Marburg University

Software engineering group

June 29, 2025

Creating web-based diagram editors for specifying and executing model transformations

Master thesis

from

Florian Weidner

Abstract

lipsum bla bla bla

CONTENTS

I. Introduction

A. Background and Motivation

In software engineering, often Model-Driven Engineering (MDE) is used to increase development productivity and quality. [**transformations-modeldriven**] Concepts are modeled closer to the domain, so that they describe important aspects of a solution with human-friendly abstractions. The models can also be used to generate application fragments, that can be directly used as a template source code. In the process of MDE, many activities need to transform source models into different target models, while following a set of transformation rules. This model transformation process is based on algebraic graph transformations. A metamodel is used to model the structure and rules of the concept. The resulting transformation language can provide automatic model creation, development, and maintenance activities. [**transformations-modeldriven**] One framework to use MDE is EMF by the Eclipse Foundation. It provides a basis for application development, using modeling and code generation facilities. Many frameworks build upon EMF, providing various MDE tools like code generators, graphical diagramming, model transformation, or model validation. [**emf**] One model transformation framework is Henshin. [**henshin-repo**] It tries to provide model transformation capabilities with a high level of usability. [**henshin-usability**] For metamodels it uses EMF Ecore files and for instance models EMF XMI files. The framework enables transformations on XMI instance files with a defined transformation language. It provides a graphical and textual syntax to create these transformation rules. [**henshin-repo**] Henshin can be used as a Eclipse plugin. Eclipse makes it easy to access, but especially for new users, the heavy editor makes the use of Henshin unintuitive. Therefore, the goal exists to create a graphical option to use the Henshin model transformations without the overhead of the heavy Eclipse editor. A web-based graphical editor would make the use of Henshin even more accessible and intuitive.

GLSP is a open-source framework by the Eclipse Foundation, which can be used to build a web-based Henshin graph editor. The framework is used to develop custom diagram editors for distributed web-applications. [**glsp-repo**] It can provide graph editors for the Eclipse Desktop IDE, Eclipse Theia, Visual Studio Code (VS Code) and a stanalone version usable in any website. It brings the support of EMF models as a data source and the Henshin SDK can be used from the Java server of GLSP. [**glsp-doc**] With these functionalities, GLSP fits to create an easy accessible, intuitive application to create and apply Henshin model transformations, called Henshin Web.

The goal of this scientific work is to provide relevant information about the used technologies. Also existing web-based model transformation tools will be compared in the related work section. In section VI, the deployment and usage of the Henshin Web editor will be discussed. The goal is to provide a web-based editor that can be used without any dependencies, like an installed IDE or other tools. The editor should be easy to access and use, so that it can be used by new users without any prior knowledge of model transformations or Henshin.

II. Background

In this section, the theoretical background of the project and used technologies are described. First the Eclipse Foundation is introduced, as many used frameworks are developed under the Eclipse Foundation. Then, the Eclipse Modeling Framework is described, as it is the core of the used frameworks. After that, the model transformation language Henshin is introduced. Finally, the framework GLSP is described, that is used to create web-based diagram editors.

A. Eclipse Foundation

The Eclipse Foundation is a not-for-profit, member-supported corporation that provides an environment for individuals and organizations for collaborative and innovative software development. [**eclipse-review**] The Eclipse Foundation grew out of the publication of the Eclipse Integrated Development Environment (IDE) code from IBM in 2001. The Eclipse Foundation itself was founded in 2004. The new organization was founded to continue the development of Eclipse IDE as an open source platform. Over time, the organization initiated numerous projects in the Eclipse environment, all operating under the Eclipse Public License. [**heise-eclipse-foundation**, **eclipse-review**] In the recent years, the key initiatives of the Eclipse Foundation are contributing to european digital sovereignty, enhancing security measures, innovating Software-Defined Vehicle (SDV), organizing community events, and improving their most popular projects. Popular projects are for example the Jakarta EE, an ecosystem for cloud-native applications with java, Eclipse Temurin, providing open source Java Development Kits and the Eclipse IDE. [**eclipse-report**] In total, the Eclipse Foundation hosts more than 400 open source projects, supported 14 european research projects in 2024, and has 117 organizations participating in commits. [**eclipse-report**]

The scope of this work remains within the Eclipse Foundation ecosystem. All frameworks used are projects from the Eclipse Foundation. The used frameworks are described in the sections II-B, II-C and II-D.

The Eclipse IDE is not the main project, but it is still an important part of the Eclipse infrastructure. It is divided into four main components: Equinox, the Platform, the Java Development Tools (JDT) and the Plug-in Development Environment (PDE). Together they provide everything to develop and extend Eclipse-based tools. Equinox and the Platform are the core of the Eclipse IDE. With expanding the core with the JDT or other plugins, the IDE can be used to develop different programming languages, like Java, C/C++, or PHP. [**emf**] Eclipse provides different packages to download, depending on the use case. One package is the Eclipse Modeling Tools package by the Eclipse Modeling Project. It provides tools and runtimes to build model-based applications. It can be used to graphically design domain models and test those models by creating and editing dynamic instances. Also, Java code can be generated from the models to get a scaffold that can be used to create applications on top. [**eclipse˙modeling**] The base of the Eclipse Modeling Tool is EMF (section II-B). Other modeling tools and projects that are built on top of the EMF core functionality provide capabilities for model transformation, database integration, or graphical editor generation. [**emf**]

B. Eclipse Modeling Framework (EMF)

> „Eclipse Modeling Framework (EMF) is a modeling framework and code generation facility for building tools and other applications based on a structured data model.“ [**emf-repo**]

Eclipse Modeling Framework (EMF) is the core part of the Eclipse Modeling Project and unifies the representation of models in UML, XML and Java. You can define your model in one of these formats and use EMF to generate the other formats.

EMF consists of three components. The EMF core part provides Ecore metamodels, runtime support for the models, and a basic API for manipulating EMF objects generically. Ecore metamodels are used to describe the structure of a model. [**eclipse˙emf**] They can be serialized in XML Metadata Interchange (XMI) 2.0, as Ecore XMI, and have the file extension *.ecore*. There are several Ecore classes to represent a model, here are the most important ones:

- **EClass**: A class in the model that is identified by a name, containing attributes and references to other classes. It can also refer to a number of other classes as its supertypes to support inheritance. [**emf**]
- **EAttribute**: An attribute of a class, that are identified by a name and have a type. [**emf**]
- **EDataType**: A simple data type like `EString`, `EBoolean` or `EJavaClass`. [**emf**]
- **EReference**: A reference to another class, containing a link to an instance of that class. [**emf**]

Together, **emf** called these classes the Ecore kernel. In Figure 1 you can see the kernel classes and their relations. These classes are enough to define simple models. **EAttribute** and **EReference** have a lot of similarities. They both define the state of an instance of an **EClass** and have a name and a type. For that, Ecore provides a common interface for both, called **EStructuralFeature**. Ecore can also model behavioral features of classes as **EOperation** using **EParameter**. All classes have the common interface **EObject**, being the root of all modeled objects. Related classes are grouped into packages called **EPackage**. It is represented by the root element when the model is serialized. [**emf**]

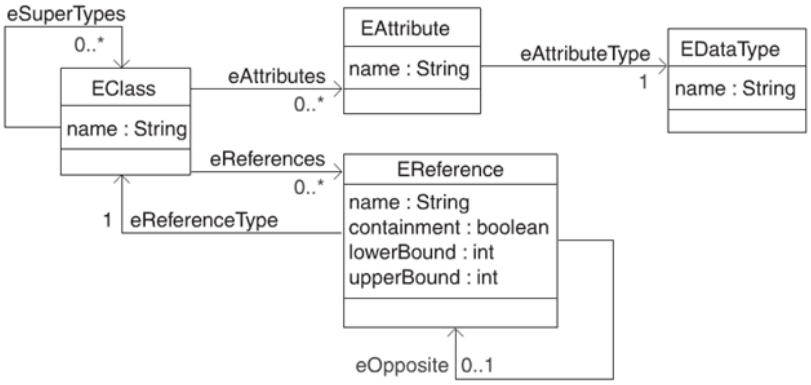

Figure 1. The Ecore kernel. Image obtained from [**emf**]

The second component of EMF is EMF.Edit. It provides generic reusable classes to build viewers and editors for EMF models. With these classes, EMF metamodels can be displayed in JFace viewers, that are part of the Eclipse UI. [**eclipse˙emf**] The Eclipse IDE can display an Ecore model in a tree viewer. Eclipse accesses the data over the `ITreeContentProvider` interface to navigate the content and the `ILabelProvider` interface to provide the label text and icons for the displayed objects. The properties of objects are displayed

in a Property Sheet over the `IPropertySourceProvider`, where the user can edit the model. EMF.Edit also provides undo and redo operations when creating or editing an instance model. For that, it uses a command framework with commands like an `AddCommand`, `SetCommand` or `CopyCommand`. [**emf**]

The third component is EMF.Codegen. It can generate Java code for a complete editor for EMF instance models of an Ecore metamodel. It provides different generation options. So, unlike EMF.Edit, that just provides generic classes for Ecore models, EMF.Codegen directly generates complete editors with a UI. [**eclipse˙emf**] The generation can be done over a wizard in the Eclipse IDE or by using the command line interface. [**emf**] The generation can be separated into three levels. The first level is to generate Java interfaces and implementations for the Ecore model classes and a factory- and package-implementation class. The second level generates specific `ItemProviders` to edit instance models based on the metamodel. The classes are structured like the EMF.Edit component for the Ecore models. The third level generates a structured editor with UI that works like the Ecore editor in the Eclipse IDE and can be a starting point for customization. [**eclipse˙emf**] There are many frameworks that build on top of EMF, using these generation capabilities to create further modeling functionality. For model transformations the most popular frameworks that build upon EMF are Eclipse Acceleo, Eclipse VIATRA, Eclipse ATL, Eclipse QVT Operational, Eclipse QVT Declarative and Henshin II-C.

