Scientific Work: Creating a web-based model transformation diagram editor

Florian Weidner
Philipps-University Marburg, Germany
Department of Mathematics and Computer Science, Software engineering group
May 06, 2025

I. Motivation and introduction

In software engineering, often Model-Driven Engineering (MDE) is used to increase development productivity and quality. 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 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. [21] One framework to use MDE is Eclipse Modeling Framework (EMF) by the Eclipse Foundation. It provides a basis for application development, using modeling and code generation facilities. Much frameworks build upon EMF, providing various MDE tools like code generators, graphical diagramming, model transformation or model validation. [22] One model transformation framework is Henshin. [24] It tries to provides model transformation capabilites with a high level of usability. [23] 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. [24] 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.

Graphical Language Server Platform (GLSP) is a open-source framework by the Eclipse Foundation to develop custom diagram editors for distributed web-applications. [16] It can be used in Eclipse Desktop IDE, Eclipse Theia, Visual Studio Code and embedded in any website. With these fuctionalities, GLSP fits to create an accessible, intuitive application to create and apply Henshin model transformations.

The goal of this scientific work is to provide relevant information about the used technologies and to research how they fit together to create a web-based model transformation editor.

II. Theoretical backend

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. [13] 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 many different projects were initiated by the organization in the Eclipse environment, all running under the Eclipse Public License. [15, 13] In the recent years, the key initatives of the Eclipse Foundation are contributing to european digital souvereignty, 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, a 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 organisations 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. [22] 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 application on top. [6] 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 capabilites for model transformation, database integration, or graphical editor generation. [22]

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.“ [7]

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 meta models, a runtime support for the models, and a basic API for manipulating EMF objects generically. Ecore meta models are used to describe the structure of a model. [5] 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.[22]
- **EAttribute**: An attribute of a class, that are identified by a name and have a type.[22]
- **EDataType**: A simple data type like `EString`, `EBoolean` or `EJavaClass`[22]
- **EReference**: A reference to another class, containing a link to an instance of that class.[22]

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. [22]

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 meta models can be displayed in JFace viewers, that are

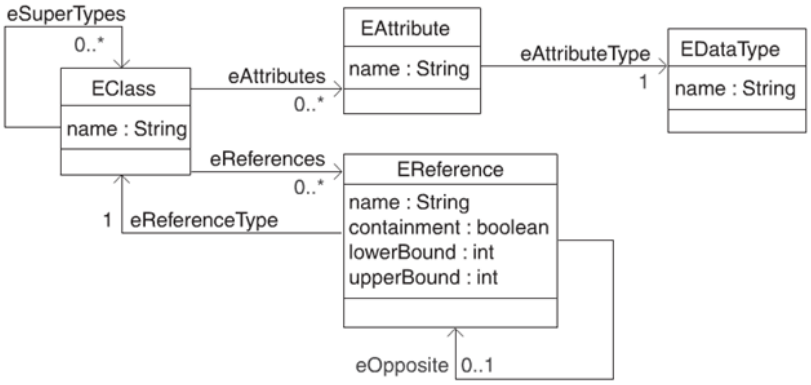

Figure 1. The Ecore kernel. Image obtained from [22]

part of the Eclipse UI. [5] The Eclipse IDE can display a Ecore model in a tree viewer. Eclipse acceses 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 a instance model. For that it uses a command framework with commands like a `AddCommand`, `SetCommand` or `CopyCommand`. [22]

The third component is EMF.Codegen. It can generate Java code for a complete editor for EMF instance models of a Ecore meta model. 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. [5] The generation can be done over a wizard in the Eclipse IDE or by using the command line interface. [22] The generation can be sepertated in to 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 meta model.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. [5] There are many frameworks that build on top of EMF using these generation capabilites to create further modeling functionality. One of these frameworks is Henshin, that is described in section 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 on 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 a interpreter with debugging support and a performance profiler. Henshin also provides confilct and dependency analysis, and state space analysis for verification. [24]

Henshin builds on top of EMF. It uses an Ecore meta model 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. [24] The meta model of the transformation rules uses another Ecore meta model 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 meta model. A rule consits 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. Henshins 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 multible Negative Application Conditions (NACs), Positive Application Conditions (PACs) and Paramters can be specified directly. [24] 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. [24] 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 paramters, a new rule can be described. If you want to define a node, you can use the keyword **node** with a its action keyword like **create** or **preserve** to specify the action of the node in the transformation.

