An internal DSL for graphical modeling tools based on GLSP

Georg Hinkel and Bodo Igler
RheinMain University of Applied Sciences, Germany

ABSTRACT The Graphical Language Server Protocol (GLSP) separates the development of graphical modeling language editors from the specification and processing of individual modeling languages. The open-source project Eclipse GLSP offers generic framework components that help create custom diagram editors (web-based clients) and language servers. However, the development of GLSP servers remains a complex and difficult task. This is mainly due to the inherent complexity of the server-side synchronization of the graphical model with the semantic model.
This paper applies recent advancements in model synchronization, in particular synchronization blocks, to simplify this task. We propose a language server infrastructure that separates the generic model processing from the details of the model synchronization for specific graphical modeling languages. The language-specific details are specified via a newly developed internal DSL. A generic server-side component based on the .Net Modeling Framework (NMF) performs the actual model synchronization.
We showcase the practicality of the DSL and the server architecture through a graphical class-diagram-like editor for NMF's meta-meta-model NMeta (similar to Ecore). The result is a concise, but fully functional prototype compliant with GLSP and partially based on Eclipse GLSP. Initial comparisons with existing GLSP implementations indicate that this approach significantly reduces the complexity and effort needed to develop graphical model editors. Moreover, this approach can be applied to any modeling language based on EMOF (Essential Meta Object Facility).

KEYWORDS Graphical DSL, GLSP, NMF, Synchronization Blocks

1. Introduction

Modeling systems requires appropriate editors. In many cases, graphical editors are desirable as they combine the editing process with a visualization of the model. Recently, there has been an increasing demand for web-based modeling tools due to their flexible deployment options (Rodríguez-Echeverría et al. 2018; Bork et al. 2024). Furthermore, models are typically edited collaboratively, which requires a distributed system with some central server to host the model instances.

In the past, many technologies have been introduced to simplify the development of graphical editors, but very often with deficits in terms of flexibility or deployment options. Editors built with technologies like GEF (Rubel et al. 2011) or technologies based on it are tied to desktops. Existing web technologies for graphical editors (Eclipse 2024; Martínez-Lasaca et al. 2023; Rocco et al. 2023; Zweihoff et al. 2019; Corley et al. 2016) mostly use a proprietary connection between frontend and backend and hence, require certain technology choices at either end, causing lock-in effects.

More recently, the Graphical Language Server Protocol (GLSP) (Bork et al. 2024) was introduced to separate the development of the actual user interface from the language semantics such that the user interface code becomes reusable, including integrations into common IDEs such as Visual Studio Code or Eclipse Theia (Bork et al. 2024). In GLSP, the client is responsible for the rendering of the diagram based on a graphical model while the server is responsible for the synchronization of

JOT reference format:
Georg Hinkel and Bodo Igler. *An internal DSL for graphical modeling tools:based on GLSP*. Journal of Object Technology. Vol. 24, No. 2, 2025.
Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2025.24.2.a11

the graphical model with the semantic model. An initial LOC analysis using the tool `cloc` on BIGUML (Metin & Bork 2023) as an example for a large project based on GLSP suggests that the server implementation requires significantly more code. In the source code of BIGUML 0.5.0, the backend consists of 1347 Java files with a total of 41.958 SLOC, while the frontend consists of 242 TypeScript files with a total of 9.789 SLOC[1]. In addition, most of the missing functionality such as undo/redo or copy/paste support is missing in the backend rather than the frontend.

This leads us to the research question of how to further simplify the development of fully-fledged graphical modeling tools based on GLSP. To this end, this paper examines the application of recent model synchronization technology results (Hinkel & Burger 2019; Boronat 2021; Buchmann et al. 2022) based on synchronization blocks. Synchronization blocks facilitate concise implementations of model synchronizations – even in the case of the heterogeneous model synchronization that a GLSP server has to perform. They are based on a solid theoretical foundation and exhibit desirable properties, in particular with respect to consistency and automation. This paper discusses the application of the formal concepts, derives requirements for a synchronization block based DSL for GLSP servers, proposes a new DSL based on NMF Synchronizations (Hinkel & Burger 2019) and validates the DSL by the construction of a class-diagram-like metamodel editor.

