Marburg University

Software engineering group

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

Master thesis

from

Florian Weidner

CONTENTS

I. Motivation and introduction

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

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

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

II. Background

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

A. Eclipse Foundation

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

The scope of this work remains within the Eclipse Foundation ecosystem. All frameworks used are projects from the Eclipse Foundation. The used frameworks are described in the sections **??**, **??** and **??**.

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 **jdt!** (**jdt!**) and the **pde!** (**pde!**). Together they provide everything to develop and extend Eclipse-based tools. Equinox and the Platform are the core of the Eclipse **ide!**. With expanding the core with the **jdt!** or other plugins, the **ide!** can be used to develop different programming languages, like Java, C/C++, or PHP. [**emf**] Eclipse provides different packages to download, depending on the use case. One package is the Eclipse Modeling Tools package by the Eclipse Modeling Project. It provides tools and runtimes to build model-based applications. It can be used to graphically design domain models and test those models by creating and editing dynamic instances. Also, Java code can be generated from the models to get a scaffold that can be used to create applications on top. [**eclipse˙modeling**] The base of the Eclipse Modeling Tool is **emf!** (section **??**). Other modeling tools and projects that are built on top of the **emf!** core functionality provide capabilities for model transformation, database integration, or graphical editor generation. [**emf**]

B. *emf! (emf!)*

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

emf! (**emf!**) is the core part of the Eclipse Modeling Project and unifies the representation of models in **uml!**, **xml!** and Java. You can define your model in one of these formats and use **emf!** to generate the other formats.

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

- **EClass**: A class in the model that is identified by a name, containing attributes and references to other classes. It can also refer to a number of other classes as its supertypes to support inheritance. [**emf**]

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

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

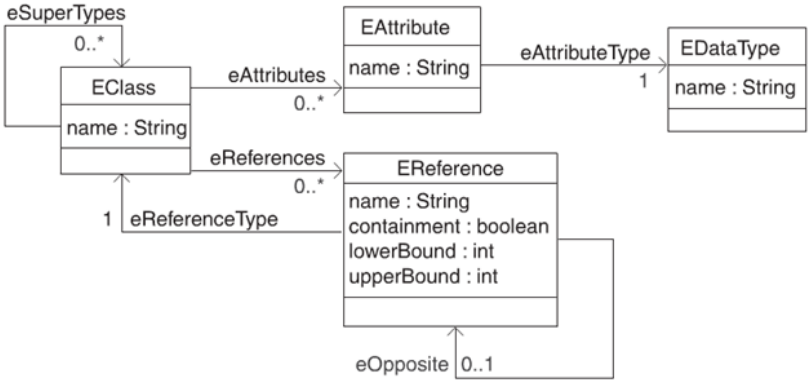

Figure 1. The Ecore kernel. Image obtained from [**emf**]

The second component of **emf!** is **emf!**.Edit. It provides generic reusable classes to build viewers and editors for **emf!** models. With these classes, **emf!** metamodels can be displayed in JFace viewers, that are part of the Eclipse **ui!**. [**eclipse˙emf**] The Eclipse **ide!** can display an Ecore model in a tree viewer. Eclipse accesses the data over the `ITreeContentProvider` interface to navigate the content and the `ILabelProvider` interface to provide the label text and icons for the displayed objects. The properties of objects are displayed in a Property Sheet over the `IPropertySourceProvider`, where the user can edit the model. **emf!**.Edit also provides undo and redo operations when creating or editing an instance model. For that, it uses a command framework with commands like an `AddCommand`, `SetCommand` or `CopyCommand`. [**emf**]

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

upon **emf!** are Eclipse Acceleo, Eclipse VIATRA, Eclipse ATL, Eclipse QVT Operational, Eclipse QVT Declarative and Henshin **??**.