C. Henshin

One part of the Eclipse Modeling Project for model transformations is Henshin. It can be used as a plugin in the Eclipse IDE or as an SDK. It provides a graphical and textual syntax to define model transformation rules and apply them to EMF XMI instance models. It can be used for endogenous transformations, where EMF model instances are directly transformed, and exogenous transformations, where new instances are generated from given instances using a trace model. It also brings efficient in-place execution of transformations using an interpreter with debugging support and a performance profiler. Henshin also provides conflict and dependency analysis, and state space analysis for verification. [**henshin-repo**]

Henshin builds on top of EMF. It uses an Ecore metamodel to define the structure of the transformation rules, resulting in a serialized XMI file with the file extension *.hensin*, that can therefore be edited in the Eclipse tree editor. [**henshin-repo**] The metamodel of the transformation rules uses another Ecore metamodel that models the model structure of the domain, to type the nodes, egdes, and attributes of the rules. [**henshin**] In Figure 2 you can see the Henshin transformation rule metamodel. A rule consists of a Right-Hand Side (RHS) graph, a Left-Hand Side (LHS) graph and attribute conditions. Additionally, mappings between the LHS and RHS graph are defined between nodes. The mapping of the edges is done implicitly by the mapping of the source and target nodes. [**henshin**] Henshin uses units to control the order of rule applications. With units, control structures can be defined. Also, parameters can be passed from the previous executed rule to the next one to have a controlled object flow. Henshin's tansformation language is based on algebraic graph transformations, complying with the syntactical and semantic structure of rules and transformation units. This ensures a language usable for formal verification or validation. [**henshin**]

In the Eclipse IDE, rules can also be edited in a graphical editor. The rules are displayed as a single graph, calculated from the LHS and RHS graphs. The nodes and edges are annotated with *<<preserve>>*, *<<create>>*, *<<delete>>*, *<<forbid>>* or *<<require>>* to indicate what happens to the nodes and edges when applying the rule. These annotations can be directly edited in the graphical editor and the the LHS and RHS graphs are then adapted to the change. Also, multiple Negative Application Conditions (NACs),

Positive Application Conditions (PACs) and parameters can be specified directly. [**henshin-repo**] When a set of transformation rules are specified, they can be applied to an EMF XMI instance model, by using a wizard in the Eclipse IDE. There, the source model, the rule, and its parameters can be selected. The result of the transformation can be seen in a new XMI instance file. [**henshin-repo**] Next to the graphical editor, Henshin also provides a textual syntax to define transformation rules and units. In a *.henshin_rule* file with the keyword **rule** a name and parameters, a new rule can be described. If you want to define a node, you can use the keyword **node** with a action keyword like **create** or **preserve** to specify the action of the node in the transformation.

The Henshin SDK consists of multiple packages oriented to the package structure of EMF. Next to a model, edit, and editor package, it provides an interpreter package, that contains a default engine to execute model transformations.

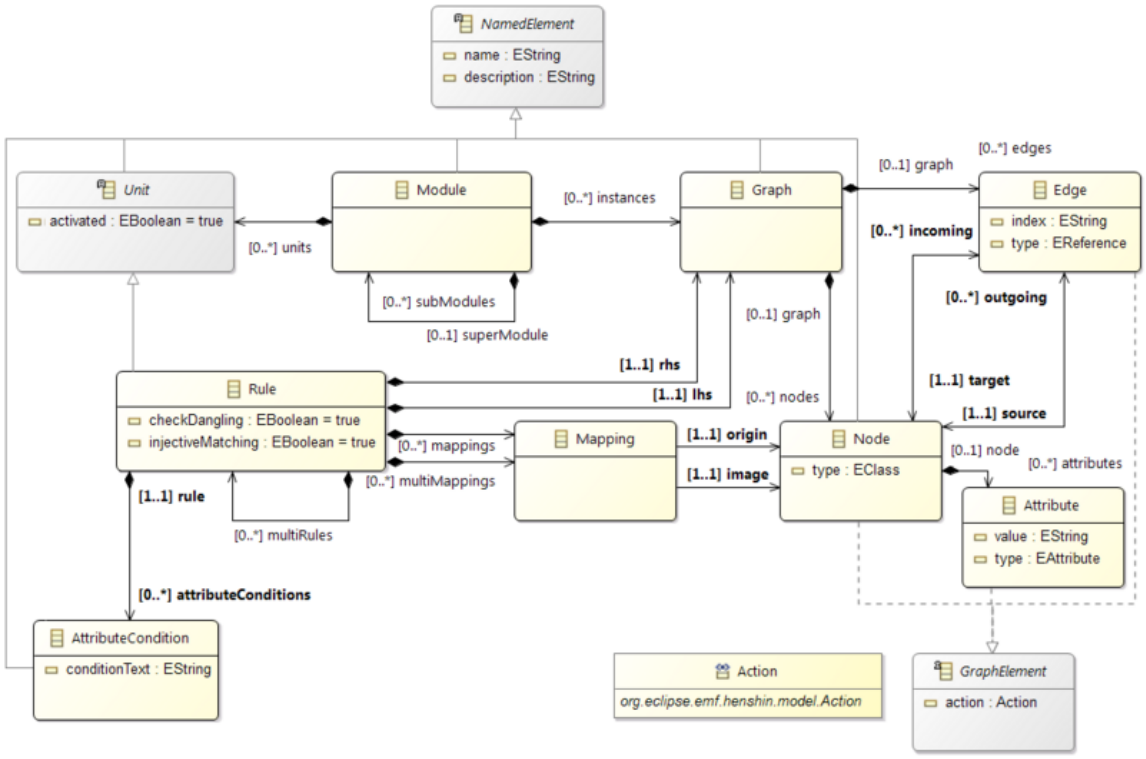

Figure 2. Henshin transformation rule metamodel. Image obtained from [**henshin-repo**]

D. Graphical Language Server Platform (GLSP)

GLSP is a framework that provides components for the development of GUIs for web-based diagram editors. [**glsp-repo**] It is organized within the Eclipse Cloud Development project. [**glsp-doc**] With the framework, custom diagram editors for Eclipse Theia, Eclipse IDE, Visual Studio Code, or standalone web apps can be created. It uses a client-server architecture, where the client is implemented with TypeScript and for the server, GLSP provides implementations in Java and TypeScript based on nodejs, even though the server could be implemented in any programming language. As the server for this project is implemented in Java,

the following discussion focuses exclusively on the Java implementation of the GLSP server. Client and server communicate over JSON-RPC with an action protocol that is similar to the Language Server Protocol [**lsp-repo**].

The GLSP server is responsible for loading a source model and defines how to transform it into the graphical model, that should be displayed. The source model can be of any format, e.g., a database, JSON file, or an EMF model. GLSP provides dedicated modules for loading EMF models. The Java server uses Google Guice [**guice-repo**] for Dependency Injection (DI). The GLSP server distinguishes between DI containers. There is one server DI container to configure global components that are not related to specific sessions. For every client session, there is a diagram session DI container, that holds session specific information, handlers, and states associated with a single diagram language. In Figure 3 you can see that the diagram session DI container run inside the server DI container. GLSP provides some abstract base classes that have to be implemented to create a working diagram server language, that can provide a diagram to display at the client. All concrete implementations of one diagram language have to be registered in a `DiagramModule`. The server can handle multiple diagram languages by providing different diagram modules. There are some classes that have to be implemented. The interface `SourceModelStorage` defines how to load and save the source model. There is already a default abstract implementation for EMF models, that loads the XMI file into a `ResourceSet`. The interface `GModelFactory` is used to map the source model to the GLSP internal graphical model structure. Here also an abstract `EMFGModelFactory` is provided. Another important part is the `GModelState` interface, that defines the state of a client session and holds all information about the current state of the original source model. All services and handlers use the `GModelState` to obtain required information for their tasks.

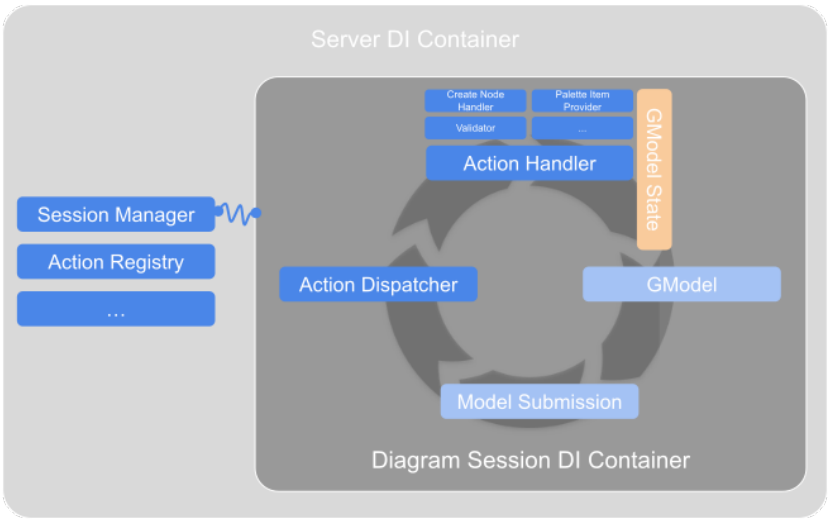

Figure 3. Server DI Container vs Diagram Session DI Container. Image obtained from [**glsp-doc**]

When the diagram should be displayed in the editor, the client sends a `RequestModelAction` with a URI of the source model to the server. The server invokes the `SourceModelStorage` to load the source model and then uses the `GModelFactory` to translate it into the graphical model, which is then sent to the client to render it. For an edit operation, the client sends the operation request to the server, where the corresponding handler is invoked. The handler modifies the source model directly. After that, the server

invokes the `GModelFactory` again to map the newly modified source model into a new graphical model, which is sent to the client to re-render. The two use cases share many steps. Since a new graphical model is created every time, the format of the source model is independent and can be of any format. [**glsp-doc**]

The GLSP client is responsible for rendering the graph and managing user interactions. The client requests all possible editing operations that can be performed on the specific model. As the client for this project is integrated into Eclipse Theia, the following discussion focuses exclusively on the Theia integration of the GLSP client. [**glsp-doc**] GLSP provides four main UI components to apply commands or edit the graph but also allows custom User Interface (UI) extensions:

- **ToolPalette**: The ToolPalette is an expandable UI element located on the top left of the diagram editor. By default, it provides basic options to switch between selection, deletion, and marquee tools, validate the model, reset the viewport, and search in the listed operations below. Below it lists all nodes and edges that can be created in the diagram by default. It can be extended with custom actions by implementing and registering the `ToolPaletteItemProvider` interface at the server. [**glsp-doc**, **glsp-repo**]
- **CommandPalette**: The CommandPalette can be invoked by pressing *Ctrl+Space*. It provides a search field to search for commands or actions that were registered. Commands can be registered by implementing and registering the `CommandPaletteActionProvider` to the server or implementing and registering the `CommandContribution` interface to the Theia frontend module. [**glsp-doc**, **glsp-repo**]
- **ContextMenu**: The ContextMenu is a popup menu that can be opened by right clicking inside the diagram editor. There, any commands or actions can be structured as needed. It can be customized by implementing and registering the `ContextMenuItemProvider` to the server or implementing and registering the `MenuContribution` interface to the Theia frontend module. [**glsp-doc**, **glsp-repo**]
- **EditLabelUI**: Labels of nodes and edges can be edited by double-clicking on the label. The EditLabelUI provides an input popup to edit the label text. [**glsp-doc**, **glsp-repo**]
- **Custom UI Components**: Custom UI extensions have to extend `AbstractUIExtension` that provides a base HTML element and can then be registered to the client. The base class also provides functionality to show, hide, or focus the element. These UI extensions can also be enabled over a `SetUIExtensionVisibilityAction` from the server. [**glsp-doc**, **glsp-repo**]