The Henshin SDK consists of multible 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. [16] It is organized within the Eclipse Cloud Development project. [4] 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 focus 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 [18].

The GLSP server is responsible for loading a source model and defines how to transform it into the graphical model, that should be displayd. 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 [10] for Dependency Injection (DI). The GLSP server distinguish 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 informations, 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

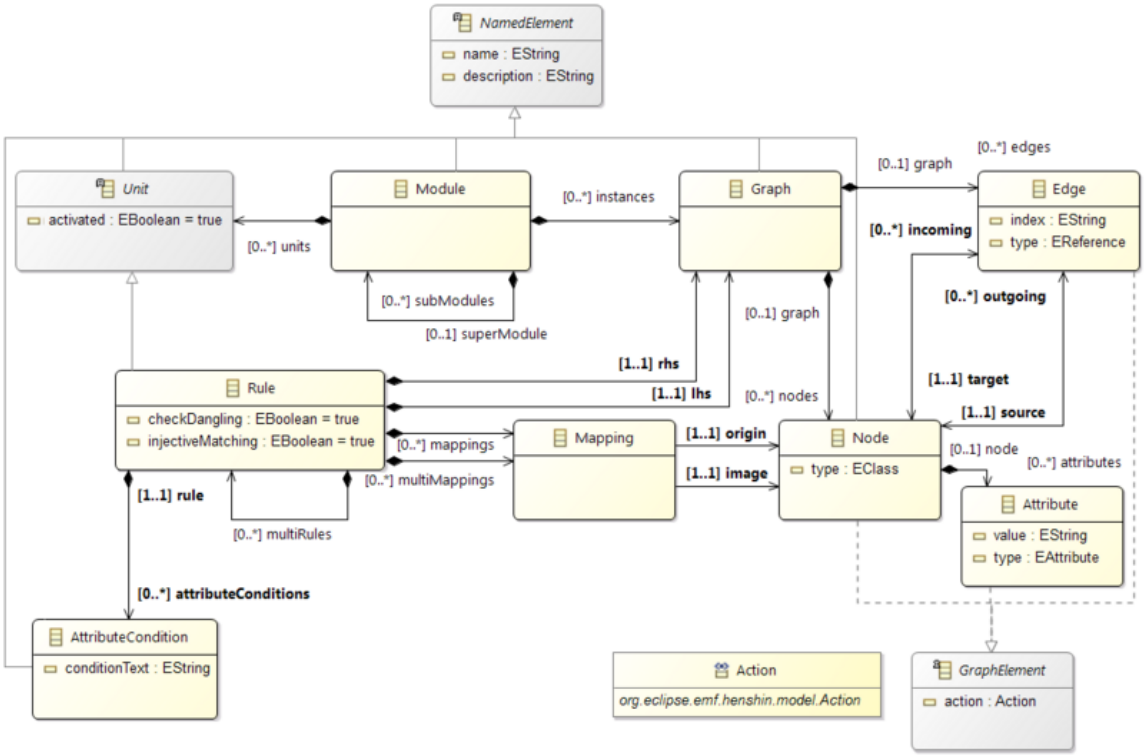

Figure 2. Henshin transformation rule meta model. Image obtained from [24]

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 registerd 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 `RessouceSet`. The interface `GModelFactory` is used to map the source model to the GLSP internal graphical model structure. Here also a abstract `EMFGModelFactory` is provided. Another important part is the `GModelState` interface, that defines the state of a client sesseion and holds all information about the current state of the original source model. All services and handlers use the `GModelState` to obtain required informations for their tasks.

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, wich 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.[4]

The GLSP client is responsible for rendering the graph and managing user interactions. The client requests