The remainder of this paper is structured as follows: In Section 2, we introduce the relevant foundations of GLSP and model synchronization with synchronization blocks. Section 3 discusses the usage of model synchronization blocks for graphical editors and derives requirements for a DSL for GLSP server-side model synchronization. Section 4 introduces a new DSL which is based on NMF Synchronization and which satisfies the stated requirements. Next, Section 5 explains the application of the DSL for a class-diagram-like metamodel editor. Section 6 describes the limitations of the prototype and discusses possible next steps. Section 7 compares our results with related work. Section 8 concludes the paper.

2. Foundations

Our approach is mainly based on two concepts: the GLSP protocol and model synchronization with synchronization blocks. This section briefly introduces both concepts.

2.1. GLSP

Compilers are necessary, but not sufficient for state-of-the-art software development. Developers expect e.g. a lot of productivity features to assist while editing (Juarez 2016). This support is typically integrated into an Integrated Development Environment (IDE). Microsoft has introduced the Language Server Protocol (LSP) (Bäumer 2023), in order to be able to provide full-featured IDE support for each popular programming language without having to implement a separate user interface for each of them. LSP allows reusing the implementation of a text editor (in many cases: Monaco (Microsoft 2024c)) for a variety of programming languages, as the new editor for a programming language essentially only has to provide an LSP server and some configuration[2]. The frontend only works on the level of text and all the semantic heavy lifting is performed by the LSP server. Microsoft lists more than one hundred LSP server implementations (Microsoft 2024a). As the Monaco editor is implemented in TypeScript and thus is able to run in a browser, the resulting editors are very flexible in terms of deployment. Hence, the usefulness of the protocol in particular for textual domain-specific languages has been proven multiple times.

Based on the success of LSP for textual languages, there has been an interest in applying a similar approach to graphical languages (Rodríguez-Echeverría et al. 2018). The Graphical Language Server Protocol (GLSP) (Bork et al. 2024) is an outcome of this idea. Like LSP, GLSP is based on the JSON-RPC (JSON-RPC Working Group 2013) protocol and establishes a bidirectional communication between the GLSP server and the frontend that merely renders the diagram. For this rendering, GLSP is based on a JSON representation of a graph, taken from the Eclipse Sprotty (Eclipse 2023) project. This model consists of elements that at least have an identifier, a type (e.g. node, edge, label), child elements and may optionally contain further information – such as layout information or other information that is necessary for the rendering. This additional information may depend on the type, e.g., a label typically requires the text that should be displayed, an edge requires ids of source and target elements. The client is responsible to render a diagram from this graphical model. Existing implementations render an SVG image in the browser using Eclipse Sprotty for this task.

Based on this graphical model, GLSP defines a number of commands (*actions*) that are sent to the server when the user interacts with the diagram. For example, there are actions sent to the server when the user resizes or moves an element, deletes an element, reconnects an edge or performs undo or redo operations. The GLSP protocol comprises type hints, too. The server provides these to the client in order to limit the actions offered to the user. The client decides on the basis of type hints whether an element may be resized, deleted or contained in a certain other element. The client will typically request the (static) type hints at the beginning of a session and keep them. For actions where the validity depends on the exact inputs (e.g. in order to check whether an edited text for a label is valid), dedicated actions are used to query the missing information from the GLSP server.

Unlike textual languages where the main editing operation, typing, is the same across all textual programming languages, the possible edit operations for a graphical language in terms of adding elements are highly specific to that language. Thus, GLSP foresees a dedicated action for the client to request the allowed operations in a given context, identified by a string. There are dedicated identifiers for the tool palette and the context menu. GLSP defines a set of standard actions like triggering a node creation or edge creation, but also allows the GLSP server to suggest custom operations. To this purpose, the actions are

[1] The numbers have been calculated using the tool *cloc 1.90*.

[2] The added configuration is an optimization: Rather simple features like syntax highlighting can usually be accomplished without a roundtrip to the LSP server.

processed using the same JSON-RPC method `process`, but use a polymorphic argument, making the protocol easily extensible. For custom operations, the JSON representation is only required to carry a property `isOperation` set to `true`. Whenever an operation has an impact on the graphical model, the GLSP server sends an updated graphical model to adjust the rendered diagram. As the actions are sent as JSON-RPC notifications, the server may send an updated graphical model at any given point in time. This facilitates e.g. concurrent write access to the same diagram from different sources.

