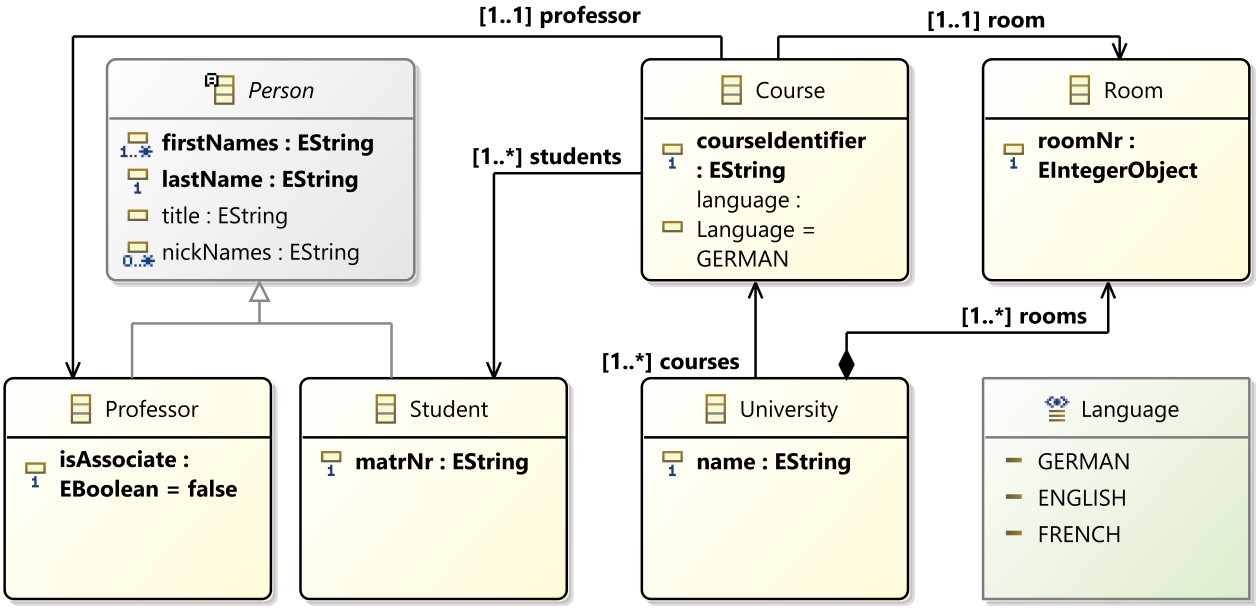

Figure 2 Visualization of a university domain metamodel in Ecore

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

Listing 4 Definition of sub-model elements

To define base elements, which can not be instantiated in the language but should provide an initial structure, it is possible to create abstract model elements. For *classes*, the *abstract* keyword can be used. However, for *interfaces*, this keyword is not supported. Therefore, a type named *ABSTRACT_ELEMENT* has been introduced, which signals for a definition of a model element using the *interface* keyword to be abstract. Listing 5 shows the definition of abstract model elements using both notations.

```
abstract class Person {}
interface Person extends ABSTRACT_ELEMENT {}
```

Listing 5 Definition of abstract model elements

Another requirement for the grammar language is the ability to create type alias elements, which can be done by the creation of union types. An example of these union types used in a Typescript-native way can be seen in Listing 6.

Finally, to define the root level structure of a model, a decorator named *@root* and a type named *ROOT_ELEMENT* has been introduced, which enable a class or an interface to be the root element, respectively. Listing 7 shows how the root element can be defined using either of the two definitions.

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

Listing 6 Definition of type alias element

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

Listing 7 Definition of root model element using class

Each grammar defined by our Typescript-based approach is validated in a twofold manner. First, it must consist of exactly one root element. Without such a root element, it is unclear how the model can contain all defined model elements. Second, it has to be serializable to enable model transformation between different systems.

3. Model Management Generation

We now investigate to what extent a generator can be realized that uses our previously introduced Typescript-based metamodel definition as an input to generate a Langium-based model management server. We first discuss the requirements and propose a concept for such a generator. Afterward, the API of the generated model management server is presented to show how it can be used and integrated into model editors.

3.1. Requirements

The main requirements for the model management generator are the following:

R1 **Initial project properties** The generator has to include an option to set the initial project properties like the name, language name, file extension, and root element.

R2 **Parsing** Parsing is required to be able to read the Typescript-based grammar definition and other configuration files. Furthermore, to ease the transition from Ecore to our approach, an `.ecore` parser would be beneficial.

R3 Validation & Transformation All parsed and generated files must be validated. This includes the *Typescript-based grammar definition*, the *LangiumDeclaration*, and the *LangiumGrammar* files. Once validation is passed, the transformators should be able to generate different types of files.

R4 File creation The file creation includes two types of files: either they can be predefined using template files, or they need to be created from scratch based on the generator's inputs. The files that need to be created from scratch include the Langium grammar definition (`.langium` file) and Langium services, which are written in Typescript and include, for example, the serializer service. These files also need to be recreated upon every change in the Typescript-based grammar definition.

R5 Package installation & build The generator has to provide the functionality to install the required packages and build the generated model management project.

3.2. Generator Concepts

Figure 3 illustrates the steps involved in generating the model management from the Typescript grammar specification. In the following, we will walk through these steps and describe the generator comprehensively.

3.2.1. R1. Initial project properties

The related concept includes the definition of which properties should be gathered in the generator's first execution. The most important properties are the project name, language name, and file extension. These values should be collected using prompting.