C. Henshin

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

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

In the Eclipse **ide!**, rules can also be edited in a graphical editor. The rules are displayed as a single graph, calculated from the **lhs!** and **rhs!** graphs. The nodes and edges are annotated with *<<preserve>>*, *<<create>>*, *<<delete>>*, *<<forbid>>* or *<<require>>* to indicate what happens to the nodes and edges when applying the rule. These annotations can be directly edited in the graphical editor and the the **lhs!** and **rhs!** graphs are then adapted to the change. Also, multiple **nac!**s (**nac!**s), **pac!**s (**pac!**s) and parameters can be specified directly. [**henshin-repo**] When a set of transformation rules are specified, they can be applied to an **emf! xmi!** instance model, by using a wizard in the Eclipse **ide!**. There, the source model, the rule, and its parameters can be selected. The result of the transformation can be seen in a new **xmi!** instance file. [**henshin-repo**] Next to the graphical editor, Henshin also provides a textual syntax to define transformation rules and units. In a *.henshin_rule* file with the keyword **rule** a name and parameters, a new rule can be described. If you want to define a node, you can use the keyword **node** with a action keyword like **create** or **preserve** to specify the action of the node in the transformation.

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

*D. **glsp!** (**glsp!**)*

glsp! is a framework that provides components for the development of **gui!**s for web-based diagram editors. [**glsp-repo**] It is organized within the Eclipse Cloud Development project. [**glsp-doc**] With the framework,

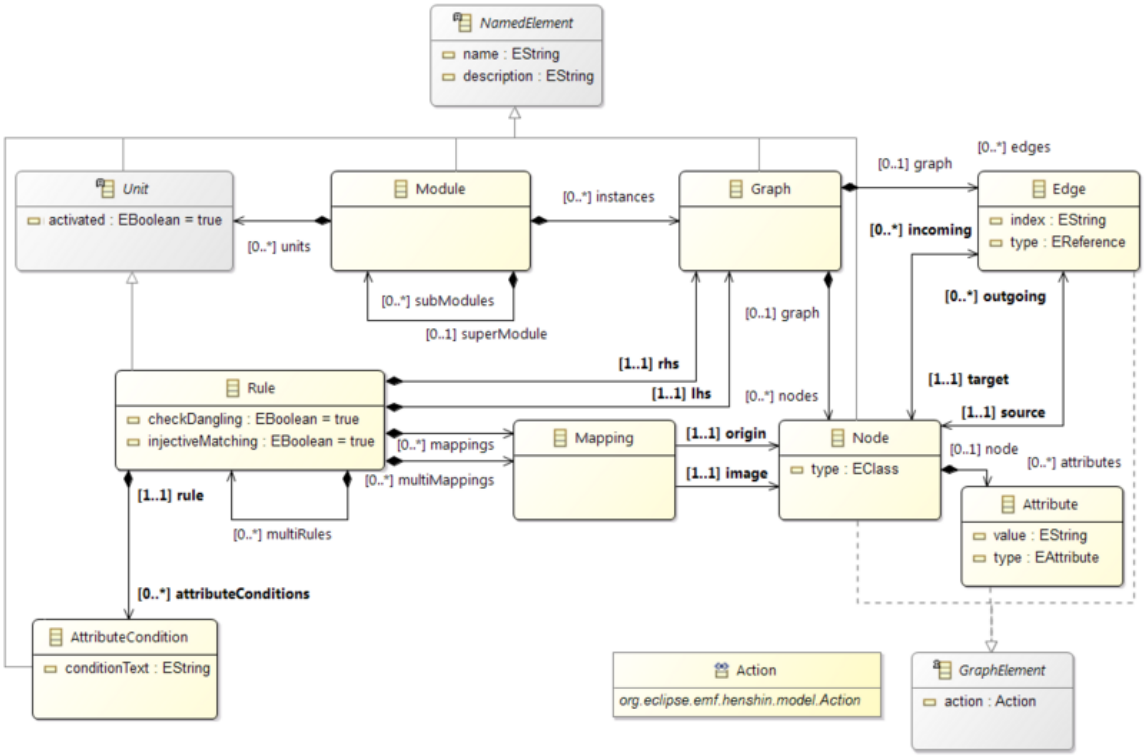

Figure 2. Henshin transformation rule metamodel. Image obtained from [**henshin-repo**]

custom diagram editors for Eclipse Theia, Eclipse IDE, Visual Studio Code, or standalone web apps can be created. It uses a client-server architecture, where the client is implemented with TypeScript and for the server, GLSP provides implementations in Java and TypeScript based on nodejs, even though the server could be implemented in any programming language. As the server for this project is implemented in Java, the following discussion focuses exclusively on the Java implementation of the **glsp!** server. Client and server communicate over JSON-**rpc!** with an action protocol that is similar to the Language Server Protocol [**lsp-repo**].