The Eclipse GLSP project provides implementations for the client (in TypeScript, based on Eclipse Sprotty) and provides a number of libraries for the server, based on either Java or TypeScript. The Java implementations typically synchronize the graphical model with a semantic model implemented in EMF or an EMF.cloud model server. The TypeScript implementation merely operates directly on the graphical model. Eclipse GLSP also provides templates for the integration of the editor into Visual Studio Code, Eclipse Theia or Eclipse RCP.

2.2. Model Synchronization using Synchronization Blocks

NMF (Hinkel 2018b) supports model-driven engineering using .NET technologies. It is based on a theoretical framework formulated and analysed in a category-theoretical setting (Hinkel & Burger 2019; Hinkel 2018a) and comprises several generic technical components. Two components are particularly relevant in the context of this paper: NMF Expressions and NMF Synchronization.

NMF Expressions (Hinkel et al. 2019) is an incrementalization system integrated into the C# language. It automatically derives change propagation algorithms from function expressions. This works by setting up a dynamic dependency graph that keeps track of the model states and adapts when necessary. The incrementalization system is extensible and supports large parts of the Standard Query Operators (SQO[3]).

NMF Synchronizations is a model synchronization approach based on the algebraic theory of synchronization blocks. Synchronization blocks are a formal tool to run model transformations in an incremental and bidirectional way (Hinkel & Burger 2019). They combine a slightly modified notion of lenses (Foster et al. 2007) with incrementalization systems.

A synchronization block declares a synchronization action between two models in two potentially different mutable type categories. Consider e.g. the synchronization block in Figure 1. It consists of a morphism $\Phi_{\mathsf{Graph}\leftrightarrow\mathsf{StateMachine}}$, that provides a one-to-one correspondence between Graph nodes and StateMachine objects, and a morphism $\Phi_{\mathsf{Element}*\leftrightarrow\mathsf{State}*}$, that provides a one-to-one correspondence between the State child nodes of Graph nodes and the States attribute of StateMachine objects.

The correspondence between objects and object attributes is modeled with *well-behaved in-model lenses*. Lenses provide a generalized approach to the view-update problem for tree-structured data. (Foster et al. 2007) We adapt this approach slightly to model the concept of getter and setter methods for object model attributes and call these *well-behaved in-model lenses*. Each such lense consists of a side-effect free GET ("getter") morphism and PUT ("setter") morphism (in general with side effect). These morphisms need to be well-behaved, i.e. inverse to each other in the following sense:

- applying PUT with the value obtained by GET does not change anything (PUTGET law of (Foster et al. 2007))
- applying GET on an attribute set with PUT yields the set value (slightly modified version of the GETPUT law of (Foster et al. 2007))

In the example of Fig. 1 we have the lense

$$\mathsf{ChildNodes}[\text{type="State"}] : \mathsf{Graph} \hookrightarrow \mathsf{Element}*$$

for the access to the Element∗ child nodes of Graph nodes and the lense

$$\mathsf{States} : \mathsf{StateMachine} \hookrightarrow \mathsf{State}*$$

for the access to the State∗ attribute of StateMachine objects.[4]

The synchronization block in Fig. 1 is used to keep the Graph nodes and their child nodes Element∗ in the left model (b) of Fig. 2 in sync with their counterparts StateMachine and State∗ in the right model (c) of Fig. 2.

$$\begin{tikzcd}
\mathsf{Graph} \arrow[r, Leftrightarrow, "\Phi_{\mathsf{Graph}\leftrightarrow\mathsf{StateMachine}}"] \arrow[d, hook, "{\mathsf{ChildNodes}\,[\text{type="State"}]}"'] & \mathsf{StateMachine} \arrow[d, hook, "\mathsf{States}"] \\
\mathsf{Element}* \arrow[r, Leftrightarrow, "\Phi_{\mathsf{Element}*\leftrightarrow\mathsf{State}*}"'] & \mathsf{State}*
\end{tikzcd}$$

Figure 1 Schematic overview of bidirectional synchronization blocks applied for the synchronization of ChildNodes and States

The usage of lenses allows the isomorphism $\Phi_{\mathsf{Element}*\leftrightarrow\mathsf{State}*}$ to be enforced automatically and in both directions, if required, by evaluating the PUT operations: Whenever the synchronization engine finds elements obtained by a GET operation no longer isomorphic to the value obtained on the other side, it uses the PUT operation to fix the correspondence.

