Marburg University

Department of Mathematics and Computer Science

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. [27] 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. [27] 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. [29] One model transformation framework is Henshin. [31] It tries to provide model transformation capabilities with a high level of usability. [30] 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. [31] 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. [19] 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. [5] 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.

B. Problem Statement

C. Research Questions

D. Scope and Limitations

E. Structure of the Thesis

II. Background

In this section, the theoretical background of the project and used technologies are described.

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. [15] 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. [17, 15] 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. [3] 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. [3]

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. [29] 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. [7] 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. [29]

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.“ [9]

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. [6] 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. [29]

- **EAttribute**: An attribute of a class, that are identified by a name and have a type. [29]
- **EDataType**: A simple data type like `EString`, `EBoolean` or `EJavaClass`. [29]
- **EReference**: A reference to another class, containing a link to an instance of that class. [29]

Together, Steinberg et al. 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. [29]

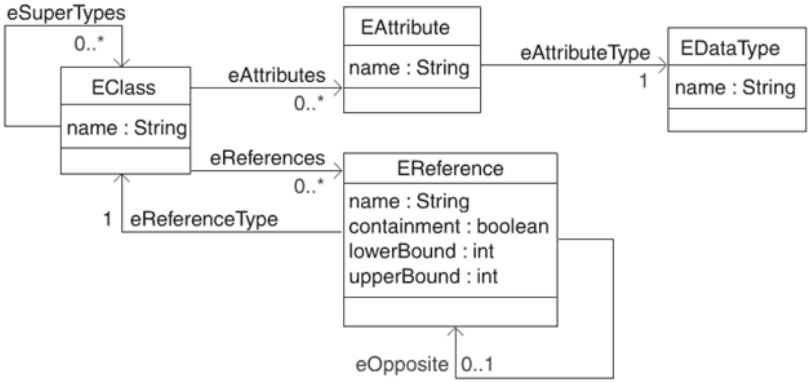

Figure 1. The Ecore kernel. Image obtained from [29]

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!**. [6] 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`. [29]

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!**. [6] The generation can be done over a wizard in the Eclipse IDE or by using the command line interface. [29] 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. [6] 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 **.!** 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. [31]

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. [31] 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. [1] 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. [1] 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. [1]

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. [31] 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. [31] 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.

D. Graphical Language Server Platform (GLSP)

GLSP is a framework that provides components for the development of GUIs for web-based diagram editors. [19] It is organized within the Eclipse Cloud Development project. [5] With the framework, custom diagram