The **glsp!** server is responsible for loading a source model and defines how to transform it into the graphical model, that should be displayed. The source model can be of any format, e.g., a database, JSON file, or an EMF model. **glsp!** provides dedicated modules for loading EMF models. The Java server uses Google Guice [**guice-repo**] for **di!** (**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 **??** you can see that the diagram session **di!** container run inside the server **di!** container. **glsp!** provides some abstract base classes that have to be implemented to create a working diagram server language, that can provide a diagram to display at the client. All concrete implementations of one diagram language have to be registered in a `DiagramModule`. The server can handle multiple diagram languages by providing different diagram modules. There are some classes that have to be implemented. The interface `SourceModelStorage` defines how to load and save the source model. There is already a

default abstract implementation for **emf!** models, that loads the **xmi!** file into a `ResourceSet`. The interface `GModelFactory` is used to map the source model to the **glsp!** internal graphical model structure. Here also an abstract `EMFGModelFactory` is provided. Another important part is the `GModelState` interface, that defines the state of a client session and holds all information about the current state of the original source model. All services and handlers use the `GModelState` to obtain required information for their tasks.

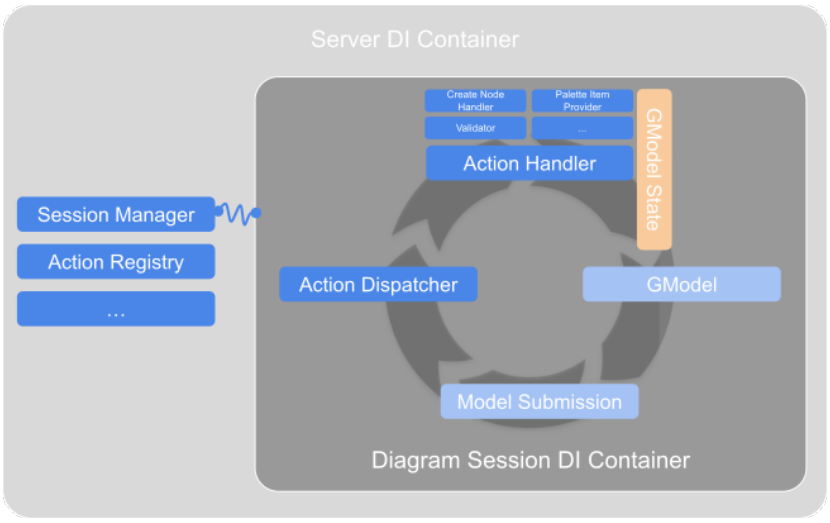

Figure 3. Server DI Container vs Diagram Session DI Container. Image obtained from [**glsp-doc**]

When the diagram should be displayed in the editor, the client sends a `RequestModelAction` with a **uri!** of the source model to the server. The server invokes the `SourceModelStorage` to load the source model and then uses the `GModelFactory` to translate it into the graphical model, which is then sent to the client to render it. For an edit operation, the client sends the operation request to the server, where the corresponding handler is invoked. The handler modifies the source model directly. After that, the server invokes the `GModelFactory` again to map the newly modified source model into a new graphical model, which is sent to the client to re-render. The two use cases share many steps. Since a new graphical model is created every time, the format of the source model is independent and can be of any format. [**glsp-doc**]

The **glsp!** client is responsible for rendering the graph and managing user interactions. The client requests all possible editing operations that can be performed on the specific model. As the client for this project is integrated into Eclipse Theia, the following discussion focuses exclusively on the Theia integration of the **glsp!** client. [**glsp-doc**] **glsp!** provides four main **ui!** components to apply commands or edit the graph but also allows custom **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. [**glsp-doc**, **glsp-repo**]
- **CommandPalette**: The CommandPalette can be invoked by pressing *Ctrl+Space*. It provides a search field to search for commands or actions that were registered. Commands can be registered by implementing and registering the `CommandPaletteActionProvider` to the server or implementing and

registering the `CommandContribution` interface to the Theia frontend module. [**glsp-doc**, **glsp-repo**]