The idea of model synchronization using synchronization blocks is that a complex model synchronization can be obtained by stacking synchronization blocks. Typically, the first correspondence of a root isomorphism is known. Then, if two model elements are isomorphic with respect to some isomorphism, the synchronization blocks defined for this isomorphism yield model elements that should be isomorphic. If they are not, one can use the PUT operation on either side to enforce the isomorphism. This process is iteratively repeated until the synchronization terminates. Based on this formal notion of synchronization blocks and in-model lenses, one can prove that model synchronizations built with well-behaved in-model lenses

[3] (Microsoft 2007); SQO is a set of language-independent standard APIs for queries, specifically defined for the .NET platform.

[4] A collection type for a base type B is denoted as $B*$, where the star symbol "∗" denotes Kleene closures.

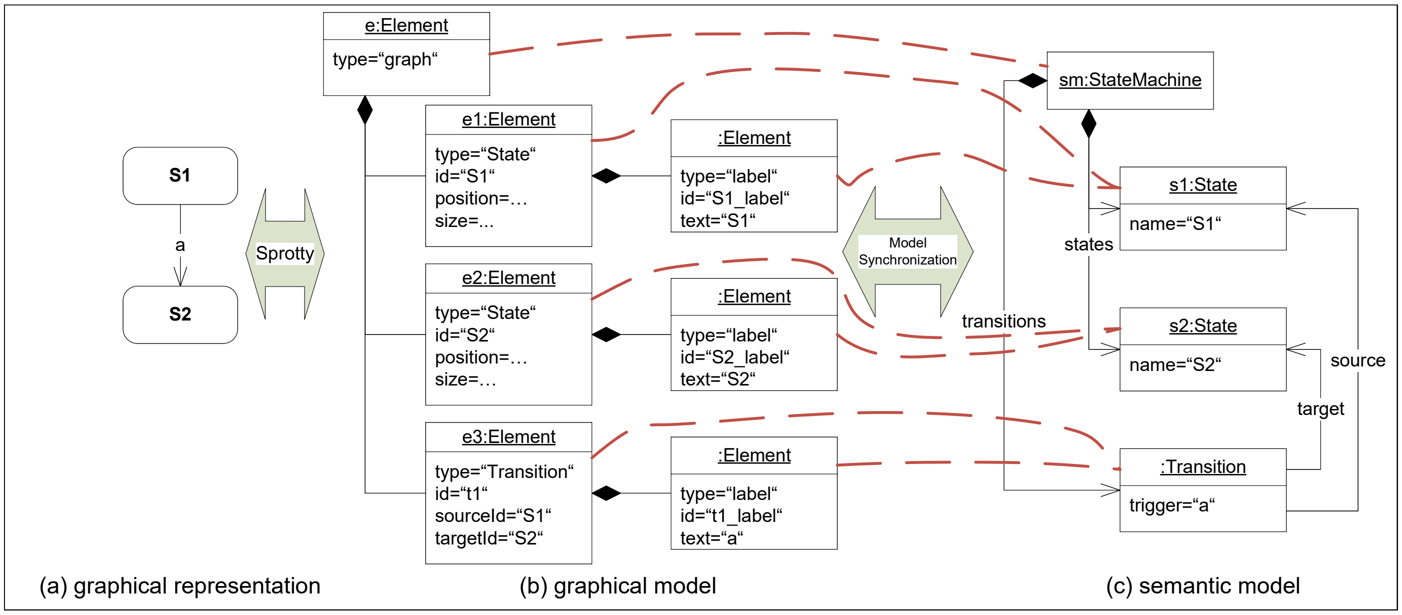

Figure 2 A simple state machine with two states and a transition as (a) graphical representation, (b) graphical model and (c) semantic model. Correspondences between graphical model and semantic model in red, dashed lines.

are correct and hippocratic, provided that the synchronization terminates (Hinkel & Burger 2019). This means that updates of either model can be propagated to the other model such that the consistency relationships are restored and an update to an already consistent model does not perform any changes.