3.2.2. R2. Parsing

For the parsing feature, a concept for three types of parsers is required: A JSON, Ecore, and a Typescript parser. As JSON format is a default structure, JSON files can be parsed by reading the data into a variable. A more enhanced parsing functionality must be implemented for Ecore and Typescript files. The fast-xml-parser npm package is utilized to parse the Ecore definition into the data structure *EcoreDefinition* which can be seen in Listing 8. In this definition, the *classes* property collects the information about the features of *EClasses* and their *EReferences* and *EAttributes*, while the *dataTypes* property is used to store the parsed elements of type *EDataType* and the *types* property to collect union types (i.e., *EEnums*).

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

Listing 8 Defintion of the data structure used for the entire Ecore definition

For the parsing of Typescript files, the typescript npm package is used, which parses a Typescript definition into its AST representation. In the parser functionality of the generator, this AST is traversed, and the relevant data is stored in the data structure that can be seen in Listings 9 and 10.

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

Listing 9 Data structures used by the parser - `Declaration`

```
export interface Property {
  name:string;
  isOptional:boolean;
  decorators:string[];
  types:Type[];
  multiplicity:Multiplicity;
}
```

Listing 10 Data structures used by the parser - `Property`

The base elements the parser needs to be able to understand are classes, interfaces, and types. These elements are parsed into the *Declaration* interface. As classes and interfaces are used to create model elements, they are both parsed into *class* types. Additionally, for type aliases, the type *type* is used as seen in Listing 9. The *name* attribute of the *Declaration* is parsed into a property of type *string*. The *isAbstract* property signals whether class definitions include the abstract keyword or interfaces extend the *ABSTRACT_ELEMENT* interface. The *decorators* property collects the annotations from classes, and for interfaces, it checks, whether the interface extends the *ROOT_ELEMENT*. In type alias declarations, this property is ignored. The *extends* property collects for each class and interfaces the classes and interfaces they are extending other than the *ABSTRACT_ELEMENT* and *ROOT_ELEMENT*. The *properties* property is used to store the properties of a class, interface, or type.

Special treatment is also required to parse properties other than *name*, *isOptional*, and *multiplicity*. The property *decorators* is used to check if a property is of type *CrossReference* or if the property has the annotation *@crossReference*. The *types* property collects the types of an attribute. As it is possible that the type is defined using a union type, this property is an array. The *Type* interface consists of two attributes: *typeName*, which stores the name of the type, and *type*, which stores whether the type argument is a simple type, a complex type, or a fixed constant type.

3.2.3. R3.Validation & Transformation

In the generator, two types of transformations need to be done. The first transformation step turns the data structure into a format that can be validated more efficiently. For this, the previously defined *Declarations* are transformed into *LangiumDeclarations* using the Langium *extendedBy* property. Therefore, instead of sub-elements knowing which elements they extend, the super-classes know by which elements they are extended. The transformation consists of three steps: First, the elements are mapped to the new data structure. This includes collecting super-elements' properties and adding them to their own properties. Secondly, the *extendedBy* property for the new data structure is filled. Finally, for all abstract elements, all properties are removed as the

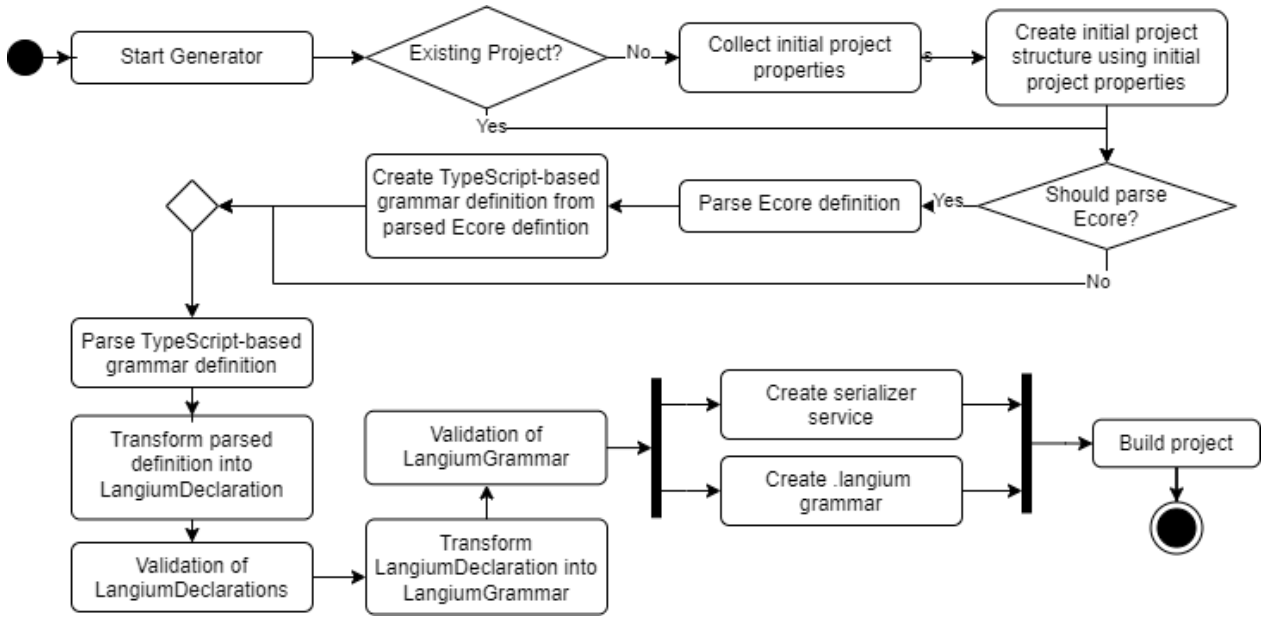

Figure 3 Workflow of the model management generator

extending element already holds these properties after the first transformation step.