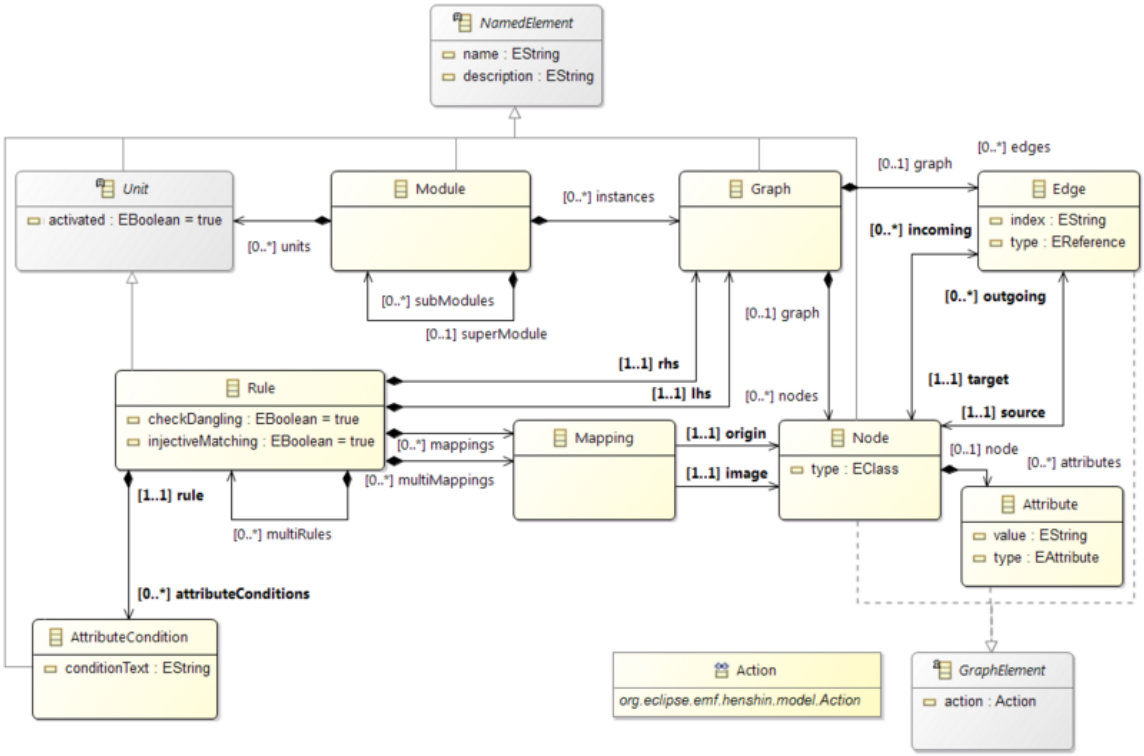

Figure 2. Henshin transformation rule metamodel. Image obtained from [31]

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 [23].

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 [13] 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.

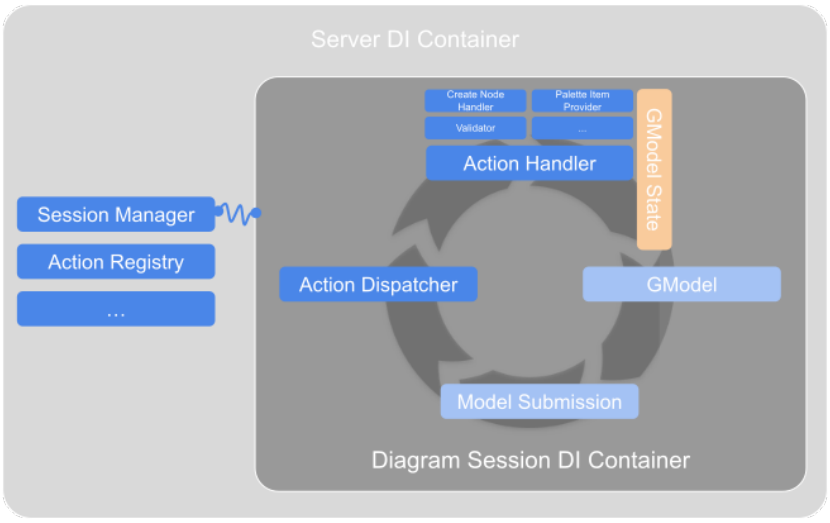

Figure 3. Server DI Container vs Diagram Session DI Container. Image obtained from [5]

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. [5]

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. [5] GLSP provides four main **ui!** components to apply commands or edit the graph but also allows custom **ui!** (**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. [5, 19]
- **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. [5, 19]

- **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. [5, 19]
- **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. [5, 19]
- **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. [5, 19]

GLSP uses Sprotty [11], 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 [14] and then sent over JSON-RPC. [5]

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. [5] 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. [19] GLSP also provides a `LayoutEngine` interface, that can be used to layout the elements of a graph that have no persisted layout yet. [5]

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. Kahani et al. created a survey in 2019 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. [16]

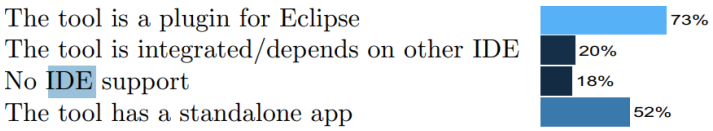

Figure 4. Execution environments of model transformation tools. Image obtained from [16]