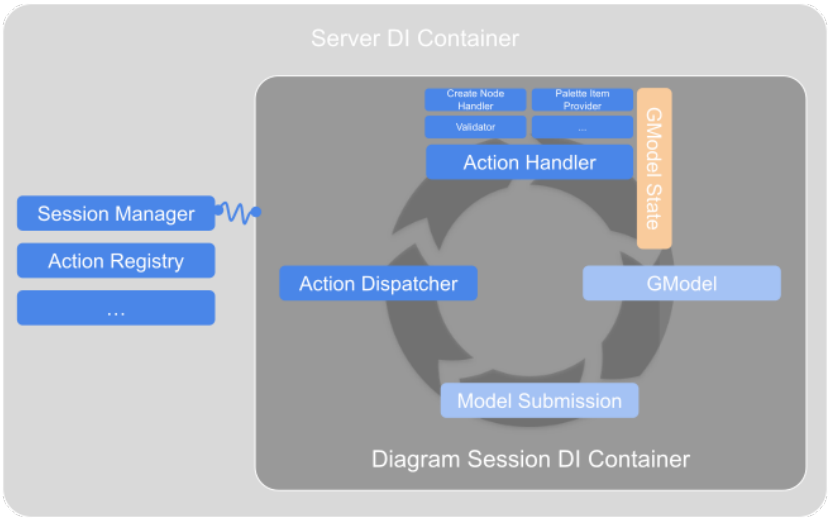

Figure 3. Server DI Container vs Diagram Session DI Container . Image obtained from [4]

all possible editing operations that can be performed on the specific model. As the client for this project is integrated in Eclipse Theia, the following discussion focus exclusively on the Theia integration of the GLSP client. [4] 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 a expandable UI element located on the top left of the diagrm 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. [4, 16]
- **CommandPallete**: 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. [4, 16]
- **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 strucutred 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. [4, 16]
- **EditLabelUI**: Labels of nodes and edges can be edited by double clicking on the label. The EditLabelUI provides a input popup to edit the label text. [4, 16]
- **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. [4, 16]

GLSP uses Sprotty [9], an 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 a edge between two nodes or ports. Four classes inherit from `GShapeElement`, that representes an element with a certain shape, position and size. They can also be nested inside another `GShapeElement`. The `GNode` can have `GLabel` or `GPort`, that 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 capabilites of EMF. To send the graphical model to the client, it is serialized into JSON using GSON [11] and then sent over JSON-RPC. [4]

The layout of a graph is divided into macro and micro layouting. The macro layouting, aranging 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. [4] For the macro layouting, GLSP provides a notation model, that persists the position and size of the elements in a seperate notation XMI file. The Notation Diagram can be added to the `GModelState` and then used in the `GModelFactory` to specify the layout. [16] GLSP also provides a `LayoutEngine` interface, that can be used to layout the elements of a graph that have no persisted layout yet. [4]

GLSP also provides a 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 a 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/Software

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, also 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/dependent on other IDEs. 18% have no IDE support, and only two tools are web-based. Also 89% of the tools have external dependencies like an IDE or other tools. Dependencies often complicate the installation and usage of the tool. [14]

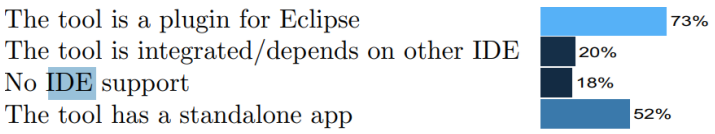

Figure 4. Execution environments of model transformation tools. Image obtained from [14]

One web-based tool included in the survey is A Tool for Multi-Paradigm Modelling (AToMPM) [25].

There are also already web-based tools for model transformations. MDEForge [12]. WebGME [17] Web-DPF[20] There is also a GLSP-based ecore 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 major updates in the GLSP framework, it provides various classes that can be used in the Henshin Web Ecore viewer. One example is the Factory code that mapps the EMF Ecore model to the graphical model. [8]

IV. Implementation

This chapter describes and shows the solution and implementation of specific problems, that appeared while implementing a Proof of Concept (POC) to disply XMI instances and apply model transformation rules on them. The main problems with the POC was the integration of Henshin into a GLSP project, the layouting of the graphical model and the indexing of EMF model elements. The GUI of the POC can be seen in figure 5.

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. [24] On the other hand, GLSP projects typically use a Maven project structure. [16] 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 a 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 from 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 parts of the Henshin SDK are needed. To use the Henshin model package, the additional dependency of the Nashorn JavaScript engine [19] is needed. The Nashorn engine is used to execute calculation expressions of transformation rules. [1]

B. Layouting

EMF Ecore meta 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. [22, 24] To provide a good user experience, the graphical editors need to provide a consistent marco 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. [4] 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 soucre 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 aditionally 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 additionaly 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. [16] This brings the overhead of updating the notation model every time, when the source model is updated. GLSP provides classes to make the syncronization of the noation model easier. The notation model ist 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 fuctionalities.