After the transformation is completed, the definition can be validated. As declared in Section 2.2, the validator iterates through all *LangiumDeclarations* and validates, that only a single element includes the root-element decorator. Following a successful first validation, the *LangiumDeclarations* are transformed into a *LangiumGrammar*, whose data structure can be seen in Listing 11. This transformation follows the following three steps:

1. Search the array of *LangiumDeclarations* for the element that contains the *@root* decorator, and map it to the *EntryRule*.

2. Map all abstract *LangiumDeclarations* of types *class* and *type* to *TypeRules*. Then, search through the remaining elements, and, if a property of an element has multiple possible types, check if there already exists a *TypeRule* that represents these types. If one exists, replace the types with the name of the *TypeRule*, otherwise, create a new *TypeRule* to represent these types.

3. Map all non-abstract *LangiumDeclarations*, that are of type *class* to *ParserRules*.

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

Listing 11 Data structures used in second transformation - `LangiumGrammar`

Following this transformation, a second validation checks, whether the grammar can be serialized. In the concept of this validation, an initial serializable set (which includes string, number, and boolean types) is created and iteratively extended until no new model elements are added in an iteration. If, by then, all model elements are in the serializable set, the entire definition is serializable and the validation was successful.

3.2.4. R4. File creation

The files that need to be created can be predefined according to a certain structure, which includes wildcard phrases. These phrases are replaced with the needed value during the file creation. In Listing 12 an example template can be seen. In the creation of the file, the generator would replace the `<%=LanguageName%>` with the actual language name that has been defined in the initial project properties.

```
export class <%=LanguageName%>Example {
}
```

Listing 12 Template file including wildcard phrase

The second type of file is recreated whenever the definition file is changed. This kind of file creation can be optionally executed if Ecore is used for the metamodel definition to create the Typescript-based grammar language definition. Furthermore, it is executed to create the Langium grammar definition and the serializer service for Langium. The Langium grammar definition is created by mapping the data structure, which has been presented in Listing 11 to Langium rules as follows:

– `EntryRule`: The entry rule is the entry element rule for the Langium grammar. The entry rule has to start with the identifier *entry*, followed by the rule name. In the body of the entry rule, the structure of the model is defined. The structure has to be in a valid JSON format to enable further processing; therefore, an opening or closing curly bracket is added before the first and after the last attribute definition. Furthermore, colons are added between the different attribute definitions.

– `TypeRule`: This rule is used for unassigned rule calls, which represent rules that assign the parsing to their sub-elements. Creating a valid JSON grammar is unnecessary for this type of rule, as this is handled within the child rule elements.
– `ParserRule`: The parser rule is used to define the valid structure of a rule element. The parser rule has the same mapping procedure as the *EntryRule*, with the only difference being that the *entry* keyword is left out in the mapping.

3.2.5. R5. Package installation & build

After all files have been created, the generator executes the *npm install* command inside the newly created project. Following that, the *npm run build* command is executed, which creates the initial build. If both actions are successful, a ready-to-use Visual Studio Code extension can be started.

3.3. Model Management API

The model management API allows graphical editors to access the AST of a Langium language server. This API provides functionalities to open, close, load, save, and update a document. A central component of the model management server is the *ModelService* that is responsible for the loading, saving, and manipulation of the model state inside a *LangiumDocument*. One of the requirements of the model management API is its easy integration with Langium. Therefore, it is implemented as an npm package consisting of a module file, which defines extensions to the default Langium services, including language-specific extensions and extensions that enhance the functionalities of the language server in general. Two crucial extensions we implemented are the *JsonSerializer*, which is essential for the implementation of the JSON patch functionality in the model management server, and the *Serializer*, which is responsible for the transformation of the model AST into its textual representation. Next, we briefly describe the core functionalities of the model management API.

open With the *uri* of a file, a *language ID*, and a *client ID*, a file can be opened. The server checks, whether a document with the provided uri is already contained in its state or if it has to be loaded.

close An already opened file shall also be closed upon request by the user. While the file is closed, cross-referencing of elements in that file shall still be supported.

request This functionality can be used to request the current state of a model.

save This functionality is used to save the model's current state to the file system.

update The update functionality supports model updates from it's textual and graphical source. The update function expects the *updated model* as a parameter, either in textual form or as the complete AST of the model. If the model is sent as an AST, it is first serialized into a textual representation, followed by an update of the internal model state. To update the AST, the update method from the Langium *DocumentBuilder* is called. Finally, the current state of the model is returned.

patch The patch functionality expects as a parameter the *JSON patch* that should be applied to the current state of the document. The previous and the new state are stored to efficiently support redo/undo. A particular challenge of applying the patch is to maintain consistency and cross-references throughout the model revision. Once all preparatory steps (e.g., retrieving all affected documents and JSON objects cross-referencing a patch element) are finished, an external library called fast-json-patch is used to execute the patch and thereby updating the model into a new consistent state. Finally, the Langium references need to be reconstructed using three Langium services *AstNodeLocator*, *NameProvider*, and *LangiumDocuments*.