GLSP uses Sprotty [**sprotty-repo**], a web-SVG-based diagramming framework, to render the diagrams. The graphical model of GLSP called *GModel* is based on the *SModel* of Sprotty and works as a compatible extension. The graphical model is composed of shape elements and edges. They are organized in a tree, that starts with the `GModelRoot`. There are several base classes, that can be extended and also new types can be added. The `GEdge` represents an edge between two nodes or ports. Four classes inherit from `GShapeElement`, which represents an element with a certain shape, position, and size. They can also be nested inside another `GShapeElement`. The `GNode` can have `GLabel` or `GPort`, which represents a connection point for edges, as children. The `GCompartment` can be used as a generic container to group elements. The Java server uses EMF to handle the graphical model internally, to profit from the command-based editing capabilities of EMF. To send the graphical model to the client, it is serialized into JSON using GSON [**gson-repo**] and then sent over JSON-RPC. [**glsp-doc**]

The layout of a graph is divided into macro and micro layouting. The macro layouting, which arranges the nodes and edges of the model, is done by the server. The client does the micro layouting by calculating the positioning and size of elements within a container element. [**glsp-doc**] For the macro layouting, GLSP

provides a notation model, that persists the position and size of the elements in a separate notation XMI file. The notation diagram can be added to the `GModelState` and then used in the `GModelFactory` to specify the layout. [**glsp-repo**] GLSP also provides a `LayoutEngine` interface, that can be used to layout the elements of a graph that have no persisted layout yet. [**glsp-doc**]

GLSP also provides an interface to validate the model. With the `ModelValidator` interface, specific validation rules can be defined by the server. The validation returns a list of markers that can be an info, warning, or error. The markers are then displayed in the GLSP client. The markers can also be integrated into the Theia Problems View.

III. Related Work

A. Scientific Literature

B. Existing Tools and Technologies

There are many existing tools for model transformations. **kahani2019survey** created a survey in **kahani2019survey** of various model transformation tools. They classified 60 different tools, including Henshin. In Figure 4, you can see how many tools provide specific execution environments. 73% of the tools provide plugins for the Eclipse IDE, and 20% of the tools are integrated or dependent on other IDEs. 18% have no IDE support, and only two tools are web-based. In total, 89% of the tools have external dependencies such as an IDE or other tools. Dependencies often complicate the installation and usage of the tool. [**kahani2019survey**]

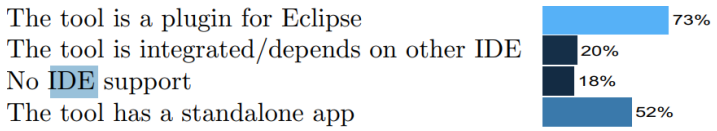

Figure 4. Execution environments of model transformation tools. Image obtained from [**kahani2019survey**]

One web-based tool included in the survey is A Tool for Multi-Paradigm Modeling (AToMPM) [**atompm**]. It is a web-based modeling tool to create Domain Specific Modeling Language (DSML) environments, performing model transformations and manipulating and managing models. [**atompm**] It was created in **atompm** and supports all model transformations that are based on T-Core [**tcore**], a minimal common basis that allows interoperability between different model transformation languages. [**tcore**] Metamodels can be defined with a simplified UML language. The graphical modeling environment offers debugging and the ability to collaborate and share modeling artifacts in the browser. [**atompm**]

There are also other web-based tools for MDE. WebGME [**webGME**] is a web-based modeling tool, created in **webGME**. It allows to collaboratively design DSMLs using model versioning and broadcasting changes to all active users. It supports prototypical inheritance, where any model can be instantiated recursively, so changes are propagated down the inheritance tree. It also provides scalability, collaborative modeling and model versioning. Metamodels and compositions can be created with WebGME, but no graph transformations can be applied to a model. Even though model transformations are not possible, the editor was one of the first solutions for web-based modeling tools. [**webGME**] The software provides extension points to customize or extend the software, but no model transformation capabilities were added by any available extension. [**webgme-website**] The tool is still hosted and maintained, to be used for free. [**webgme-website**]

WebDPF [**webDPF**] is another web-based modeling tool, published in **webDPF**. Compared to WebGME and AToMPM, it supports model navigation and element filter capabilities, a JavaScript editor for writing

predicate semantics, reusability of transformation rules, partial model completion, and a termination analysis. These features try to improve the usability of the tool. [**webDPF**] Even though the tool had improvments upon existing tools, the originally mentioned hosted WebDPF portal is offline by now.

There is also a GLSP-based Ecore metamodel editor, created by the GLSP development team. It was implemented with the GLSP version 0.9 but never updated further. It allows to create and edit EMF Ecore models in a Theia web editor. Even though the project cannot be used directly, due to the use of another source model format and breaking changes in major updates of the GLSP framework, it provides various classes that can be used as a template for the Henshin Web Ecore viewer. One example is the factory code that maps the EMF Ecore model to the graphical model. [**glsp-ecore-repo**] The findings show, that there are many existing model transformation tools, but only very few web-based solutions, that provide an easy entry into MDE and model transformations. Henshin web tries to fill this gap.

C. Comparison and Gaps

IV. Requirements Analysis

The purpose of this chapter is to systematically identify, analyze, and document the requirements of the software system developed in the context of this thesis. The chapter outlines functional and non-functional requirements as well as the system stakeholders and constraints.

A. Stakeholders and User Needs

- **Students**: Students who want to learn about MDE and transformation rules, want a very simple and intuitive entry into the topic. Trying out transformations in the browser is a good start, without having to install a lot of software. For Students the core functionality is sufficient, as they only want to try out transformation rules and learn how they work.
- **Researchers**: Researchers, that are researching for MDE and come arcross Henshin, want to be able to test or try out model transformations of Henshin. For Researchers the core functionality could be sufficient, but editing capabilites of the transformation rules and metamodels are pracitcal for them.
- **Software Engineers**: Software engineers, that are using MDE, want to be able to test their transformation rules. They want a powerful editor and a colaborative environment to work on their models. For Software Engineers the defined additional functionality as well as some code generation support are needed to cover their needs.

The different Stakeholders show, that for more features the application provides, more users can be reached. With more features and use cases the application can cover, it provides more value for enterprise users, that work for production systems.

B. System Scope and Context

The Eclipse IDE plugin of Henshin works as a template for the functionality of the application. It provides functionality to create, edit and apply Henshin transformation rules for Ecore metamodels on XMI instances. To create a full enterprise application, that will get used for projects in the industry, the application has to also provide very similar functionality. The defined core functionality is the minimum set of features that the application must provide to be useful for the users. They are only a subset of requirements to provide a full web-based copy of the Henshin Eclipse plugin.

The core functional requirements extends already existing functionality, that Eclipse Theia and GLSP already provide. Theia provides various views like the explorer for basic file management, including opening, saving, closing, creating and deleting files, a problems view, a integrated terminal, or edit operations like copy or paste. GLSP provides default functionality for each graphical editor. That includes selection of elements, moving nodes, realinging edges, zooming, or moving and reseting the viewport. Most of these features can be further configured to be able to create an editor fitting the specific needs.

C. Functional Requirements

The main use case that the application should support is that a user can try out transformation rules on EMF XMI instance files. In this usecase, the user already has a metamodel and transformation rules. He wants to test the transformation rules on various instances in an accessible, intuitive, and easy to use graphical editor. From that use case the following core functional requirements can be derived.

Core Functionallity:

- EMF XMI instance files should be displayed in a graphical editor. That contains the nodes, edges and attributes of the model.
- The XMI instance editor should provide editing functionality to create, update and delete nodes, edges and attributes.
- In the XMI instance editor all applicable transformation rules should be listed. When a rule is selected to get applied, all parameters have to be specifiable.
- When a rule gets appplied, the graphical editor of the XMI instance should be updated to reflect the changes made by the transformation rule. The application should also support undo and redo functionality for the applied transformation rules.
- Henshin transformation rule files should be displayed in a graphical editor. That contains the nodes, edges and attributes of the model and their action types. The user should be able to switch between all rules of a *.henshin* file.
- EMF Ecore metamodel should be displayed in a graphical editor. That contains the nodes, edges and attributes of the metamodel.

Next to the core use case, the second use case is that a user wants to create a full transformation language from scratch, that can be used to model and test the system and generate production code from it. In this use case, the user wants to create and edit metamodels and transformation rules. To support this use case, the following additional functional requirements are defined:

Additional Functionallity:

- The Henshin rule editor should provide editing functionality to create, update and delete nodes, edges, attributes and their action types.
- The Ecore metamodel editor should provide editing functionality to create, update and delete nodes, edges and attributes.
- Henshin transformation units are also listed in the XMI instance editor and can be applied.
- Show the possible transformation rule matches in the XMI instance editor, when selecting a transformation rule.
- Provide the functionality to apply a State Space analysis on a XMI instance.
- Provide the functionality to apply a conflict and dependency analysis on a XMI instance.

There exist many more use cases for model transformations and MDE in general. The application can grow to a web-based platform for MDE in th future. Additional functionality will be disscused in section X-B but these usecases are not scope of this thesis.

D. Non-Functional Requirements

In addition to the core functionality, the system must meet several non-functional requirements:

Non-Functional Requirements:

- The application should be web-based and preferably accessible via a web browser.
- The application should be responsive and work on different screen sizes. It does not have to support mobile devices and touch interactions, since GLSP is also not supporting touch interactions [**glsp-repo**].
- The application should be user-friendly and intuitive to use. For that, the application should follow the design principles of GLSP and Eclipse Theia. That includes the use of views of theia, like the exploerer and the predefined UI contorls of GLSP, like the tool palette or the context menu.

E. System Constraints

One constraint is the use of Henshin as a tranformation language. Henshin is a Java-based framework, which menas that the application needs a possibility to run Java code in the backend. The easiest way for that is to use a Java-based backend, that can directly use the Henshin SDK code. The use of Henshin also brings the constraint that, metamodels and instances are based on EMF.