One web-based tool included in the survey is A Tool for Multi-Paradigm Modeling (AToMPM) [33]. It is a web-based modeling tool to create Domain Specific Modeling Language (DSML) environments, performing model transformations and manipulating and managing models. [33] It was created in 2013 and supports all model transformations that are based on T-Core [32], a minimal common basis that allows interoperability between different model transformation languages. [32] 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. [33]

There are also other web-based tools for MDE. WebGME [21] is a web-based modeling tool, created in 2014. 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. [21] The software provides extension points to customize or extend the software, but no model transformation capabilities were added by any available extension. [35] The tool is still hosted and maintained, to be used for free. [35]

WebDPF [26] is another web-based modeling tool, published in 2016. 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. [26] 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. [10] 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 [19].
- 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. 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. [5] 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 5 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 [5]. 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-C for more details).

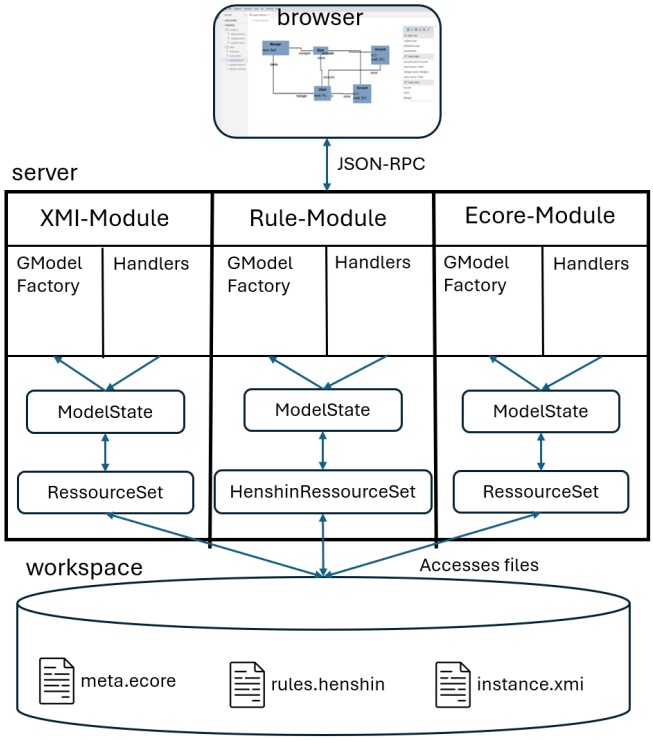

Figure 5. High-Level Architecture of the System

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.

B. 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 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.

C. Data Flow and Control Flow

D. Component Design

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 final UI can be seen in appendix 8, 9 and 10.

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. [5] 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. [20] 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. [20] 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. [4] 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. [24] The GLSP VS Code integration can provide a *.vsix* file. [19] VS Code is the most used IDE. 73.6% of developers use VS Code due to the survey of Stack Overflow In 2024 [28]. An advantage to Eclipse is that VS Code provides a browser version, which brings the same capabilites as the desktop IDE. [24] 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 [28], 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. [8] 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 6.

- **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. [8]

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. [19] 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.