Synchronization blocks have been implemented in NMF Synchronizations, an internal DSL with C# as host language (Hinkel 2015a; Hinkel & Burger 2019). It maps isomorphisms to nested classes of a model synchronization class. The synchronization blocks are specified in a dedicated `DeclareSynchronization` method through a dedicated API. The code corresponding to the synchronization block from Fig. 1 is depicted in Listing 1. The isomorphism $\Phi_{\mathsf{Graph}\leftrightarrow\mathsf{StateMachine}}$ and the types Graph and StateMachine are automatically derived from the surrounding context, i.e. the class `Graph_StateMachine` that represents this isomorphism.

```
Synchronize(SyncRule<Element_State>(), <f>, <g>);
```

Listing 1 Code for the synchronization block from Fig. 2

In this listing, `<f>` and `<g>` are placeholders for lenses that are usually specified as lambda expressions. NMF lifts the specification of the GET method to a lens by inferring the PUT operation. In case NMF Expressions cannot infer a PUT method, an annotation `LensPut` is provided to specify the PUT method explicitly. Alternatively, the `Synchronize` method also has overloads to pass in a custom PUT implementation.

For the incrementalization, NMF Synchronizations uses the extensible incrementalization system NMF Expressions.

Using the incrementalization system and the inversion based on lenses, NMF Synchronizations is able to lift the specification of a model transformation/synchronization in three orthogonal dimensions:

- **Direction:** A client may choose between transformation from left to right, right to left or check-only mode.
- **Change Propagation:** A client may choose between three change propagation options: whether changes to the input model should be propagated to the output model, also vice versa or not at all.
- **Synchronization:** A client may execute the transformation in synchronization mode between a left and a right model. In that case, the engine finds differences between the models and handles them according to the given strategy (only add missing elements to either side, also delete superfluous elements on the other side or full duplex synchronization).

This flexibility makes it possible to reuse the specification of a transformation in a broad range of different use cases. Furthermore, the fact that NMF Synchronizations is an internal language means that a wide range of advantages from mainstream languages, most notably modularity and tool support, can be inherited (Hinkel et al. 2017).

3. Using Model Synchronization Technology for Graphical Editors

The basic idea of our approach is twofold:

1. A GLSP server performs a continuous model synchronization between a graphical model and a semantic model.

2. Continuous model synchronizations can be adequately specified and performed via synchronization blocks.

This section briefly illustrates how synchronization blocks can be applied for this task and discusses the design of a corresponding DSL.

3.1. Using Synchronization Blocks for Graphical Editors

Graphical editors are used incrementally, i.e. the user will only change a small portion of a diagram at a time. In the setting of GLSP, the graphical model comprises only the information required by the frontend to render it as a graph. The semantic model contains all the semantic information to represent the concepts or system at hand. In general, the synchronization between these two models is heterogeneous as there is usually no model that can be extracted completely from the other model: Not all of the information from the semantic model is necessarily represented in the graphical model, other information such as layout information from the graphical is not present in the semantic model. Hence, reifying graphical editors as a model synchronization problem, we need bidirectional model synchronization approaches with support for incremental change propagation.

A model transformation paradigm that fits these requirements is model transformation through synchronization blocks. The various solutions to Transformation Tool Contest (TTC) case studies in the past (Hinkel 2015b, 2017; Anjorin et al. 2020; Hinkel 2020, 2021) have shown a wide applicability of model synchronization through synchronization blocks.

As an example, consider the case of a simple state machine that consists of two states connected by a transition and depicted as UML state machine diagram in Fig. 2 (a). In order to render such a diagram, GLSP editors typically use Sprotty, which internally uses a graphical model such as depicted in Fig. 2 (b). The graphical model is what the client gets from a GLSP server. To support downstream applications, ideally the GLSP server should synchronize this graphical model with a semantic model, in our example with the model in Fig. 2 (c).

To describe the synchronization of the graphical model in Fig. 2 (b) with the semantic model in Fig. 2 (c), one has to create synchronization blocks for the different kinds of correspondences (red dashed lines in Fig. 2). One of these synchronization blocks has already been sketched in Fig. 1 in Section 2.2. In the general case, we have isomorphisms like $\Phi_{\mathsf{Graph} \leftrightarrow \mathsf{StateMachine}}$ and $\Phi_{\mathsf{Element}* \leftrightarrow \mathsf{State}*}$ that describe how the different elements of the semantic model are represented in the diagram. These are connected through synchronization blocks that synchronize the child nodes or edges or labels of a graphical element using a lens from the semantic model. In the case of Fig. 1, we synchronize the states with the child nodes of the graph.