Another constraint is the use of web-based technologies and preferably a resulting web application. For model transformations, there exist many applications, but not many of them are web-based. This constraint is also a non functional requirement and was also motivated in previous sections. The initial version of the application will support English only.

V. System Design and Architecture

In this chapter, the architecture of the system is described. The system is designed to achieve following goals. The system should be modular and easily extensible to allow future extensions towards a full model transformation platform for production use cases. The system should also be maintainable. In the folowing sections, the high-level architecture following the GLSP architecture is described. Then the design of the components, control flow and data models is described. In the end the UI design is explained.

A. Design Decisions

In the section II the used frameworks and technologies were described. The selection of these framworks still leave some open design decisions. One open decision was which platform integration to use for the GLSP client. GLSP can be used as an extension for Eclipse Theia or VS Code, a plugin for the Eclipse IDE or as a standalone web application. They can also be used in combination, but to avoid overhead and complexity, only one platform integration is initially used. In table I the different integration options are compared. Since the integration into an existing IDE fits the graph editors, the standalone editor is not an option. For that, many additional features like a file explorer have to be implemented. The integration into the Eclipse IDE is also not an option, since it is not based on web technologies and therefore not satisfying the requirements of the project. Between the Eclipse Theia and VS Code integration, Eclipse Theia is providing more flexiblity in the usage and deployment of the application. Next to the usage as an extension that can be added during runtime, Theia also provides the option to bundle your own IDE including the GLSP graph editors. That

makes it deployable as a complete application, where no additional plugins are needed. This flexibility is the main reason to choose the Eclipse Theia integration as the inital main platform for Henshin Web. With that usage and deployment flexibility, different additional platform integrations are probably not needed in the future.

Table I
COMPARISON OF GLSP PLATFORM INTEGRATIONS

Criteria	Eclipse Theia	VS Code	Eclipse IDE	Standalone
Deployment Options	Web-app, Desktop (Electron)	Desktop, Web-app	Desktop	Custom (Web or Desktop)
Extendability	Access to all Theia internal APIs	Through VS Code Extension APIs	Moderate, via Eclipse plugins (OSGi-based)	Fully customizable (with own implementations)
Provided Environment	Complete IDE	Complete IDE	Complete IDE	No other features included
Result Format	Own IDE or as a plugin	VS Code extension	Eclipse IDE plugin	javascript based web editor module
Dependencies Needed	browser	VS Code Desktop or browser	Eclipse IDE	browser

Another decision was to select a edge routing style. GLSP provides two different routing algorithms, the Manhattan and Polylime stlyes. The Manhattan style was invented by **manhattan** to achieve wire length optimization in circuit design. It only uses vertical and horizontal lines to connect nodes. [**manhattan**]. The conncetion can be split into multiple segments, changing from a horizontal to a vertical line or the other way round to create a stair like connection between nodes. The Polyline style on the other hand uses straight lines to connect nodes. They line can also be split into multiple segments with arbitrary angles between them. For this use case, the main aspect is the clarity and readability of the graph. You can see the comparison of the two styles in figure 5. Advantages of the Manhattan style are that can prevent edge crossings, and therefore can help reduce visual clutter. In complex diagrams with many nodes and edges, it is easier to trace the horizontal and vertical lines. On the other hand, it can overlap with other edges, which can make it hard to follow the edge. Especially named edges that overlaps partly with another edge can't be followed without clicking and highlighting it. You can see that in figure 5 between the *Bank-Account* and the *Account-Client* edge. To prevent that, the edge routing needs be stored in the notation file to be able to persist changes in the routing that remove overlapping edges. The Polyline style on the other hand is generally simpler and more compact. It can get very clutterd with many edges crossing and other nodes overlapping the diagonal lines. Metamodel, transformation rules and instances are typically not that complex, so that a simple line bewteen nodes is sufficient and additonal edge segments are not needed. Because of that, the edge placement doens't have to be stored in the notation file. The edge automatically aligns itself when a node is moved. The rotation of the edge label that it runs parallel with the line supports the simple and compact design of the Polyline style. To additionally prevent crossing lines, a option to dynammically hide the root node and its edges in XMI graphs is introduced.

One main question in the UI design was where to put the selection of the transformation rules of a *.henshin* file. There are several options to display the rule seletion. The first option is to add the list of rules to the tool palette as an additional pallete group next to the nodes and edges. Here no additional new UI element must be placed in the graph editor, but the main focus of the tool palette is to provide editing tools of the graph elements. Swapping between the rules doesn't fit into the main purpose of the tool. Another option

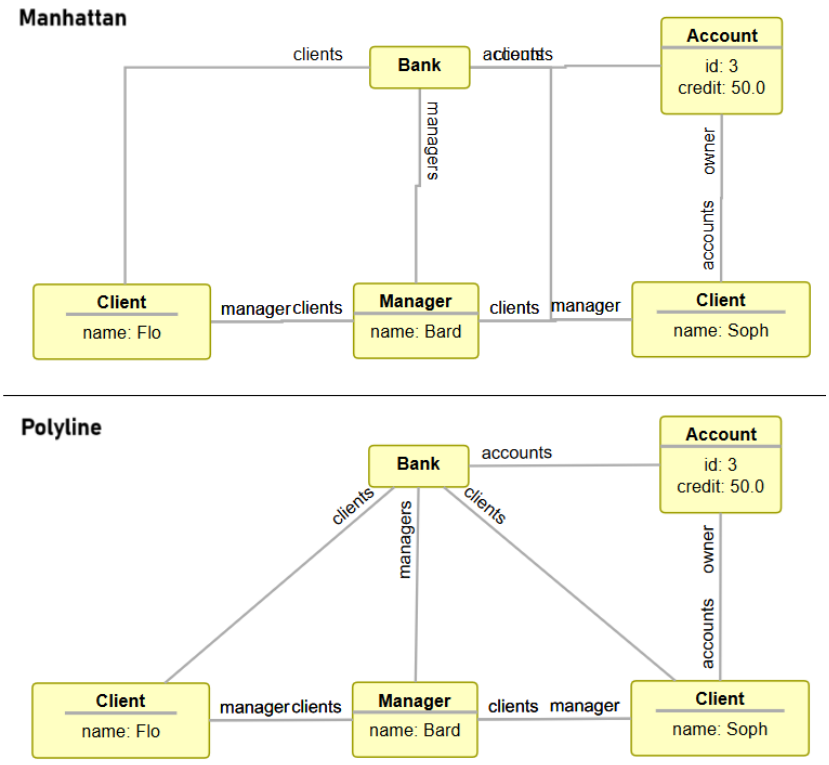

Figure 5. Visual comparison of the two edge routing styles of GLSP

is to create a custom UI element that is displayed in the rule graph editor. I would be easy to implement, directly integrated into the graph module and platform independent. But it would take up additional space in the graph editor view. The rule graph should be the main focus of the application and should have enough space to display the graph elements, especially for larger graphs.

For these two options, the user also has always to switch to the rule graph editor first to select a rule. It can negatively impace toe user experience. That would not be the case if the rule selection is integrated into the Theia explorer. Since the custom explorer is used anyway to select between different instance, transformation rule or metamodel files, it is a good place to also select the transformation rules. The explorer can also be collapsed to save more space for the graphs and for very many rules it automatically supports scrolling. Extending theia internal API makes it more effort to add aditional GLSP platform integrations, because the custom changes need to be newly implemented for the new platform, it may even not support the same extendability. Since the theia integraion provides the most options to use and deploy the application, more platform integrations are probably not needed. Aditionally, the extension of the theia explorer prevents the occupation of additional space in the GLSP editor widget to select a rule. This improvement of the user experience and intuitiveness outweights the possible additional effort for new platform integrations. The implementaiton of the custom explorer is described in section VII-C5.

B. Following the GLSP Architecture

The system is based on the GLSP architecture, that uses a client-server architecture. The client and server communicate via a websocket connection and JSON-RCP. The GLSP server can be implemented with Java

or Node.js, but due to the constraint that Henshin is implemented in Java, the server is also implemented in Java. The client is implemented in TypeScript. GLSP provides a defined protocol for the communication between client and server, which can extended with custom commands and actions. The communication is performed using Action Messages, that can be sent from the client and the server to each other or also to itself. The client and the server have Action Handlers, that process the Action Messages and perform the corresponding actions. Each client connection starts its own server instance, therefore each server is only responsible for one client. [**glsp-doc**] Since each client needs to be able to display three different graph editors for different file types, the server consists of three diagram modules. Each diagram module defines a different diagram language. The `XMIDiagramModule` is responsible for the editor of XMI instance files, the `RuleDiagramModule` is responsible for the editor of Henshin rule files and the `EcoreDiagramModule` is responsible for the editor of Ecore metamodel files. In figure 6 you can see the high-level architecture of a server and client instance. The architecture of the three diagram modules is quite similar. Each diagram module has a `ModelState` which is the central statefull object within a client session [**glsp-doc**]. The `ModelState` is accesed by all other services and handler and represents the current state of the actual source model. GLSP supports the integration of EMF models as the underlying source model for the diagrams by default. For that The `EMFSourceModelStorage` can load a EMF file as a `RessourceSet`, that is then attached to the `ModelState`. That allows an simple integration of the Henshin SDK, since it based on EMF and provides a `HenshinRessourceSet` can be loaded directly over the EMF integration of GLSP into the `ModelState`.

The `ModelState` of each diagram module also contains an index and a notation model for the layout of the elements in the graphical editor. To be able to have a consistent layout of the elements, not changing after every reload or action, the position and size of each element for each model file is stored in a seperate *.notation* file. The index of the `ModelState` is used to map the elements of the source model to the graphical model of GLSP. For each diagram module, the indexing is implemented in a different way. (see section VII-C4 for more details).

Another important part of each diagram module is the `GModelFactory`, which is responsible for creating the graphical model that is sent to the client from the source model. Since metamodel, transformation and instance EMF model are strucutred differently, each `GModelFactory` of each diagram module is implementing its own mappings.

C. Data Models and Structures

All three diagram modules have an EMF based source model. For the Ecore metamodel and the XMI instances, The standard data model of EMF is used. As described in section II-B different implementations of `EObject` are used, representing all used elements like nodes, attribues or references. For the Hensin transfomations model, the data model of the Henshin Software Development Kit (SDK), that builds upon the EMF data model, are used. No additional data structures are needed, since every created domain model is based on the EMF data model. The data model of the graphical representation is provided by GLSP. It can be extended with custom elements, but the default elements are sufficient for the current use cases.