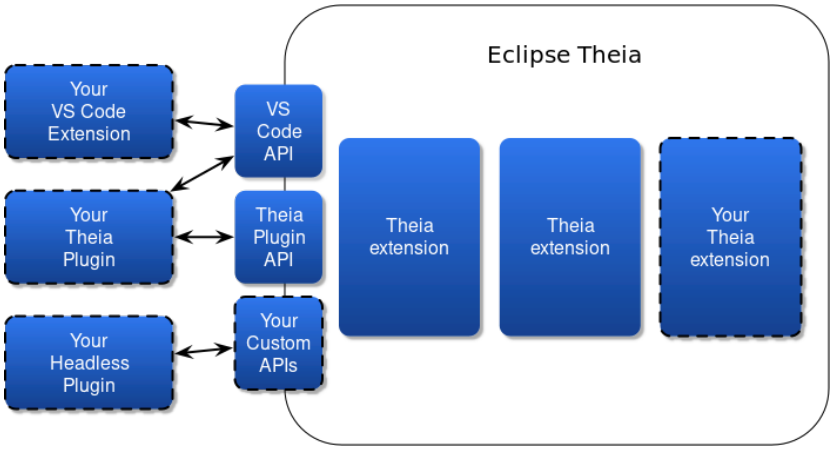

Figure 6. Theia high level extensions and plugins architecture. Image obtained from [8]

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 [34] to deploy Theia based products on Kubernetes clusters [18]. 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 7. The service provides two preconfigured configurations for quickly trying out Theia Cloud on a cluster. [34]

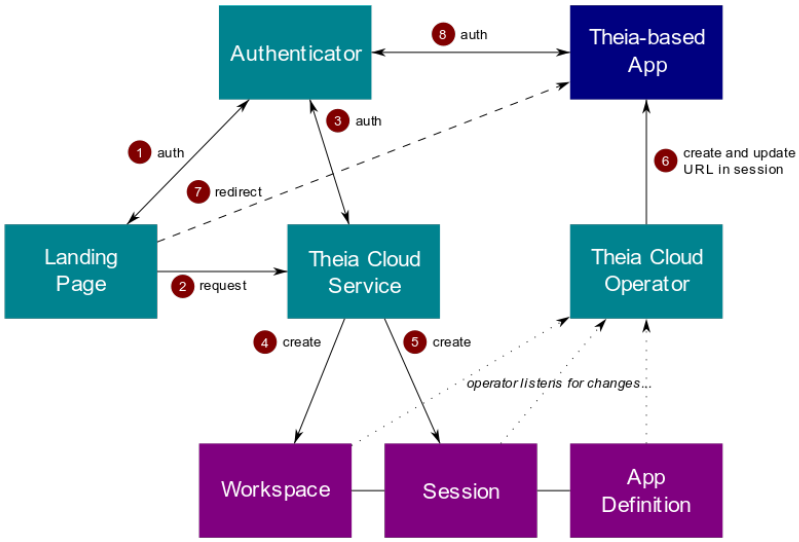

Figure 7. Interaction between Theia Cloud components. Image obtained from [34]

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. [22] 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 [12] 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 I 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 I
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 and shows the solution and implementation of specific problems, that appeared while implementing a Proof of Concept (POC) to display XMI instances and apply model transformation rules on them. The main problems with the POC were the integration of Henshin into a GLSP project, the layout of the graphical model, and the indexing of EMF model elements. The GUI of the POC can be seen in figure **??**.

A. 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 is structured as an Eclipse project and is available as a set of Eclipse plugins and features. [31] On the other hand, GLSP projects typically use a Maven project structure. [19] 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. To provide the Henshin dependencies to anyone, the packages are stored in the Giblab package registry of the Henshin Web GitLab repository. 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 [25] is needed. The Nashorn engine is used to execute calculation expressions of transformation rules. [1]

B. 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. [29, 31] To provide a good user experience, the graphical editors need to provide a consistent macro layout of 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. [5] 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. [2] 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. [19] 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` when 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. For the XMI instance models, layouting over the notation models is implemented in the POC. The *.notation* file is created when the instance model is loaded for the first time. Here, ELK can be used to create a fitting initial layout. When creating the graphical model 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 information. [29] 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.