- **ContextMenu**: The ContextMenu is a popup menu that can be opened by right clicking inside the diagram editor. There, any commands or actions can be structured as needed. It can be customized by implementing and registering the `ContextMenuItemProvider` to the server or implementing and registering the `MenuContribution` interface to the Theia frontend module. [**glsp-doc**, **glsp-repo**]
- **EditLabelUI**: Labels of nodes and edges can be edited by double-clicking on the label. The EditLabelUI provides an input popup to edit the label text. [**glsp-doc**, **glsp-repo**]
- **Custom ui! Components**: Custom **ui!** extensions have to extend `AbstractUIExtension` that provides a base **html!** element and can then be registered to the client. The base class also provides functionality to show, hide, or focus the element. These **ui!** extensions can also be enabled over a `SetUIExtensionVisibilityAction` from the server. [**glsp-doc**, **glsp-repo**]

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

The layout of a graph is divided into macro and micro layouting. The macro layouting, which arranges the nodes and edges of the model, is done by the server. The client does the micro layouting by calculating the positioning and size of elements within a container element. [**glsp-doc**] For the macro layouting, **glsp!** provides a notation model, that persists the position and size of the elements in a separate notation **xmi!** file. The notation diagram can be added to the `GModelState` and then used in the `GModelFactory` to specify the layout. [**glsp-repo**] **glsp!** also provides a `LayoutEngine` interface, that can be used to layout the elements of a graph that have no persisted layout yet. [**glsp-doc**]

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

III. Related Work

There are many existing tools for model transformations. **kahani2019survey** created a survey in **kahani2019survey** of various model transformation tools. They classified 60 different tools, including Henshin. In Figure **??**, 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 **ide!**s. 18% have no **ide!** support, and only two tools are web-based. In total, 89% of the tools have external dependencies such as an **ide!** or other tools. Dependencies often complicate the installation and usage of the tool. [**kahani2019survey**]

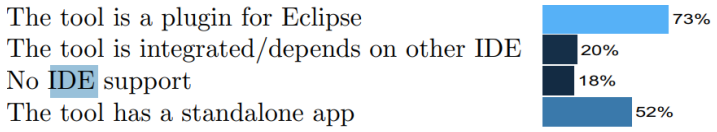

Figure 4. Execution environments of model transformation tools. Image obtained from [**kahani2019survey**]

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

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

WebDPF [**webDPF**] is another web-based modeling tool, published in **webDPF**. Compared to WebGME and **atompm!**, it supports model navigation and element filter capabilities, a JavaScript editor for writing predicate semantics, reusability of transformation rules, partial model completion, and a termination analysis. These features try to improve the usability of the tool. [**webDPF**] Even though the tool had improvments upon existing tools, the originally mentioned hosted WebDPF portal is offline by now.

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

IV. 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, **vscode!**, and as a standalone web application. Each integration brings different integration possibilities, deployment, and usage options for the editor. [**glsp-doc**] The main considerations for the deployment and usage are:

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

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

The other **glsp!** integrations are **ide!** integrations and therefore provide many features out of the box. For the Eclipse **ide!** integration, Eclipse has to be installed, and the **glsp!** plugin has to be added to the Eclipse installation. The plugin can be installed from the Eclipse Marketplace or manually by downloading the plugin jar file. [**eclipse-doc**] The **vscode!** integration also provides this option. The **ide!** can be installed and the **glsp!** editor can be added as an extension. The extension can be installed from the Marketplace or manually using a *.vsix* file. [**vscode-doc**] The **glsp! vscode!** integration can provide a *.vsix* file. [**glsp-repo**] **vscode!** is the most used **ide!**. 73.6% of developers use **vscode!** due to the survey of **stackoverflow2024survey** In **stackoverflow2024survey** [**stackoverflow2024survey**]. An advantage to Eclipse is that **vscode!** provides a browser version, which brings the same capabilites as the desktop **ide!**. [**vscode-doc**] So this integration provides the advantage that no **ide!** has to be installed to be able to use Henshin Web. The user can open **vscode!**, 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 **vscode!** [**stackoverflow2024survey**], but its focus is not to provide a ready **ide!** but to provide tools to create custom **ide!**s. The Eclipse Theia project is part of the Eclipse Foundation and is used as a basis to create your own **ide!**s based on web technologies. [**theia-doc**] They provide the Theia IDE that acts as a template editor and can be downloaded and used on all common operating systems or used in as a web editor in the browser. Due to the focus on providing a framework to build custom **ide!**s, 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 **??**.