Applying model synchronization through synchronization blocks seems a viable candidate here, as the two theoretical properties of correctness and hippocraticness translate well to desirable properties for graphical editors: a correct synchronization means that the information shown in the graphical model is consistent with the information contained in the semantic model. Hippocraticness means that

1. changes made to the diagram that do not impact the semantic model (such as layout changes) would not change the semantic model and

2. changes made to the semantic model affecting properties that are not shown in the diagram will not change the diagram.

Such changes would probably be surprising to the user and thus should be avoided.

3.2. Requirements

We share the vision in (Bork et al. 2024) that GLSP is a solid foundation for the future development of flexible graphical model editors. To make it as easy as possible for language engineers to create new graphical languages, we see the following requirements to support language engineers using synchronization blocks:

R1: GLSP-specific model synchronizations NMF Synchronizations provides a generic DSL for synchronization blocks. It has to be adapted respectively enhanced in order to provide a tailored development experience when using the DSL. For instance, the above-mentioned heterogeneity phenomenon requires the treatment of n:m-relations between elements of the graphical and semantic models. This can be manually achieved by a combination of synchronizations blocks which were originally designed for 1:1-relations. However, the usability of the DSL benefits from dedicated language elements that address this phenomenon. Another example is the layout. As the layout information is agnostic of the domain and stored separately from the semantic model, developers of a graphical language should not have to specify it explicitly.

R2: Reusable graphical models The information needed by the clients in order to correctly render elements in the diagram typically depends on the specific use case: Sometimes, visual elements such as arrow types have a semantic meaning, sometimes not. In the former case, the graphical model needs to contain this information while in the latter case, it is often omitted in order to reduce the size of the graph. If the visuals cannot be adjusted using CSS classes, the graphical model needs to contain this semantic information. Therefore, the code in existing GLSP backend implementations often contains dedicated element classes to define these properties and a considerable amount of code is necessary to populate these properties and to register the corresponding code, e.g. with the dependency injection container. For instance, the information that a state is a final state is highly relevant for the graphical model as UML foresees an entirely different rendering of this state in the diagram that cannot be expressed only by CSS classes. However, in the semantic model this may be just a boolean attribute. Custom classes to represent the graphical model lead to bloated code. Therefore, we think that the language engineer should not have to bother and be able to reuse the graphical model.

R3: Metadata support GLSP requires servers to provide metadata that is difficult to provide using plain model synchronization approaches, sometimes very specific for GLSP. For instance, servers are requested to return shape hints and edge hints that the client uses to decide to which other nodes a given type of nodes can be added to or which nodes can be possible sources or targets of an edge. Furthermore, a GLSP server is also responsible for deciding about the actions possible in a given context, such as for the tool palette. In order to obtain that information in a flexible way (that also allows for localization of tool labels), the framework needs to provide metadata for the synchronization fitting with the abstractions needed for a GLSP implementation.

R4: Protocol support A framework supporting GLSP server development should abstract from protocol implementation details of GLSP.

3.3. Architecture

Fig. 3 sketches the architecture of a typical graphical editor using NMF-GLSP, in the form of an FMC[5] block diagram. Model-relevant user actions are performed by the backend. The main task of the frontend consists in propagating model-relevant user actions to the server, obtaining the changed graphical model from the GLSP server and rendering it in SVG (e.g., using the Eclipse GLSP client implementation).

In the diagram of Fig. 3, we printed components of the architecture that can be reused as unchanged packages[6] in blue (Eclipse GLSP client implementation) or green (NMF-GLSP). In the frontend, the language developer has to specify a rendering (in the example, states in the graphical model should be shown as rectangles with rounded corners) and potentially an extension to the graphical model, in case custom renderers require some semantic information in the graphical model (cf. R2). Meanwhile, the component to implement the communication with the GLSP server and to update the graphical model accordingly can be reused without modifications.

On the server-side of GLSP, NMF-GLSP allows us to keep the implementations of all the server-side GLSP actions and also the graphical model entirely agnostic of the graphical language such that the language developer can reuse these without modifications. Domain-specific actions such as custom operations use generic implementation and call into the synchronization to perform model actions. The layout metamodel and synchronization is also agnostic of the graphical language and can be reused, only the semantic (meta-)model of the language and the synchronization with the graphical model have to be provided. The dependency on the semantic metamodel is clear as this forms the abstract syntax of the graphical language. This metamodel can be provided either in NMeta or Ecore format. To simplify specifying the synchronization, we offer a DSL based on synchronization blocks.