C. Indexing EMF models

Next to layout information, EMF Ecore metamodels and EMF XMI instance models don't by default contain unique identifiers for nodes, edges, or attributes. [29, 9] During the transformation of the source model into the graphical model, elements need to be accessed multiple times. For example, a node is accessed from the EMF package when it is mapped into a `GNode` and then again for all its connected edges and attributes. To avoid multiple lookups in the EMF model, an index is created for each node, edge, or attribute. Additionally, the graphical model of GLSP needs identifiers for each element, to be able to specify the element for operations on the source model. When no identifiers are specified, GLSP generates its own unique identifiers for each element, but they cannot be mapped back to the corresponding EMF element. To keep a unique identifier close to the EMF element, an EMF `Adapter` is used.

For adding indexes to the graphical model elements, random Universally Unique Identifiers (UUIDs) can be used. The same identifiers can be held for the lifetime of the corresponding client session. During this time, operations on the source model can access EMF elements by their UUIDs over a `HashMap` and then apply the operation. The use of content-independent identifiers 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. One use case for that are the notation models, where the UUIDs cannot be used, because the same notation model needs to be loaded across client sessions. To achieve a session-independent solution to connect the notation model's elements to the EMF elements, content hashes are used as identifiers. Each node in the source model is identifiable by the class name and its combined attribute values. These content hashes are stored in the notation model and generated every time the graphical model is created for the first time in a session. The combination of the UUIDs and content hashes allows flexibility for editing the source model, while maintaining the connection to the notation model.

D. Development Process

E. Key Features and Functionality

F. Tooling and Environment

G. Code Examples...

VIII. Testing and Evaluation

A. Testing Strategy

B. Test Results and Coverage

C. Performance Evaluation

D. User Feedback

E. Comparison with Requirements

IX. Discussion

A. Interpretation of Results

B. Challenges and Limitations

X. Conclusion and Future Work

A. Summary of Contributions

B. Suggestions for Future Development

XI. Acronyms

GLSP	Graphical Language Server Platform
EMF	Eclipse Modeling Framework
MDE	Model-Driven Engineering
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