The user has to select or create a workspace in the UI, where all the source models are located. Each workspace for Henshin Web has to be in a specific structure. It should contain one *.ecore* metamodel and one *.henshin* transformations file. Additionally, arbitrary *.xmi* instance files can be added. All of these files should be stored in the root folder of the workspace. When creating a new model file or opening it for the

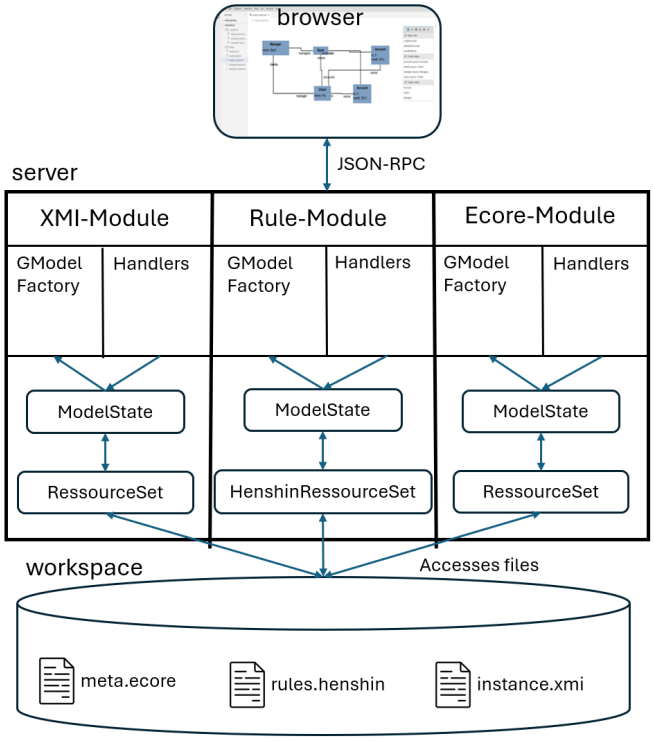

Figure 6. High-Level Architecture of the System

first time, a new notation file is generated and stored in the *.notation* subfolder of the workspace. These notation files are not displayed in the theia explorer.

D. GLSP Client Structure

The GLSP client is divided in different main modules. The *henshin-glsp* module is responsible for platform independent code. It contains client side action handlers, custom UI extensions and custom graph elements. This module is used by the three theia specific modules, that are responsible for the integration of the GLSP client into the Eclipse Theia framework. There is one module for each diagram type. The target specific code is loacted here. One example is the customization of the Theia explorer view, that it also displays all rules of a *.henshin* file and hides the notation files. These three theia extensions are then combined in the *henshin-browser-app* module, that has no additional code, but only combines the three theia modules into one application.

E. User Interface Design

The design of the user interface is based on the design principles of GLSP and Eclipse Theia. For editing the graphs, the main UI element is the tool palette on the right side of the graph editor. It lists all available nodes and edges, that can be added to the graph as well as the transformation rules that can be applied. It also contains a set of predefined GLSP actions, which are switching between selection mode, deletion mode and marquee mode, as well asreseting the viewport and search for tool palette entries. For keyboard usage, ther is also the command palette, that can be opened with the [Ctrl + Space] shortcut. It opens a searchbar with

a list of options below. Here all editing operations are registered listed and can be performed by searching or navigating through the list and selecting the desired operation.

The design of the custom UI elements like the parameter selection form or the display of the transformation rule information follows the design of GLSP. The design of Theia uses a flat design with minimal gradients, shadwos or 3D elements. Compared to that uses GLSP a more 3D-like design with shadows and gradients, because the UI elements need to be on top of the main graph plane. That shows that the UI elements are not part of the graph, but are additoinal elements to interact. The tool palette, comamnd pallete and context menu all use shadows and rounded eges. New custom UI elements like the parameter selection also use the same shadow and rounded corners to also show that they are not part of the graph, but elements to interact with.

The colors of the custom UI elements follows the color theme of Theia. Important is that Theia uses the dark mode as a default theme. Graph editors typically use a light backgroung. All custom UI elements are designed to adjust to the dark mode and use the according colors as the default dark Theia elements. The final UI can be seen in appendix 9, 10 and 11.

VI. Deployment and Usage

A GLSP editor can be deployed and used in production in various ways. GLSP provides platform integrations for the Eclipse Desktop IDE, Eclipse Theia, VS Code, and as a standalone web application. Each integration brings different integration possibilities, deployment, and usage options for the editor. [**glsp-doc**] The main considerations for the deployment and usage are:

- The user should need as few dependencies as possible. Dependencies are a browser runtime, an IDE to install, or an extension to install.
- The app should be easy to access. Possible barriers are the creation of an account or the installation of dependencies.
- Using a self-hosted server or a cloud service. With a self-hosted server, the user has full access of local files to open and edit. With a cloud service, the user has to upload and download files to the server.

To use GLSP as a standalone web application, a dependency injection container with the custom GLSP client is added to a TypeScript browser application. Like that the editor of a certain file as a data source can be displayed. When the app is hosted, no other dependency than a browser runtime is needed to use the standalone diagram editor. [**glsp-client-repo**] This option provides the most flexibility, as it can be used in any web application, but also requires the most effort to implement, when developing a complete editor. All features, like file management, window management, or other features a IDE brings, need to be implemented by the developer. [**glsp-client-repo**] For our use case, the standalone web application is not an option, as these additional features are needed.

The other GLSP integrations are IDE integrations and therefore provide many features out of the box. For the Eclipse IDE integration, Eclipse has to be installed, and the GLSP plugin has to be added to the Eclipse installation. The plugin can be installed from the Eclipse Marketplace or manually by downloading the plugin jar file. [**eclipse-doc**] The VS Code integration also provides this option. The IDE can be installed and the GLSP editor can be added as an extension. The extension can be installed from the Marketplace or manually using a *.vsix* file. [**vscode-doc**] The GLSP VS Code integration can provide a *.vsix* file. [**glsp-repo**] VS Code is the most used IDE. 73.6% of developers use VS Code due to the survey of **stackoverflow2024survey** In **stackoverflow2024survey** [**stackoverflow2024survey**]. An advantage to Eclipse is that VS Code provides

a browser version, which brings the same capabilites as the desktop IDE. [**vscode-doc**] So this integration provides the advantage that no IDE has to be installed to be able to use Henshin Web. The user can open VS Code, add the extension, and directly open a metamodel, rule, or instance model file and start editing.

The Eclipse Theia IDE is not as widely popular as VS Code [**stackoverflow2024survey**], but its focus is not to provide a ready IDE but to provide tools to create custom IDEs. The Eclipse Theia project is part of the Eclipse Foundation and is used as a basis to create your own IDEs based on web technologies. [**theia-doc**] They provide the Theia IDE that acts as a template editor and can be downloaded and used on all common operating systems or used in as a web editor in the browser. Due to the focus on providing a framework to build custom IDEs, Theia provides more options to use extensions and plugins to extend the functionality. You can see the options and their architectural integration into Theia in figure 7.

- **VS Code extensions** Theia provides the VS Code extension API, so that existing VS Code extensions can be used in Theia. They only interact with the API and therefore can be installed at runtime.
- **Theia plugins** are working like VS Code extensions. They interact with the Theia plugin API and can also access the VS Code extension API. They can access some Theia specific features, that VS Code extensions cannot access, like directly contributing to the frontend. They can also be installed at runtime, or be pre-installed at compile time.
- **Headless plugins** are also working like VS Code extensions. They can also be installed at runtime and can access custom extended Theia backend services.
- **Theia extensions** are the core architecture parts of Theia. Theia is fully built using Theia extensions in a modular way. The template Theia IDE contains Theia extensions, including the core. Custom Theia extensions can be developed and added to Theia with full access to all Theia functionality via dependency injection. They need to be installed at compile time. [**theia-doc**]

The GLSP Theia integration is creating a Theia extension, that is packed into a custom Theia IDE. It is also possible to use the GLSP VS Code integration that provides a VS Code extension, that can also be added to a Theia IDE at runtime. [**glsp-repo**] The option to use the diagram editor in the browser makes the GLSP Eclipse integration not interesting for Hensin Web. VS Code has the advantage of popularity and simplicity to use the editor without any registration or installation. Eclipse Theia has the advantage of modularity and further extensibility. Further features can be added in the future to provide a web-based environment for MDE. Theia also provides different ways to deploy a Theia IDE. These considerations show that the Theia integration is the best option for deploying the Henshin Web editor. Theia combines the advantages of browser-based access, modularity, and extensibility.

There are different options to provide a GLSP Theia application. The Theia editor, consisting of the TypeScript client and the Java server, can be hosted in the cloud and accessed via a web browser. The Eclipse Foundation provides the Theia Cloud project [**theia-cloud-doc**] to deploy Theia based products on Kubernetes clusters [**kubernetes**]. Theia Cloud introduces three custom Kubernetes resource types. *App Definitions* contain all necessary information about the Theia based product. *Workspaces* define persistent storage solutions, where metamodel, rule, or instance model files can be stored for each user. *Sessions* are acting as a runtime representaions. Theia Cloud includes components like a landing page, authentication, authorization, a cloud monitor, and a cloud operator, that deploys sessions and manages workspaces. You can see the different components and their interactions in figure 8. The service provides two preconfigured configurations for quickly trying out Theia Cloud on a cluster. [**theia-cloud-doc**]

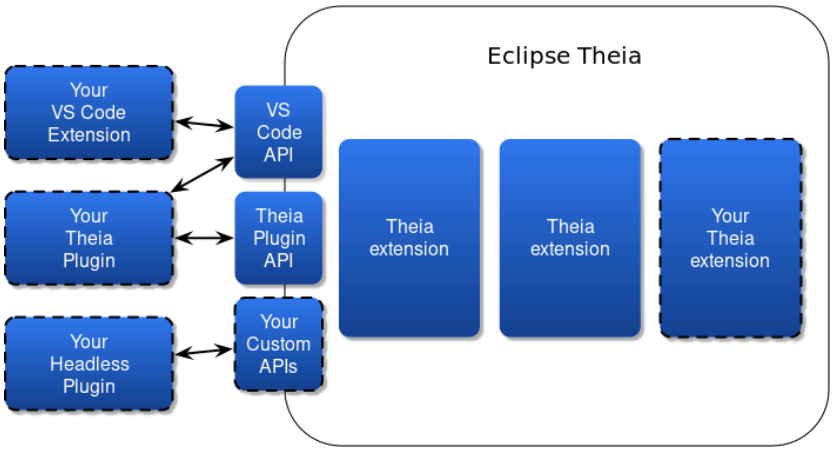

Figure 7. Theia high level extensions and plugins architecture. Image obtained from [**theia-doc**]