The *rebuildLangiumReferences* function consists of four nested functions:

– `linkNode`: This function recursively visits child elements of AST nodes to recreate the Langium references. It also makes sure that the reconstructed references are correct Langium nodes, i.e., it adds the *$container*, *$containerProperty* and *$containerIndex* properties.
– `reviveReference`: This function can be used to transforms a JSON reference into a Langium reference.
– `getRefNode`: This function can be used to search for an AST node, given a reference element, which consists of a *__documentUri* and a *__id*, or *__path*.
– `getAstNodeById`: Given the root AST node of a document and an id, this function uses the Langium utility function *streamAst* to produce a stream of AST nodes that is searched for the AST node with the given id.

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

redo and undo To redo or undo a patch, the *ModelService* calls the respective functions inside a *PatchManager* that provides a Map of previous and current model states.

4. Evaluation

To evaluate our approach, we report on two comprehensive case studies where we used our approach to generate the grammar and the model management layer for two open-source modeling tools, one for workflows[4] and BIGUML (Metin & Bork 2023a) for UML. The two examples show breadth and depth of our solution as the Workflow editor comes with a Typescript-based GLSP client and server implementation already while BIGUML comes with a Typescript-based GLSP client, a Java-based GLSP server, and a Java-based model management. Thus,

[4] https://eclipse.dev/glsp/examples/#workflowoverview

the BIGUML example is much richer in requirements and extends the workflow editor requirements from an architectural point of view (Java-based GLSP server and model management server) and in functionality (i.e., having a Property palette and an outline view). All sources and generated files of both case studies and the university example of Section 2 can be found online[5].

The challenge in both examples was to rebuild the modeling tool functionality by using *i*) our Typescript-based grammar generator and *ii*) our generated Langium-based model management server.

4.1. Realization and Integration of the Model Management Server

4.1.1. Workflow Model Management Server The GLSP-based Workflow editor already provides a client implementation and a TypeScript-based GLSP server implementation. Therefore, the existing implementation can be used to rebuild the workflow diagram example, albeit with the necessary adjustments to create a connection to the newly created model server API so that the model server can handle the model management. Additionally, the commands employed for model editing need to be adapted to utilize the JSON patch functionality of the model server API, as opposed to direct editing of the source model and subsequent transmission of the updated model to the server.

The workflow diagram language is a rather small modeling language consisting of only a few different types of nodes and one type of edge. The definition of the workflow diagram language, using the TypeScript-based grammar language (see[5]) served as the input for the Langium grammar generator and the model management server generator. For using the new model management server, the extension startup code needed to be extended by one line to also startup the generated model management server.

In order to facilitate the integration of the novel model management server, it was necessary to redirect a number of specific function calls to that server. For instance, the loading and saving of workflow models was redirected to the `SourceModelStorage`. Moreover, all model state changes needed to be related to the Langium-based AST in the `ModelState` class. This class has now been assigned responsibility for handling all model state changes through the JSON patch method. In order to support the rendering of the workflow model on the client, the `GModelFactory` class was revised to map the AST nodes to model elements. Finally, the implementation of all GLSP operators responsible for handling model updates performed by the modeler (CRUD operations on the model) needed to be adjusted. Instead of sending an updated model to the GLSP server, the new handlers create a JSON patch representing the model changes. This patch is then applied to the model state in the new model management server.

Table 1 lists the supported features after integrating the new model management server. It can be derived, that all basic modeling requirements (CRUD operations) are fully supported.

[5] Online supplementary material: https://drive.google.com/drive/folders/1W_ejDYv9H7dm63TE03QqH5y0AkPfxviR?usp=sharing

4.1.2. BIGUML Model Management Server BIGUML consists of three main components: A GLSP client, a GLSP server, and a Java-based model server. As the GLSP server implementation of BIGUML is Java-based, a Typescript implementation for the GLSP server has been added. However, as this paper focuses on the model management aspect, the steps to recreate the GLSP server will not be discussed in more detail.

The challenge was to use our Typescript-based grammar specification and generator to realize a new Langium-based model management server for BIGUML. The generator's functionality has been utilized for the initial setup of the model management, then the metamodel definition had to be created. Because BIGUML uses the UML metamodel, we created a partial Typescript-based grammar representation of the UML metamodel. Listing 13 shows the Typescript-based UML metamodel root element definition, the full specification can be found online[5].

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

Listing 13 Root element definition for the UML metamodel

The implementation of the GLSP server for the BIGUML tool is based on the already existing TypeScript-GLSP-server of the workflow diagram example with all the already discussed required adjustments to work with the newly created Langium-based model management. After the metamodel has been defined in the Typescript-based grammar language, the generator workflow has been executed (cf. Figure 3). Afterward, the rebuilt BIGUML modeling tool has been tested against the standard modeling tool features like CRUD operations on model elements (cf. Table 1). The focus then moved to further support the advanced features like a property palette and a model outline view offered by BIGUML.

The property palette in BIGUML offers a form-based view that supports direct editing of model element properties like attributes and operations of a UML class. To be able to handle actions of the property palette, handlers for the `RequestPropertyPaletteAction` and `UpdateElementPropertyPaletteAction` had to be implemented.

The `RequestPropertyPaletteAction` handler must prepare the property palette's form according to the selected model element. To prepare this form, a utility class named `PropertyPaletteBuilder` has been created, which eases the creation of property palette items using the builder pattern. The implementation of the `UpdateElementPropertyPaletteAction` handler creates a `UpdateOperation`, which updates the value of a selected property inside the property palette. After the registration of these new handlers, the property palette was fully functional based on the new model management server.