XII. Appendix

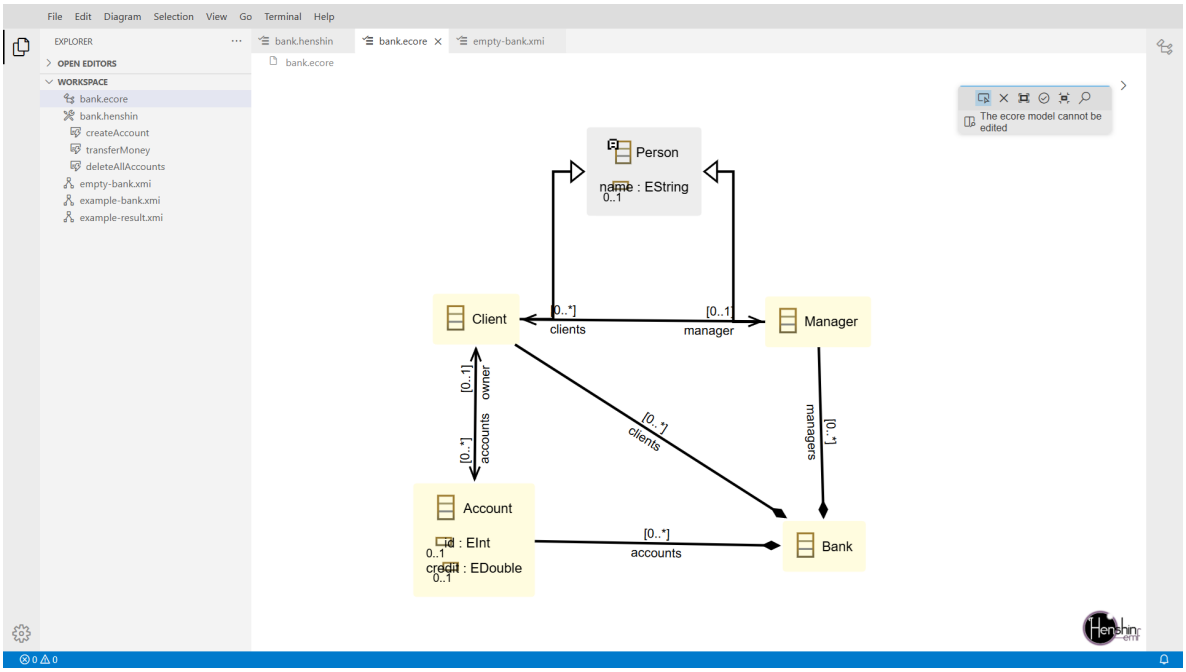

Figure 8. Henshin Web Ecore graph editor

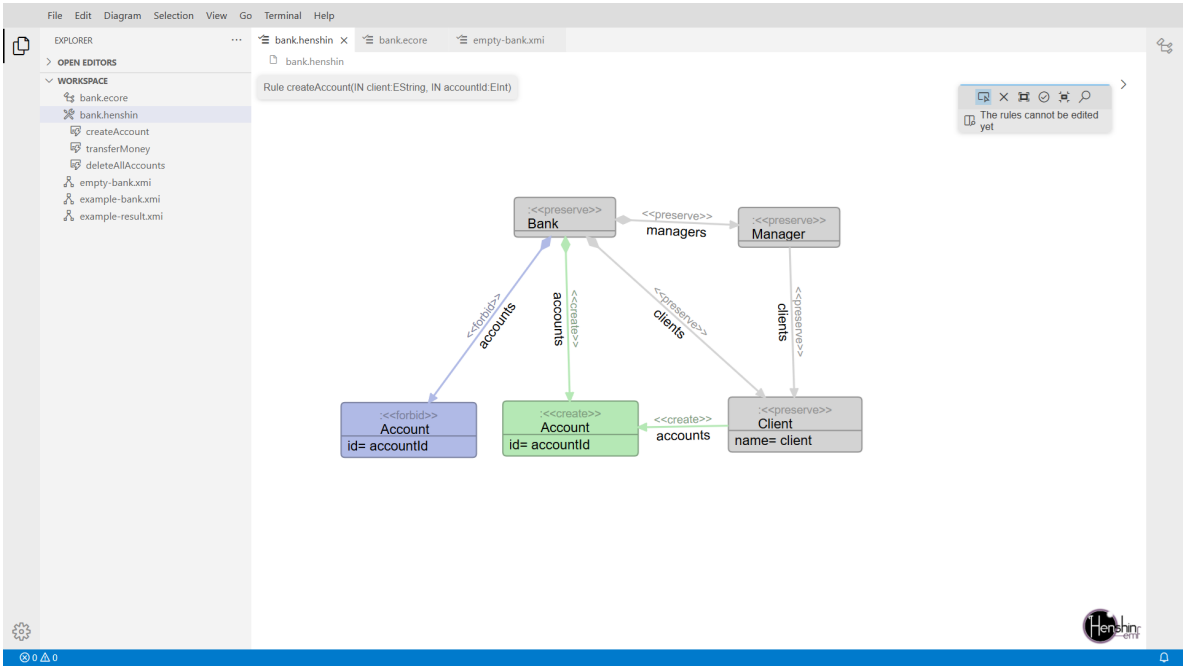

Figure 9. Henshin Web Rules graph editor

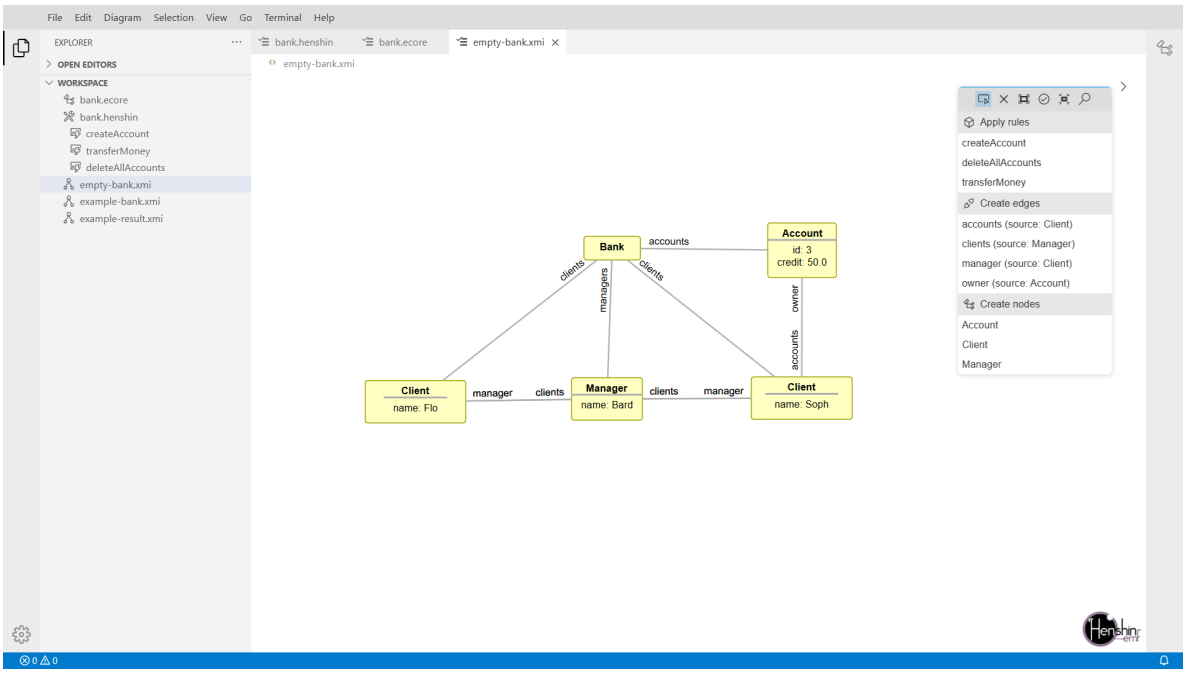

Figure 10. Henshin Web Instance graph editor

REFERENCES

[1] Thorsten Arendt et al. "Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations". In: *Model Driven Engineering Languages and Systems*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 121–135. ISBN: 978-3-642-16145-2.

[2] Sören Domrös et al. *The Eclipse Layout Kernel*. 2023. arXiv: 2311.00533 [cs.DS]. URL: https://arxiv.org/abs/2311.00533.

[3] Eclipse Foundation. *2024 Annual Community Report*. en. https://www.eclipse.org/org/foundation/reports/annual_report.php. Accessed: 2025-6-2. 2024.

[4] Eclipse Foundation. *Eclipse Documentation - Current Release*. Accessed: 2025-06-28. 2025. URL: https://help.eclipse.org/latest/index.jsp.

[5] Eclipse Foundation. *Eclipse GLSP™ – Documentation*. Accessed: 2025-05-20. 2025. URL: https://eclipse.dev/glsp/documentation.

[6] Eclipse Foundation. *Eclipse Modeling Framework*. Accessed: 2025-06-02. 2025. URL: https://eclipse.dev/emf/.

[7] Eclipse Foundation. *Eclipse Modeling Project*. Accessed: 2025-06-02. 2025. URL: https://eclipse.dev/modeling/.

[8] *Eclipse Theia Documentation*. Accessed: 2025-06-28. Eclipse Foundation, 2025. URL: \url{https://theia-ide.org/docs/}.

[9] Eclipse Foundation. *Eclipse Modeling Framework*. https://github.com/eclipse-emf/org.eclipse.emf. 2025.

[10] Eclipse Foundation. *ecore-glsp*. https://github.com/eclipsesource/ecore-glsp. 2025.

[11] Eclipse Foundation. *Sprotty*. https://github.com/eclipse-sprotty/sprotty. 2025.