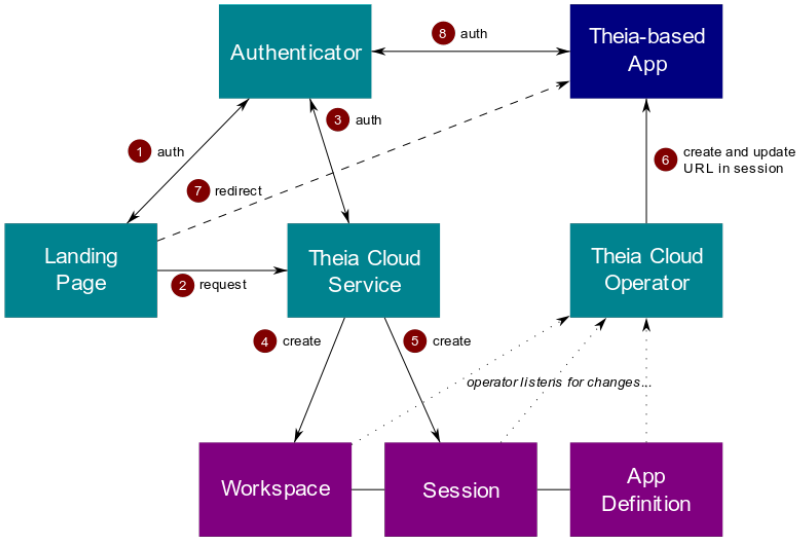

Figure 8. Interaction between Theia Cloud components. Image obtained from [**theia-cloud-doc**]

Because of the limited file access of the browser, the user has to upload and download all files to the server to use them. To be able to access the local file system of the user directly, the server needs to be hosted locally. For that, GLSP Theia application can be hosted in a Docker container. [**docker**] The Docker container can contain the Java server and the TypeScript client, that are started together. The user can then access the editor via a web browser. On a machine with a Docker environment, this solution can be started locally in an easy way and has the access to the file system. The Docker container can also be used to deploy the application on a server so that it can be accessed by multiple users. The single Docker container solution doesn't provide as much scalability as using a cluster with Theia Cloud.

The GLSP Theia application can also be used as a desktop application. Theia uses Electron [**electron-repo**] to bundle the application into a desktop application, that can be installed via an installer. This approach also provides access to the local file system, since the electron application works like a self-hosted web application, and therefore the GLSP Java server is started locally. All in all, the GLSP Theia integration

provides all different options to use the Henshin Web editor. Further clients can always be added later if needed.

In table II the different deployment options are compared. The self-hosted Docker and Desktop Electron bring no costs to provide the application. Here the Desktop option is easier to install and use, as no external dependencies need to be installed before. Even thought it doesn't use the browser, it uses web-based technologies. Theia Cloud on the other hand brings many features for deploying the application with authentication, authorization and predefined workspaces with it. On the other hand, it brings a more complicated deployment and costs for hosting the kubernetes cluster. To sum it up, the two best options are to use Electron or Theia Cloud.

Table II
COMPARISON OF GLSP THEIA DEPLOYMENT OPTIONS

Option	**Self-hosted Container**	**Cloud Hosted Container**	**Theia Cloud**	**Desktop (Electron)**
Installation Effort	Local Docker Setup required	None (access via browser)	None (creating an account)	Installing over a standard installer
Dependencies	Docker runtime, web browser	Web browser only	Web browser only	Application installer
Multi-user Support	Single user	Multi-user possible but no shared editor	Built-in shared workspaces, but no shared editor	Single user per installation
Hosting Requirements	Local	Cloud service (Container hosting)	Kubernetes cluster	Local machine
Cross-platform	Yes (via browser)	Yes (via browser)	Yes (via browser)	Platform-specific builds
Offline Usage	Yes	No	No	Yes
File System Access	Full local access	Upload/download required	Upload/download required	Full local access
Costs	No Costs	Cloud server costs	Cost of Google Cloud Kubernetes cluster	No Costs (maybe provisioning of installer)

VII. IMPLEMENTATION

This chapter describes the development process and shows the solution and implementation of specific problems, that appeared while implementing the application. The first challenge was to integrate the Henshin SDK into the GLSP project. Another challenge was to index the elements of the different EMF models and formats. One big UI desicion was where to place the selection of transformation rules in the application.

A. Development Process

The development of the Henshin Web GLSP editor was done by one person in a time span of about 6 months. Derived from the functional requirements (see section IV-C) the project was split into 7 milestones. The milestones were defined as follows:

- **Milestone 1:** Setup the project and create a diagram editor that can display *.xmi* files.
- **Milestone 2:** Create editing capabilities for XMI instance files.
- **Milestone 3:** Henshin transformation rules can be displayed and applied to the instance model.
- **Milestone 4:** Create an additional diagram editor that can display Henshin rules.
- **Milestone 5:** Create an additional diagram editor that can display Ecore metamodels.
- **Milestone 6:** Create editing capabilities for Henshin rules.

- **Milestone 7:** Create editing capabilities for Ecore metamodels.

Each milestone was split into smaller issues. The first milestone was used to create Proof of Concept (POC) to test the integration of Henshin into a GLSP project. In this phase, the focus was to get to now how the frameworks EMF, GLSP, Henshin and Eclipse Theia work. Even though GLSP provides a well structured documentation and project templates, they didn't cover many use cases for the development of Henshin Web. Henshin also doesn't provide an documentation of their API. Therefore for these frameworks a lot of source code reading and understanding was needed. Git was used as a version control system. The development of a milestone was don in a seperate development branch. When all features of the milestone were implemented, the state of the application was additionally tested and then merged into the main branch.

B. Tooling and Environment

For the development of Henshin Web, VS Code [**vscode**] was used to develop the client and InteliJ IDEA [**intellij**] was used to develop the Java server. For understanding the source code of not well documented frameworks, the use of Chatbot Agents was very helpful. I used Github Copilot in VS Code with the model Claude Sonnet 4 [**claude˙sonnet**] in agent mode. Because it has access to the source code of the dependend frameworks, it can search for specific classes or methods or explain certain concepts. Git was used as version control system and GitLab was used as a remote repository. It was also used for the project management, where the milestones were defined and the issues were created. A issue board was used to show the current sate and progress of the project. The GitLab package registry was also used to store the Henshin maven packages, to be able to access them from the GLSP project. These packages are then available to every contributor of the GitLab, that want to develop on the project. More about the creation on Maven packages will be shown in the next section.

C. Code Examples...

This section shows the solution and implementation of specific problems, that occured during the development process.

1) Integration of Henshin into a GLSP project: The Henshin source code provides both the Eclipse IDE plugin and a Java SDK for using the Henshin interpreter. The project of Henshin is structured as an Eclipse project and is available as a set of Eclipse plugins and features. [**henshin-repo**] On the other hand, GLSP projects typically use a Maven project structure. [**glsp-repo**] To add dependencies to a Maven project, the dependencies should ideally be available as Maven artifacts. However, Henshin doesn't provide a Maven artifact, since that is not needed for an Eclipse plugin. The Henshin version 1.8.0 is compatible with JDK 11 and higher. GLSP version 2.3.0 has the prerequisite of JDK 17. Therefore, the versions are compatible to run together. The Henshin code consists of 45 plugins, of which 22 are contained in the Henshin SDK, that we need as a dependency in our Henshin Web GLSP project. Each plugin can be downloaded as a JAR file. To create Maven packages from the JARs, a PowerShell script is used. It reads all JARs files from a folder, renames them to the correct Maven artifact name, creates a basic `pom.xml` file for them, deploys them to the GitLab package repository, and creates a list that needs to be included in the Maven `pom.xml` file of the GLSP project. A package of each plugin is created, because for the Henshin Web editor, only some parts of the Henshin SDK are needed. To use the Henshin model package, the additional dependency of the Nashorn JavaScript engine [**nashorn-repo**] is needed. The Nashorn engine is used to execute calculation expressions of transformation rules. [**henshin**]

2) GModelFactory: The heart of a GLSP server diagram module is the `GModelFactory`. It is responsible for creating the graphical model from the source model. In listing 3 you can see the implementation of parts of the creation of the graphical nodes. The method `fillRootElement(GmodelRoot newRoot)` gets called when a new graphical model should be created. It fetches the source model elements from the `ModelState`, iterates over them, and creates `GNode` elements using a builder pattern. In the method `createNode(DynamicEObjectImpl eObject)`, it is configured how the node should look in the editor. It sets the id, adds CSS classes and configures rounded corners. It also sets the type of the node. The type can be a default type, or custom types, that have their own customized client implementation. In listing 3 you can see for example that the root node of the XMI instance model gets a different type. The type is used to configure that only one root node can exist and that it cannot be deleted, if other child nodes exist. The method `applyShapeData(eObject)` adds the layout information from the notation model. In the `GNodeBuilder` also the builded child elements like the header or the attributes are added. This creates a tree strucutre of graphical elements, that are all atached to the `GModelRoot`. The `RuleGModelFactory` and the `EcoreGModelFactory` work similar to the `XMIGModelFactory`, but they create different node types and handle the source model elements differently.

3) Layouting: EMF Ecore metamodel files (*.ecore*), Henshin rule files (*.henshin*) and EMF instance files (*.xmi*), don't contain information about the position or size of elements in a graph. [**emf**, **henshin-repo**] To provide a good user experience, the graphical editors need to provide a consistent macro layout for nodes and edges. Newly created nodes should not overlap with existing nodes, and the nodes should stay in the same place after reloading the editor. In general, the GLSP server is responsible for the macro layouting. [**glsp-doc**] GLSP provides multiple options to layout the graph. The interface `LayoutEngine` can be used to create a custom layout algorithm, that is applied after the creation of the graphical model from the source model. GLSP provides the `ElkLayoutEngine` implementation, that uses the Eclipse Layout Kernel (ELK) to layout the graphical model. [**elk-engine**] With ELK, different layout algorithms can be used and additionally configured. Even though ELK provides much flexibility for the layout, the layout is newly created after every change to the source model. This means that the layout is not consistent and nodes can move around after every change. To provide a consistent layout, the position of nodes need to be stored in addition to the source model. The GLSP server provides a notation model, that can be used to store the position and size of nodes and edges. [**glsp-repo**] This brings the overhead of updating the notation model every time when the source model is updated. GLSP provides classes to make the synchronization of the notation model easier. The notation model is stored in an additional *.notation* file, that is loaded together with the source model and applied to the graphical model in the `GModelFactory` using the `NotationUtil.applyShapeData(shape, builder)` method. To capture changes of position and size of nodes, the GLSP client sends the `ChangeRoutingPointsOperation` and `ChangeBoundsOperation` operations automatically when moving or resizing a node or edge. At the server, the corresponding handlers are updating the notation model using commands to provide undo and redo functionalities.