Figure 4 shows an example of the property palette for a Class in the class diagram. As can be seen, all properties of the class are listed in the property palette, and input fields, checkboxes, and choice elements exist to change the values of these proper-

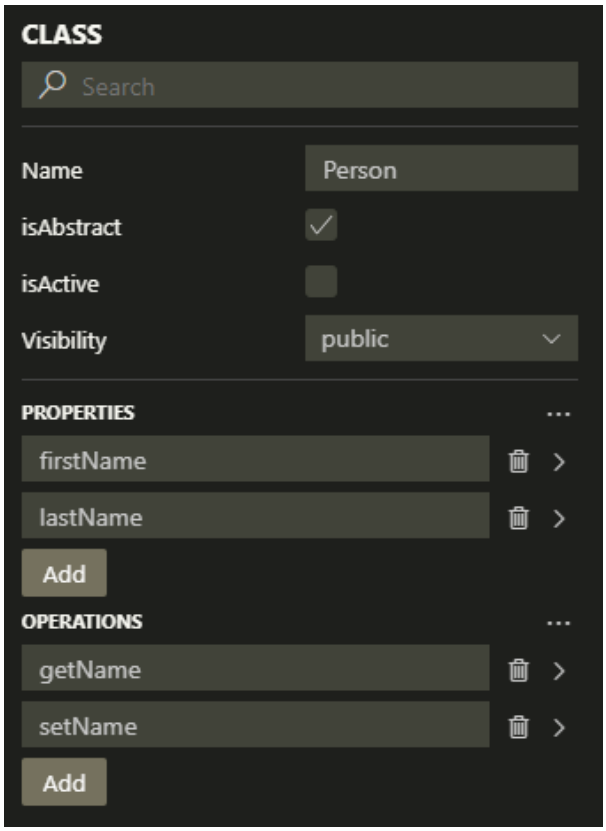

Figure 4 Property palette in the rebuilt BIGUML tool.

ties. Furthermore, references exist for the child elements that can be used to open the property palette for the selected child element as can be seen next to the properties and operations. We can thus state, that our new model management server fully supports the property palette in BIGUML (cf.Table 1).

The only requirement that is currently not yet fully supported is the outline tree view, which has only been partially implemented. However, this functionality can also be realized by creating an action handler for the *RequestOutlineTreeView* action, which is responsible for loading and transforming the model data into the tree view data. The handler would need to make an API call to request the current ModelState and then transform the retrieved AST into the required structure by the tree view.

4.2. Discussion

Now, we focus on the core contributions of this research, which are aligned to three research questions: **RQ-1:** how should a type definition language be conceptualized to accurately and comprehensively define metamodels; **RQ-2:** how can the previously defined type definitions be used to automatically generate a valid Langium language specification; and **RQ-3:** how can the AST created by Langium be made available to model-oriented clients to enable model editing directly in the browser?

We presented a solution to create a browser-based model management solution using a Typescript-only technology stack (**RQ-1**). For this, a Typescript-based grammar language was conceptualized. This step included analyzing Langium's capabilities and the widely used Ecore metamodel. After gathering all requirements, the concept of the Typescript-based grammar language was developed. This encompasses the creation of simple model elements as well as the addition of cross-references between them. While this first definition already includes a lot of functionalities like multiplicities, containment- and cross-references, some specific aspects of the very large Ecore language are left for future extension (like *derived*, *transient*, and *ordered* attributes). The online supplementary material features a detailed analysis of the supported features[5].

In response to **RQ-2**, we developed a concept and an implementation for a generator capable of generating a standard JSON grammar and the entire setup for a Langium-based model management server. In the evaluation, we showed that the generator is able to create the model management for two state-of-the-art modeling tools, one for Workflows, the other for the UML.

Finally, regarding **RQ-3**, to enable model-oriented clients to access Langium's AST, the requirements for a model server API were analyzed and a model server API was created, which provides the implementation to fulfill these requirements. This model server API has been integrated within Langium. In the evaluation, we showed that the functionality of BIGUML and the Workflow editor could be rebuilt using the Langium-based model management solution.

4.3. Threats to Validity

This research is not exempted from threats to validity. Primarily, we want to stress that our approach currently has been tested on two instances, the Workflow editor and BIGUML. While this is limited concerning the number of instances, both modeling tools come with a heterogeneous technology stack and functionality, and rich metamodels. Consequently, this initial evaluation shows the feasibility of our approach. Future applications will need to further test the expressivity and generality of our solution.

5. Implications for Research and Practice

To show the far-reaching implications of our proposal, we aimed to realize some advanced web-modeling capabilities on top of it. Blended modeling has been a popular advanced topic in the field of model engineering (David et al. 2022). So far, blended modeling tools have primarily been developed based on traditional frameworks e.g., Xtext (Eclipse Foundation 2024; Glaser & Bork 2021) or EMF (Eclipse Foundation 2024a). Novel approaches (Petzold 2022; Giraudet 2022) using web-based technologies either do not provide enhanced editing possibilities on the graphical model, only generate a read-only visual representation of the model, or do not consider non-semantic information in the textual representation of the model e.g. comments or formatting, that would get lost after modifying the model in the graphical representation.

In the following, we show how our generated model management server, combined with GLSP and Langium enables the generation of web-based modeling tools for hybrid textual and graphical modeling in BIGUML. Consequently, we need to extend our generated model service to not only enable model-oriented clients but also textual model clients to access and

Table 1 Evaluation of the requirements of the rebuilt modeling tools. Legend:
✓ ⇒ supported, (✓) ⇒ partially supported; n/a ⇒ not supported by the original workflow editor.