- **vscode! extensions** Theia provides the **vscode!** extension **api!**, so that existing **vscode!** extensions can be used in Theia. They only interact with the **api!** and therefore can be installed at runtime.
- **Theia plugins** are working like **vscode!** extensions. They interact with the Theia plugin **api!** and can also access the **vscode!** extension **api!**. They can access some Theia specific features, that **vscode!** 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 **vscode!** extensions. They can also be installed at runtime and can access custom extended Theia backend services.
- **Theia extensions** are the core architecture parts of Theia. Theia is fully built using Theia extensions in

a modular way. The template Theia **ide!** contains Theia extensions, including the core. Custom Theia extensions can be developed and added to Theia with full access to all Theia functionality via dependency injection. They need to be installed at compile time. [**theia-doc**]

The **glsp!** Theia integration is creating a Theia extension, that is packed into a custom Theia **ide!**. It is also possible to use the **glsp! vscode!** integration that provides a **vscode!** extension, that can also be added to a Theia **ide!** at runtime. [**glsp-repo**] The option to use the diagram editor in the browser makes the **glsp!** Eclipse integration not interesting for Hensin Web. **vscode!** 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.

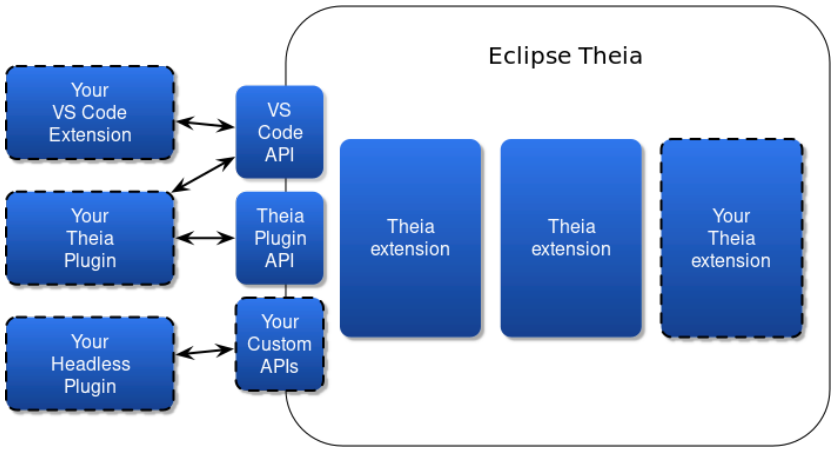

Figure 5. Theia high level extensions and plugins architecture. Image obtained from [**theia-doc**]

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

Because of the limited file access of the browser, the user has to upload and download all files to the server to use them. To be able to access the local file system of the user directly, the server needs to be hosted locally. For that, **glsp!** Theia application can be hosted in a Docker container. [**docker**] The Docker container can contain the Java server and the TypeScript client, that are started together. The user can then access the editor via a web browser. On a machine with a Docker environment, this solution can be started locally in an easy way and has the access to the file system. The Docker container can also be used to deploy

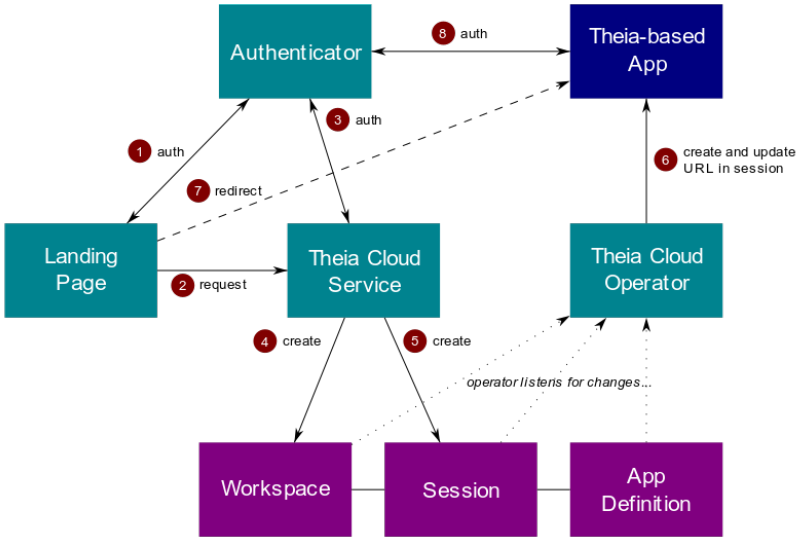

Figure 6. Interaction between Theia Cloud components. Image obtained from [**theia-cloud-doc**]

the application on a server so that it can be accessed by multiple users. The single Docker container solution doesn't provide as much scalability as using a cluster with Theia Cloud.