To achieve layouting in the Henshin Web editor, notation models for the metamodel, Hensin 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 createing the graphical model in the `GModelFactory`, the shape data from the noatation 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. [22] To connect the notation to an element the `NotationAdapter.getOrAssignNotation()` method checks if the element already has a notation, ether 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. [22, 7] During the transformation of the source model into the graphical model, elements need to be accesed multiple times. For example, a node is accesed 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, a index is created for each node, edge or attribute. Additionally the graphical model of GLSP needs identifiers for each element, to be able to spefify the element for operations on the source model. When no identifiers are specified, GLSP generates own unique identifiers for each element, but they can not be mapped back to the corresponding EMF element. To keep an uniqui 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 hold 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 usecase 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 models elements to the EMF elements, content hashes are used as identifiers. Each node in the source model is identifieable by the class name and their combined attribute values. These content hashes are stored in the notation model and generated every time the graphical model is created the first time in a session. The combination of the UUIDs and content hashes allows flexibility for editing the source model, while keeping the connection to the notation model.

V. Conclusion

This scientific work motivated a web-based model transformation graph editor and introduced all important frameworks needed for the planned project. Also related work and similar software was presented. With the discussed implementations of the POC, it was shown that the introduced frameworks can be used together to

create a web-based model transformation editor. The POC is a piercing, representing the workflow from the data source to the displayed graphical model. To achieve a fully functional editor, realizing all requirements, the POC can to be extended with the required features.

VI. Acronyms

GLSP Graphical Language Server Platform
EMF Eclipse Modeling Framework
MDE Model-Driven Engineering
UI User Interface
GUI Graphical User Interface (UI)
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
HTMI 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 Modelling

VII. Appendix

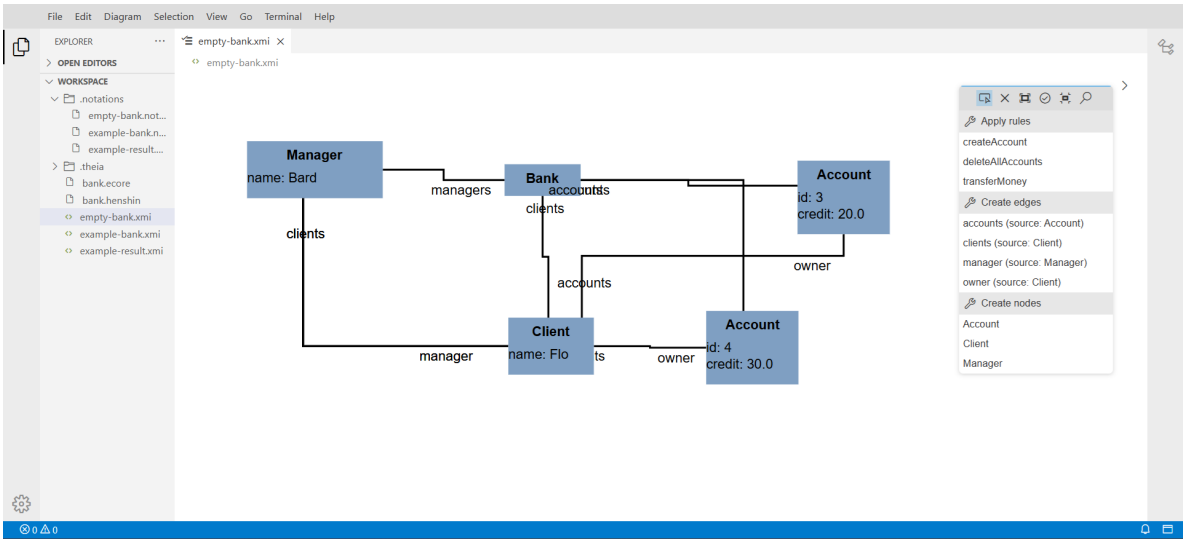

Figure 5. Henshin Web Theia POC graph editor

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 GLSP™ – Documentation*. Accessed: 2025-05-20. 2025. URL: https://eclipse.dev/glsp/documentation.

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

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

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

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

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

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

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

[12] MDE Group. *MDEForge*. https://github.com/MDEGroup/MDEForge. 2025.

[13] 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.

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

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

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

[17] 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.

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

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

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.