To achieve layouting in the Henshin Web editor, notation models for the metamodel, Henshin rules, and instances are used. The *.notation* file is created when the source model is loaded for the first time. Here, ELK can be used to create a fitting initial layout. For the XMI instance models, when the graphical model gets created in the `GModelFactory`, the shape data from the notation model is added to the EMF elements over an EMF `Adapter`. Each EMF `EObject` has a list of adapters, that can be used to store additional informa-

tion. [**emf**] To connect the notation to an element, the `NotationAdapter.getOrAssignNotation()` method checks if the element already has a notation, either returning the existing notation or appending a new Adapter with the notation information. For the Henshin rules and the Ecore metamodels, the notation element mapping is stored in the model index that is contained in the `ModelState`. The reason for the different indexing approaches and their implementations will be explaine in the next section.

4) Indexing EMF models: Like the layout information, EMF Ecore metamodels and EMF XMI instance models don't by default contain unique identifiers for nodes, edges, or attributes. [**emf**, **emf-repo**] The graphical model of GLSP on the other hand uses identifiers for each element that is displayed. If no identifiers are specified when creating the graphical elements, GLSP generates its own internal unique identifiers. These identifiers are used in edit operations like renaming or deleting a node, where the graphical element needs to be mapped back to the source model element and only the identifier of the graphical element is sent from the client to the server. To be able to map the graphical element back to the source model element, custom identifiers need to be stored. Additionally, during the transformation of the source model into the graphical model, elements need to be accessed multiple times. For example, a source node is accessed over the EMF package when it is mapped into a `GNode` and then again for all its connected edges and attributes. An indexing of the elements avoid multiple lookups in the EMF source model. To be able to support any created domain meta and instance models and to prevent prerequisites for the use EMF models in Henshin Web, the GLSP server needs to create own inexes for the elements of the source model.

The idexing of the three different source model types is implemented in differnt ways, due to the different internal structures and stored informations. The simplest approach is used for the Henshin rule model. Henshin already creates identifiers for each node and edge of a transformation rule. These identifiers are also stored in the *.henshin* file. When building the graphical model, the identifiers can be accessed over the method `getURIFragment(element)` of the EMF resource. When a new element is created, the index is stored in a bidirectional hash map in the `RuleModelIndex` that is accesible over the `ModelState`. This index can also be used for the notation model, where the semantic element id needs to be stored to be able to map the layout information back to the source model element. One problem of storing the Henshin identifiers is that a transformation rule is stored as a LHS and RHS part. Each part hs its own identifier, even though it is only one element in the graph. For that Henshin also stores mappings of the LHS and RHS elements in the *.henshin* file. To be able to correctly map the source model elements to the graphical model elements, these mappings are also stored in the `RuleModelIndex`. In listing 1 you can see the implementation of the methods `getRuleElement(id)` and `getRuleElementId(element)` that are used to get the element from the index or get the index of an element. You can see that before searching the index, the mapping list is checked to ensure that the LHS element is preferably returned, if it exists. That is for example needed for setting the source and target nodes of an edge. If the edge only appears in the RHS part and it should get deleted when applying the rule the `getSource()` method returns the RHS node element, but the source node was initially created from the LHS element. Without the mapping, the source node would not be found in the index and therefore creating an new index, that results in a invalid route in the graphical model.

```
1 public void indexRuleElement(String id, GraphElement element) {
2     if(ruleElementIndex.containsKey(id))
3         return;
4     ruleElementIndex.put(id, element);
5 }
6
```

```
public GraphElement getRuleElement(String id) {
    if (rhsToLhs.containsKey(id)) {
        String lhsId = rhsToLhs.inverseMap().get(id);
        if (ruleElementIndex.containsKey(lhsId)) {
            return ruleElementIndex.get(lhsId);
        }
    }
    return ruleElementIndex.get(id);
}

public String getRuleElementId(GraphElement element) {
    String id = element.eResource().getURIFragment(element);
    if(rhsToLhs.inverseMap().containsKey(id)){
        return rhsToLhs.inverseMap().get(id);
    }
    if(lhrToRhs.inverseMap().containsKey(id)){
        return lhrToRhs.inverseMap().get(id);
    }

    return ruleElementIndex.inverse().get(element);
}
```

Listing 1. Parts of `RuleModelIndex`

This problem doesn't appear for the Ecore metamodel indexing because no content independent indexes are stored in the EMF model. Here the indexing is used from the existing GLSP Ecore editor [**glsp-ecore-repo**]. The `EcoreModelIndex` stores an index for the semantic elements, the notation elements and an additional index for inheritance edges. For the semantic index, random Universally Unique Identifiers (UUIDs) are created. They are used until the client session is closed. During this time, operations on the source model can access EMF elements by their UUIDs over the stored `HashMap` and then apply the operation on the EMF element. The indentifiers are content-independent, which has the advantage, that the identifiers are not changing when nodes are updated. The problem with temporary identifiers on the other hand is, that they cannot be mapped to the source elements after the client session is closed. Therefore, the UUIDs cannot be used in the notation model, because the same notation model needs to be loaded across client sessions. Here, the name of the EMF class is used, since it is unique for each element in the Ecore metamodel. This index has to be updated if a class is renamed. The inheritance index for the Ecore metamodel is used to find already created inheritance edges and retrieve their bend points. With that information, the edges can be connected at bend points to create the typical inheritance arrow structure.

For the notation models of XMI instance models, also content hashes are used as identifiers. Here the name of a object is not unique, because multiple objects of one class can exist. Therefore the content hash, is created from the class name and the names and values of all its attributes. A hash for the class *Client* can look like this: *Client:DynamicEObjectImpl-name:EString=Alice* This content hashes is generated every time the graphical model is created for the first time in a session. It needs to be updated when a attribute value is changed. Content hashes for edges would be even more complex, because they need to include the source and target node hashes combined with the edge type. This is also a reason, why the edge layout information is not stored in the notation model, since the hashes need to be changed for many edit operations to the source model. For XMI instance elements, the additional use of adapters are used. The `NotationAdapter` and the `UUIDAdapter` store the index in the Adapter, which is then directly attaced to the EMF element. This

has the advantage, especially for the `NotationAdapter`, that when the content hash has to be updated, the notation model can be directly fetched from the EMF element. It also contains the hashing algorithm for nodes. You can see the implementation of the `NotationAdapter` in listing 2. These content hashes need to be used for session independent identifiers, but using them as the only identifier would need a lot of overhead to update the hashes. Therefore, the indexing of the semantic elements works like the metamodel indexing, where UUIDs are used. The combination of the UUIDs and content hashes allows flexibility for editing the source model, while maintaining the connection to the notation model.

5) Custom UI extensions: This section demonstrates the creation of custom UI extensions by two different examples. GLSP provides a predefined interface for creating custom UI elements, that could be used in all platoform integrations. For that the abstract class `AbstractUIExtension` must be extended and added to the *henshin-glsp* application module. One simple example is the transformation rule name with its parameters that is displayed in the top left of the rule editor. The extension needs a defined id and a parent container id. With the `SetUIExtensionVisibilityAction`, the UI element can be made visible from external over the id. With the method `initializeContents(containerElement)`, the HTML elements can be created and added to the container. After the model initialization and over a public updated method, the class requests the rule name and its parameters over the `IActionDispatcher` and updates the UI. This update method can be called from any other class, when the `RuleNameUIExtension` is registered and injected over the dependency injection. One example is the explorer view, where the rule can be opened and therefore the rule name must be updated.

This custom explorer is a Theia exclusive extension, accessing the Theia internal APIs. It cannot be used for other GLSP platform integrations. To use a custom theia explorer was already discussed in section V-A. To implement a custom explorer, the classes `FileNavigatorModel`, `FileNavigatorTree`, and `FileNavigatorWidget` are extended and registered in the theia specific *rules-theia* module via dependency injection. To add additional virtual elements in the explorer tree, the two new tree nodes `HenshinRootNode`, that contains a list of children, and `HenshinRuleNode`, that contains information like the rule name, are created. The method `resolveChildren(parent)` is overwritten in the `FileNavigatorTree`. Here, if it iterates over a *.hensin* file node, it requests the transformation rules from the server and creates the corresponding `HenshinRuleNode` for each rule. It creates also an additional node that works as a „add rule“ button. In the `HenshinNavigatorWidget`, the method `onSelectionChanged` event is subscribed. It checks if a virtual `HenshinRuleNode` was selected. If that is the case, it tries to find the GLSP rule wdiget and opens it. It also sends the selected rule name to the server, that is then selecting the rule in the `RuleGModelFactory`, where the graphical model is created. To provide a fitting look to the new tree nodes, the `HenshinNavigatorWidget` implements the methods `toNodeName(node)` and `toNodeIcon(node)`. Here, fitting icons are selected and the displayed names are configured.

VIII. Testing and Evaluation

A. Testing Strategy

For the Java backend, unit tests wer implemented using JUnit. For every milestone of the development process, unit tests were added to ensure the added functionality works as expected. The tests cover the core functionality of the backend. Mocking was used to simulate the behaviour of some components, like the `ModelState`.

To test the UI, automated UI tests were created using Playwright.