The **glsp!** Theia application can also be used as a desktop application. Theia uses Electron [**electron-repo**] to bundle the application into a desktop application, that can be installed via an installer. This approach also provides access to the local file system, since the electron application works like a self-hosted web application, and therefore the **glsp!** Java server is started locally. All in all, the **glsp!** Theia integration provides all different options to use the Henshin Web editor. Further clients can always be added later if needed.

V. Conclusion

This scientific work motivated the development of a web-based model transformation graph editor and introduced all the important frameworks needed for the planned project. **glsp!** is a fitting framework to create a web-based diagram editor, that already provides solutions for **emf!** source models. That makes the integration of the Henshin **sdk!** easier, as it is also based on **emf!**. **glsp!** also provides many features out of the box, that are needed for an intuitive diagram editor, like file or window management. The related work showed, that many model transformation tools rely on **ide!**s or other dependencies. There are not many web-based model transformation tools, that can be easily used. The advantage of Eclipse Theia, that provides a complete **ide!** in the browser, removes the issue of a complicated installation process and provides an easy to use editor. The deployment and usage section showed all options that **glsp!** provides. The Theia integration provides a modular, extensible and web-based solution to use for the Henshin Web editor. There are three main options to provide the editor to the users. It can be packaged as a desktop application, including an installer. The **glsp!** client and server can be hosted in a Docker container, that can be used to run locally or host in the cloud. The third option is to use Theia Cloud, that simplifies the deployment of Theia based products and provides easy to use features like user workspaces, authentication, and authorization.

VI. Acronyms

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

VII. Appendix

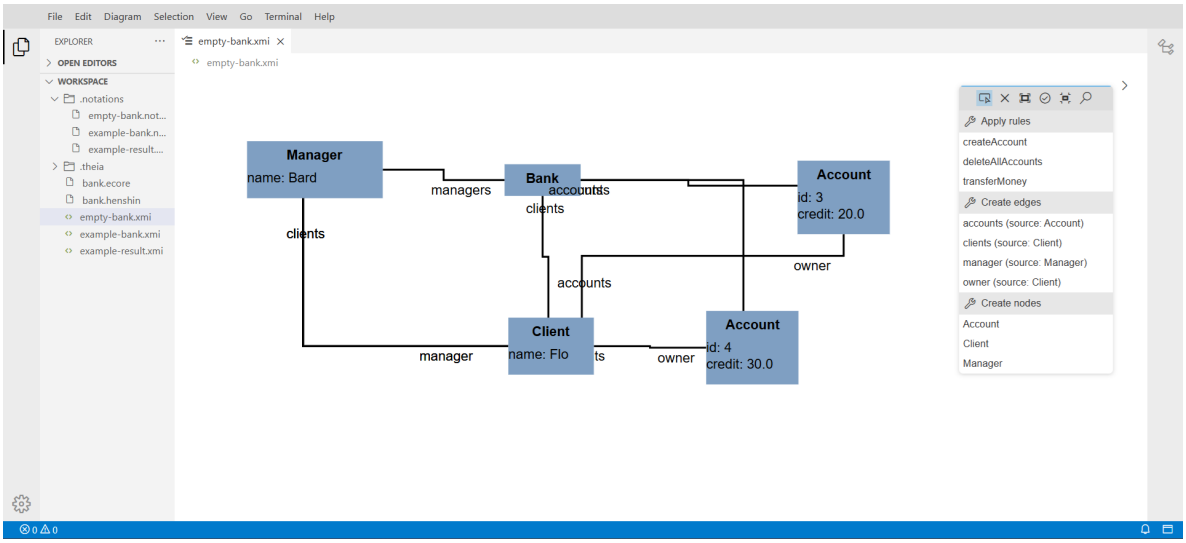

Figure 7. Henshin Web Theia **poc!** (**poc!**) graph editor