Requirement	Workflow editor	BIGUML
Creating, Editing, and Deleting nodes	✓	✓
Moving and Resizing nodes	✓	✓
Creating, Editing, and Deleting edges	✓	✓
Showing model elements in property palette	n/a	✓
Creating child nodes via the property palette	n/a	✓
Editing nodes via the property palette	n/a	✓
Deleting child nodes via the property palette	n/a	✓
Showing the model in the outline view	n/a	(✓)

make changes to the AST. The challenges here were threefold: *i*) how can the model service API allow textual and graphical editors to manage and manipulate the underlying AST jointly; *ii*) how to implement the modification model to allow for simultaneous modifications on the textual and graphical models; and *iii*) how to handle non-semantic information?

The developed approach had to provide a solution to pursue updates on the model in two directions: a) updates made in the text editor where the language server has to delegate the action handling to GLSP; and b) updates made in the graphical representation where GLSP has to delegate the action handling to the text editor and the language server. As the model service already provides the up-to-date AST of the model to requesting clients, the direction a) can be implemented via listening to changes on the model and updating GLSP's source model with the new version of the model. After GLSP receives an update, it re-generates its graphical model and delegates the updated graphical model to the client to update the graphical representation of the model.

To implement updates in the direction b) two approaches were developed. Firstly, updates in the graphical representation of the model were directly applied to the model's AST by the GLSP server, and the new version of the AST was sent to the model server to serialize it and update the content of the text editor. However, this method raises the problem of removing the non-semantic information of the model's textual representation, as the serializer would not consider comments and white spaces in the model, as the AST does not contain this information. To mitigate this problem, the GLSP server has to implement methods for updating, deleting, and inserting nodes in the model's textual syntax. This can be achieved as the AST generated by Langium provides the positions of the nodes in the textual syntax, and therefore, nodes can be directly edited, deleted, or inserted without losing the non-semantic information of the model's textual syntax. GLSP then sends the updated model as the textual syntax to the model service, which updates the text editor's content and rebuilds the *LangiumDocument* containing the model's updated AST to keep the model server up-to-date.

The developed concept was implemented both on the Workflow editor and on the generated BIGUML editor as a VS Code Extension (Microsoft 2024). The implemented extension successfully provides simultaneous editing in the graphical and textual editors for package and class diagrams as visualized in a screenshot in Figure 5 and a video online[5].

6. Conclusion

In order to bring modeling tools into the cloud, it is crucial to build these tools with a homogeneous technology stack consisting of web technologies that natively allow cloud-based deployment and use in web browsers. In this paper, we proposed a novel Typescript-only approach for generating Langium-based modeling language grammars and subsequently generating entire model management servers. Through two use cases, we evaluated our approach and showed, how developers can build next-generation web-based modeling tools. All software components described in this paper will be published open-source to enable wide industrial and academic adoption and use.

References

Bainczyk, A., Busch, D., Krumrey, M., Mitwalli, D. S., Schürmann, J., Dongmo, J. T., & Steffen, B. (2022). Cinco cloud: A holistic approach for web-based language-driven engineering. In T. Margaria & B. Steffen (Eds.), *Leveraging applications of formal methods, verification and validation. software engineering - 11th international symposium, isola 2022, rhodes, greece, october 22-30, 2022, proceedings, part II* (Vol. 13702, pp. 407–425). Springer. doi: 10.1007/978-3-031-19756-7_23

Bork, D., Karagiannis, D., & Pittl, B. (2020). A survey of modeling language specification techniques. *Inf. Syst.*, *87*. doi: 10.1016/J.IS.2019.101425

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

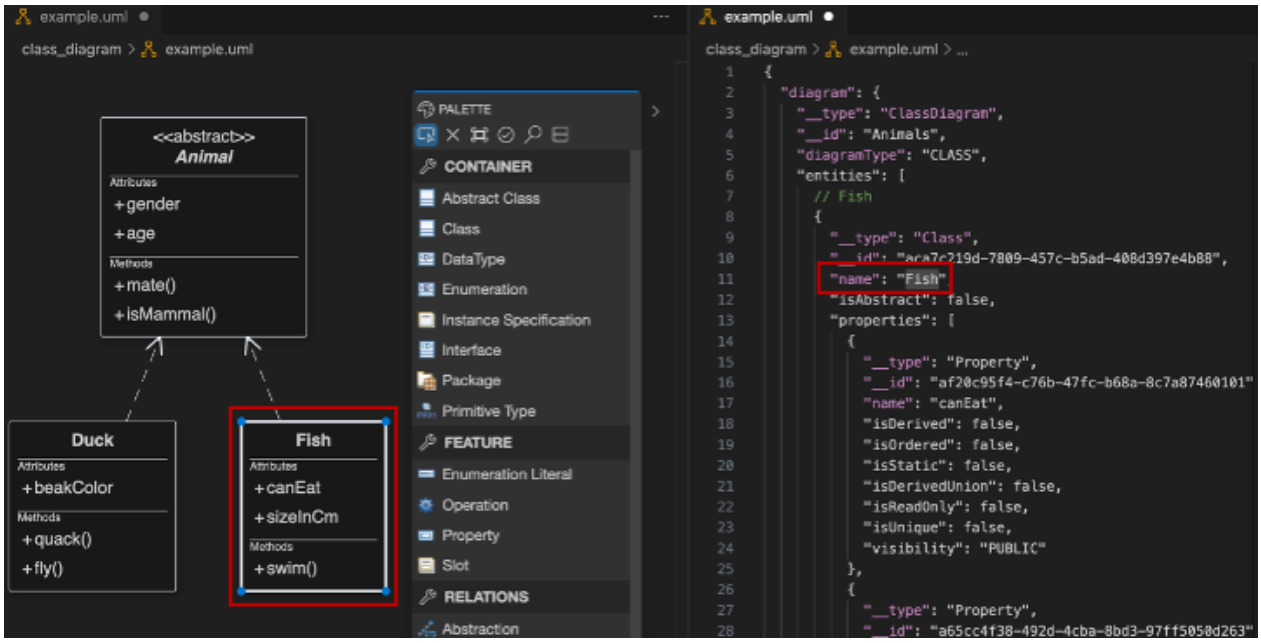

Figure 5 Blended modeling with BIGUML: the synchronized selection is highlighted.