[12] OpenJS Foundation and Electron contributors. *Electron*. https://github.com/electron/electron. 2025.

[13] Google. *Google Guice*. https://github.com/google/guice. 2025.

[14] Google. *Gson*. https://github.com/google/gson. 2025.

[15] Michael Jastram. *Die Eclipse Foundation wird 20 Jahre alt: Die interessantesten SE-Projekte*. de. https://www.se-trends.de/eclipse-foundation-wird-20-jahre-alt/. Accessed: 2025-6-2. Dec. 2024.

[16] Nafiseh Kahani et al. "Survey and classification of model transformation tools". In: *Software & Systems Modeling* 18 (2019), pp. 2361–2397.

[17] Alexandra Kleijn. *Eclipse und die Eclipse Foundation*. Accessed: 2025-6-2. 2006. URL: https://www.heise.de/-221997.

[18] *Kubernetes Documentation*. Accessed: 2025-06-28. Linux Foundation, 2025. URL: \url{https://kubernetes.io/docs/home/}.

[19] Philip Langer and Tobias Ortmayr. *Eclipse GLSP*. https://github.com/eclipse-glsp/glsp. 2025.

[20] Philip Langer and Tobias Ortmayr. *Eclipse GLSP - Client*. https://github.com/eclipse-glsp/glsp-client. 2025.

[21] Miklós Maróti et al. *Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure*. White paper / Technical report. White paper available at https://webgme.org/WebGMEWhitePaper.pdf. 2014.

[22] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux journal* 2014.239 (2014), p. 2.

[23] Microsoft. *Language Server Protocol*. https://github.com/microsoft/language-server-protocol. 2025.

[24] Microsoft Corporation. *Visual Studio Code Documentation*. Accessed: 2025-06-28. 2025. URL: https://code.visualstudio.com/docs.

[25] OpenJDK. *Nashorn Engine*. https://github.com/openjdk/nashorn. 2025.

[26] Fazle Rabbi et al. "WebDPF: A web-based metamodelling and model transformation environment". In: *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 2016, pp. 87–98.

[27] S. Sendall and W. Kozaczynski. "Model transformation: the heart and soul of model-driven software development". In: *IEEE Software* 20.5 (2003), pp. 42–45. DOI: 10.1109/MS.2003.1231150.

[28] Stack Overflow. *Stack Overflow Developer Survey 2024: Technology*. Accessed: 2025-06-28. 2024. URL: \url{https://survey.stackoverflow.co/2024/technology#most-popular-technologies-new-collab-tools}.

[29] David Steinberg et al. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009. ISBN: 0321331885.

[30] Daniel Strüber et al. "Henshin: A Usability-Focused Framework for EMF Model Transformation Development". In: *Graph Transformation*. Ed. by Juan de Lara and Detlef Plump. Cham: Springer International Publishing, 2017, pp. 196–208. ISBN: 978-3-319-61470-0.

[31] Daniel Strueber. *Henshin*. https://github.com/eclipse-henshin/henshin/wiki. 2025.

[32] Eugene Syriani, Hans Vangheluwe, and Brian Lashomb. “T-Core: a framework for custom-built model transformation engines”. In: *Softw. Syst. Model.* 14.3 (July 2015), 1215–1243. ISSN: 1619-1366. DOI: 10.1007/s10270-013-0370-4. URL: https://doi.org/10.1007/s10270-013-0370-4.

[33] Eugene Syriani et al. “AToMPM: A web-based modeling environment”. In: *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): September 29-October 4, 2013, Miami, USA*. 2013, pp. 21–25.

[34] *Theia Cloud*. Accessed: 2025-06-28. Eclipse Foundation, 2025. URL: \url{https://theia-cloud.io/documentation/}.

[35] *WebGME: Web-based Generic Modeling Environment*. Accessed: 2025-06-18. 2025. URL: https://webgme.org/.