XI. Acronyms

GLSP	Graphical Language Server Platform
EMF	Eclipse Modeling Framework
MDE	Model-Driven Engineering
UI	User Interface
GUI	Graphical User Interface
IDE	Integrated Development Environment
SDV	Software-Defined Vehicle
JDT	Java Development Tools
PDE	Plug-in Development Environment
SDK	Software Development Kit
API	Application Programming Interface
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
LHS	Left-Hand Side
RHS	Right-Hand Side
NAC	Negative Application Condition
PAC	Positive Application Condition
RPC	Remote Procedure Call
DI	Dependency Injection
HTML	Hypertext Markup Language
SVG	Scalable Vector Graphics
URI	Uniform Resource Identifier
JDK	Java Development Kit
JAR	Java Archive
ELK	Eclipse Layout Kernel
POC	Proof of Concept
UUID	Universally Unique Identifier
AToMPM	A Tool for Multi-Paradigm Modeling
DSML	Domain Specific Modeling Language
VS Code	Visual Studio Code
CSS	Cascading Style Sheets

XII. Appendix

A. Figures

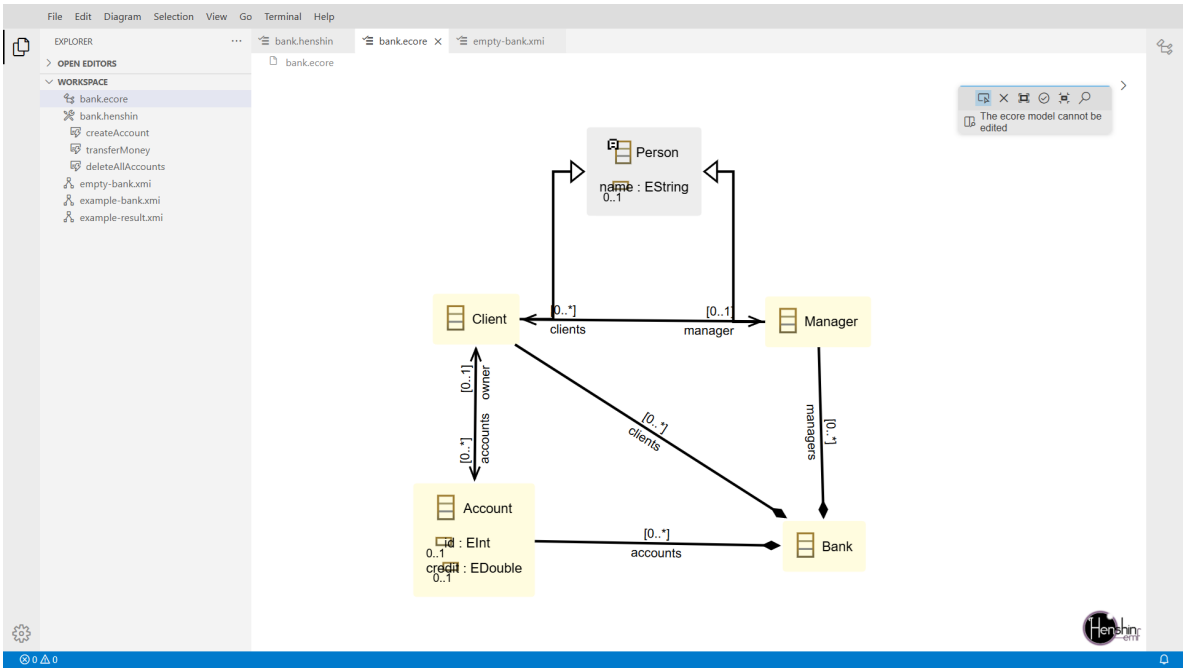

Figure 9. Henshin Web Ecore graph editor

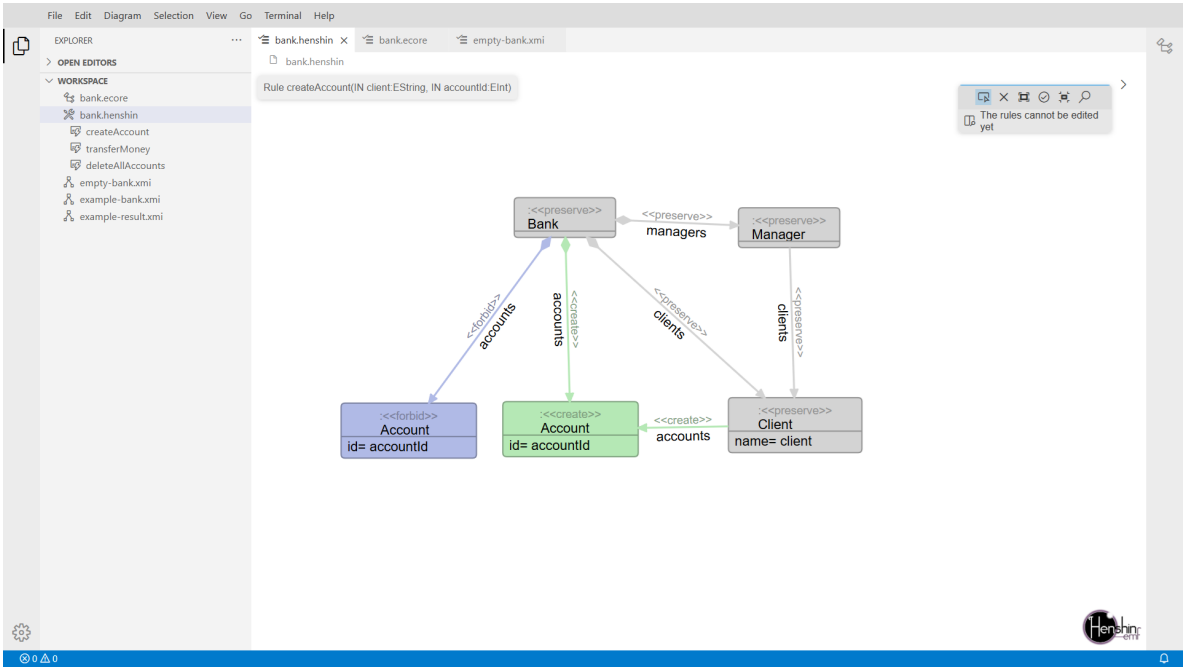

Figure 10. Henshin Web Rules graph editor

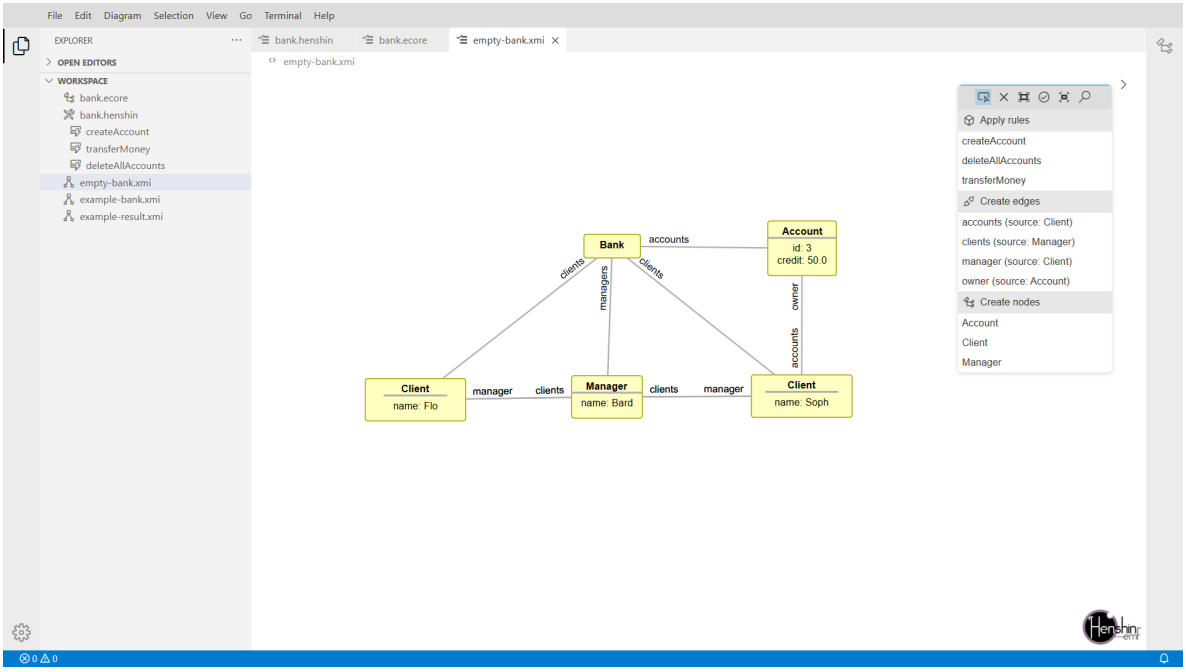

Figure 11. Henshin Web Instance graph editor

B. *Code Listings*

```java
public class NotationAdapter extends AdapterImpl {
    private Shape shape;

    public NotationAdapter(Shape shape) {
        this.shape = shape;
    }

    @Override
    public boolean isAdapterForType(Object type) {
        return type == NotationAdapter.class;
    }

    public static Shape getOrAssignNotation(DynamicEObjectImpl obj) {
        // Return existing Notation if present
        for (var adapter : obj.eAdapters()) {
            if (adapter instanceof NotationAdapter) {
                return ((NotationAdapter) adapter).getShape();
            }
        }

        // Assign new Notation
        String hashId = hashENodeObject(obj);
        Shape shape = notationMap.get(hashId);
        if(shape == null) {
            shape = XMINotationFactory.createNewShape(obj);
        }
        NotationAdapter newAdapter = new NotationAdapter(shape);
        obj.eAdapters().add(newAdapter);

        return shape;
    }

    public static String hashENodeObject(DynamicEObjectImpl eObject) {
        StringBuilder result = new StringBuilder();

        result.append(eObject.eClass().getName()).append(":");
        result.append(DynamicEObjectImpl.class.getSimpleName());

        for (EStructuralFeature feature : eObject.eClass().getEAllStructuralFeatures()) {
            if (feature instanceof EAttribute) {
                result.append("-").append(feature.getName());
                result.append(":").append(feature.getEType().getName());
                result.append("=").append(eObject.eGet(feature).toString());
            }
        }

        return result.toString();
    }

    public static void dispose() {
        notationMap.clear();
    }
}
```

Listing 2. Parts of `NotationAdapter`

```
@Override
protected void fillRootElement(GModelRoot newRoot) {
    EGraph instanceNodes = modelState.getInstanceGraph();

    newRoot.getChildren().addAll(instanceNodes.stream()
            .map(eObject -> (DynamicEObjectImpl) eObject) //
            .map(this::createNode)//
            .toList());

    ...
}

public GNode createNode(DynamicEObjectImpl eObject) {
    String type = DefaultTypes.NODE;
    if(eObject.eContainer() == null) {
        type = HenshinTypes.XMI_ROOT_NODE;
    }

    GNodeBuilder b = new GNodeBuilder(type) //
            .id(UUIDAdapter.getOrAssignId(eObject)) //
            .layout(GConstants.Layout.VBOX) //
            .addCssClass(HenshinCss.XMI_NODE) //
            .addArguments(GArguments.cornerRadius(3))
            .add(buildHeader(eObject));
    if(!eObject.eClass().getEAllAttributes().isEmpty()) {
        b.add(createAttributesCompartment(eObject.eClass().getEAllAttributes(), eObject));
    }
    applyShapeData(eObject,b);
    return b.build();
}

private GLabel buildHeader(EObject eObject) {
    return new GLabelBuilder(DefaultTypes.LABEL) //
            .id(UUIDAdapter.toLabelId(eObject))
            .addCssClass(HenshinCss.XMI_LABEL)
            .text(eObject.eClass().getName()) //
            .build();
}

...
```

Listing 3. Parts of `XMIGModelFactory`