Bork, D., Langer, P., & Ortmayr, T. (2023). A vision for flexible glsp-based web modeling tools. In J. P. A. Almeida, M. Kaczmarek-Heß, A. Koschmider, & H. A. Proper (Eds.), *The practice of enterprise modeling - 16th IFIP working conference, poem 2023, vienna, austria, november 28 - december 1, 2023, proceedings* (Vol. 497, pp. 109–124). Springer. doi: 10.1007/978-3-031-48583-1_7

client.IO. (2024). *Jointjs.* https://www.jointjs.com/. (Accessed: 24.05.2024)

David, I., Latifaj, M., Pietron, J., Zhang, W., Ciccozzi, F., Malavolta, I., ... Hebig, R. (2022, 06). Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study. *Software and Systems Modeling*, *22*. doi: 10.1007/s10270-022-01010-3

Eclipse Foundation. (2024). *Eclipse GLSP.* https://eclipse.dev/glsp/. (Accessed: 01.02.2024)

Eclipse Foundation. (2024). *Eclipse Graphical Language Server Platform - Servers & Integrations.* https://eclipse.dev/glsp/documentation/integrations/. (Accessed: 24.03.2024)

Eclipse Foundation. (2024a). *EMF.* https://www.eclipse.org/modeling/emf/. (Accessed: 24.06.2023)

Eclipse Foundation. (2024b). *EMF.cloud.* https://www.eclipse.org/emfcloud/. (Accessed: 22.06.2023)

Eclipse Foundation. (2024c). *Sirius web.* https://eclipse.dev/sirius/sirius-web.html. (Accessed: 24.05.2024)

Eclipse Foundation. (2024). *Sprotty.* https://projects.eclipse.org/projects/ecd.sprotty. (Accessed: 25.03.2024)

Eclipse Foundation. (2024). *Xtext.* https://www.eclipse.org/Xtext/. (Accessed: 24.06.2023)

EclipseSource. (2024). *Glsp.* https://eclipse.dev/glsp/. (Accessed: 14.02.2024)

Giner-Miguelez, J., Gómez, A., & Cabot, J. (2022). Describeml: A tool for describing machine learning datasets. In *Proceedings of the 25th international conference on model driven engineering languages and systems: Companion proceedings* (p. 22–26). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3550356.3559087 doi: 10.1145/3550356.3559087

Giraudet, T. (2022). *Langium + sirius web = heart.* https://blog.obeosoft.com/langium-sirius-web. (Accessed: 08.02.2024)

Glaser, P., & Bork, D. (2021). The biger tool - hybrid textual and graphical modeling of entity relationships in VS code. In *25th international enterprise distributed object computing workshop, EDOC workshop 2021, gold coast, australia, october 25-29, 2021* (pp. 337–340). IEEE. doi: 10.1109/EDOCW52865.2021.00066

Jarke, M., Gallersdörfer, R., Jeusfeld, M. A., & Staudt, M. (1995). Conceptbase - A deductive object base for meta data management. *J. Intell. Inf. Syst.*, *4*(2), 167–192. doi: 10.1007/BF00961873

Jézéquel, J.-M., Barais, O., & Fleurey, F. (2011). Model driven language engineering with kermeta. In J. M. Fernandes, R. Lämmel, J. Visser, & J. Saraiva (Eds.), *Generative and transformational techniques in software engineering iii: International summer school, gttse 2009, braga, portugal, july 6-11, 2009. revised papers* (pp. 201–221). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from https://doi.org/10.1007/978-3-642-18023-1_5 doi: 10.1007/978-3-642-18023-1_5

Kelly, S., Lyytinen, K., & Rossi, M. (1996). Metaedit+: A fully configurable multi-user and multi-tool CASE and CAME environment. In P. Constantopoulos, J. Mylopoulos, & Y. Vassiliou (Eds.), *Advances information system engineering, 8th international conference, caise'96, heraklion, crete, greece, may 20-24, 1996, proceedings* (Vol. 1080, pp. 1–21). Springer. doi: 10.1007/3-540-61292-0_1

Lafontant, L., & Syriani, E. (2020). Gentleman: a lightweight web-based projectional editor generator. In E. Guerra

& L. Iovino (Eds.), *MODELS '20: ACM/IEEE 23rd international conference on model driven engineering languages and systems, virtual event, canada, 18-23 october, 2020, companion proceedings* (pp. 1:1–1:5). ACM. doi: 10.1145/3417990.3421998

Maróti, M., Kecskés, T., Kereskényi, R., Broll, B., Völgyesi, P., Jurácz, L., … Lédeczi, Á. (2014). Next generation (meta)modeling: Web- and cloud-based collaborative tool infrastructure. In D. Balasubramanian, C. Jacquet, P. V. Gorp, S. Kokaly, & T. Mészáros (Eds.), *Proceedings of the 8th workshop on multi-paradigm modeling co-located with the 17th international conference on model driven engineering languages and systems, mpm@models 2014, valencia, spain, september 30, 2014* (Vol. 1237, pp. 41–60). CEUR-WS.org. Retrieved from https://ceur-ws.org/Vol-1237/paper5.pdf

Metin, H., & Bork, D. (2023a). 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* (pp. 40–44). IEEE. Retrieved from https://doi.org/10.1109/MODELS-C59198.2023.00016 doi: 10.1109/MODELS-C59198.2023.00016

Metin, H., & Bork, D. (2023b). On developing and operating glsp-based web modeling tools: Lessons learned from bigUML. In *Proceedings of the 26th international conference on model driven engineering languages and systems, MODELS 2023*. IEEE. Retrieved from https://model-engineering.info/publications/papers/MODELS23-GLSP-Development-Web.pdf

Michael, J., Bork, D., Wimmer, M., & Mayr, H. C. (2023). Quo vadis modeling? findings of a community survey, an ad-hoc bibliometric analysis, and expert interviews on data, process, and software modeling. *Software and Systems Modeling*. (in press) doi: 10.1007/s10270-023-01128-y

Microsoft. (2024). *Extension api | visual studio code.* https://code.visualstudio.com/api. (Accessed: 22.02.2024)

Microsoft. (2024). *Language Server Protocol.* https://microsoft.github.io/language-server-protocol/. (Accessed: 10.02.2024)

Paige, R. F., & Cabot, J. (2024). What makes a good modeling research contribution? *Software and Systems Modeling*, 1–5. doi: 10.1007/s10270-024-01177-x

Petzold, J. (2022). *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. (Accessed: 08.02.2024)

Rocco, J. D., Ruscio, D. D., Salle, A. D., Vincenzo, D. D., Pierantonio, A., & Tinella, G. (2023). jjodel - A reflective cloud-based modeling framework. In *ACM/IEEE international conference on model driven engineering languages and systems, MODELS 2023 companion, västerås, sweden, october 1-6, 2023* (pp. 55–59). IEEE. doi: 10.1109/MODELS-C59198.2023.00019

Rodríguez-Echeverría, R., Izquierdo, J. L. C., Wimmer, M., & Cabot, J. (2018a). An LSP infrastructure to build EMF language servers for web-deployable model editors. In R. Hebig & T. Berger (Eds.), *Proceedings of MODELS 2018 workshops* (Vol. 2245, pp. 326–335). CEUR-WS.org.

Rodríguez-Echeverría, R., Izquierdo, J. L. C., Wimmer, M., & Cabot, J. (2018b). Towards a language server protocol infrastructure for graphical modeling. In A. Wasowski, R. F. Paige, & Ø. Haugen (Eds.), *Proceedings of the 21th ACM/IEEE international conference on model driven engineering languages and systems, MODELS 2018, copenhagen, denmark, october 14-19, 2018* (pp. 370–380). ACM. doi: 10.1145/3239372.3239383

Steinberg, D. (2009). *Emf : Eclipse modeling framework* (2nd ed. ed.). Boston: Addison Wesley.

Storm. (2024). *React diagrams.* https://github.com/projectstorm/react-diagrams. (Accessed: 24.05.2024)

Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Mierlo, S. V., & Ergin, H. (2013). Atompm: A web-based modeling environment. In Y. Liu et al. (Eds.), *Joint proceedings of models'13 invited talks, demonstration session, poster session, and ACM student research competition co-located with the 16th international conference on model driven engineering languages and systems (MODELS 2013), miami, usa, september 29 - october 4, 2013* (Vol. 1115, pp. 21–25). CEUR-WS.org. Retrieved from https://ceur-ws.org/Vol-1115/demo4.pdf

TypeFox GmbH. (2024). *Langium.* https://langium.org. (Accessed: 01.02.2023)

xyflow. (2024). *React flow.* https://reactflow.dev/. (Accessed: 24.05.2024)

About the authors

David Jaeger David Jäger is a Fullstack Software Engineer at Inercomp, where he helps develop a web-based platform to manage energy portfolios. You can contact the author at david.jaeger10@gmail.com.

Adam Lencses Adam Lencses is a Fullstack Software Engineer at Certible, where he is involved in the design, development, and maintenance of online and live personal certification solutions. You can contact the author at adam@lencses.com.

Martin Fleck Martin Fleck is Lead Software Architect at EclipseSource, where he drives the development of web-based IDEs, modeling tools, and custom engineering environments. He is a core contributor to projects like Eclipse GLSP, EMF Cloud, and CDT Cloud, and actively maintains and contributes to open source technologies including Eclipse Theia. You can contact the author at mfleck@eclipsesource.com.

Philip Langer Dr. Philip Langer is CEO of EclipseSource, supporting companies in architecting and developing custom IDEs, domain-specific tools, and modeling environments with his extensive experience on Eclipse tool platforms and modeling technologies. His current focus is on frameworks for web-based IDEs, graphical modeling tools, custom C/C++ IDEs, and AI integrations in engineering tools. Philip is the project lead of several successful open source technologies, such as Eclipse GLSP, EMF Cloud, and CDT Cloud. Moreover, he is an active

committer on Theia, Sprotty, and more. You can contact the author at planger@eclipsesource.com.

Dominik Bork Dominik Bork is Associate Professor for Business Systems Engineering at TU Wien. His research interests comprise conceptual modeling and model engineering as well as their application in domains such as modeling tool development and web modeling. A primary focus of ongoing research is on the mutual benefits of conceptual modeling and artificial intelligence. You can contact the author at dominik.bork@tuwien.ac.at or visit https://model-engineering.info/.