4. NMF-GLSP

Our GLSP server framework provides the basis for the implementation of a concrete graphical language server. Such an implementation comprises the specification of the graphical language via NMF-GLSP and the invocation of this specification via dependency injection. The remaining server-side tasks (GLSP protocol implementation, model manipulation, undo/redo) have not to be specified via the DSL. They are either independent of the specific graphical model language or can be inferred by our framework from the DSL-based specification of the graphical language.

[5] Fundamental Modelling Concepts, (Knopfel et al. 2006)
[6] NPM packages for TypeScript, NuGet packages for .NET parts

```
protected override void DefineLayout() {
    Nodes(D<StateDescriptor>(), m => m.States);
}
```

Listing 2 Defining that the semantic property `States` should be used to render child nodes.

4.1. Structure of the Internal DSL

The basic idea of our DSL is very similar to the construction of NMF Synchronizations, which in turn is based on the construction of the NMF Transformation Language (NTL) (Hinkel et al. 2017): A graphical language is represented by a class that inherits from `GraphicalLanguage` and consists of rules as nested classes that inherit from a few abstract base classes provided by the DSL. In order to create the rules for the synchronization, the `GraphicalLanguage` class instantiates each nested type and generally identifies rules with their type. Then, synchronization blocks are specified as calls inside a method `DefineLayout`. As we know the target model statically, the available methods represent synchronization blocks tied to specific lenses at the graphical model.

The DSL defines three base classes, all of which take the semantic element type as a generic type parameter:[7]

– `NodeDescriptor<T>` for isomorphisms describing how to render a semantic model element as a node
– `EdgeDescriptor<T>` for isomorphisms describing how to render a semantic model element as an edge
– `LabelDescriptor<T>` for isomorphisms describing how to render a semantic model element as a label.

`EdgeDescriptor` defines methods for the creation of synchronization blocks for the source and target of the respective edge or labels along the edge. `NodeDescriptor` defines methods to create synchronization blocks to denote child nodes, edges, compartments and child labels. The `LabelDescriptor<T>` class only defines methods to adjust the label text. All of these classes inherit methods to adjust the type attribute of the graph elements (which the client uses to decide how the element is rendered), configure the CSS classes and forward information as static content.

In all cases, the classes representing the isomorphisms are also responsible for the creation of new elements. NMF provides reflection-based instance creation for generic type parameters (even if the generic type parameter is the interface for a metamodel class), but typically one wants to create pre-initialized objects. Therefore, all these classes have a common generic base class `ElementDescriptor<T>` that allows to override the creation of new elements.

The major advantage of this construction is that, like NTL and NMF Synchronizations, our DSL can inherit modularization concepts from the host language, i.e., it is rather easy to create reusable DSL modules and inherit the tool support from the host language.

[7] This structure is based on the generic graphical model structure as recommended by Eclipse GLSP.

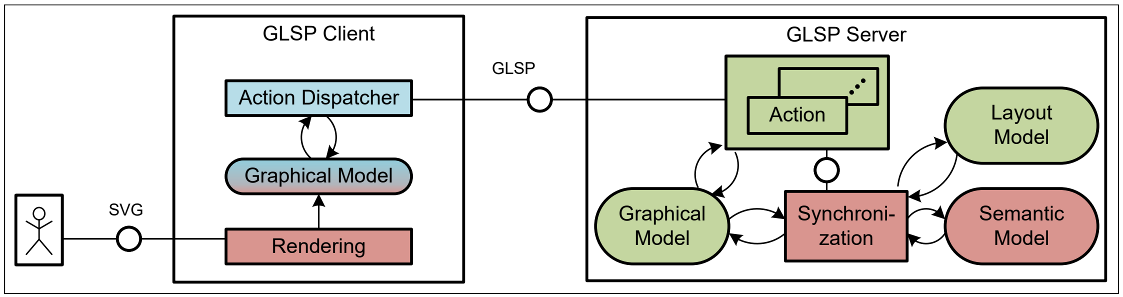

Figure 3 Architecture of a GLSP-based editor using our DSL. Reusable components from Eclipse GLSP-client in blue, reusable components from NMF-GLSP in green, parts with necessary custom code in red.

As an example of this usage, Listing 2 sketches how to create a synchronization block defining that the semantic elements of a property `States` should be rendered using the rule `StateDescriptor`. This listing uses a method `D` that selects the rule instance of a given rule type. The lens `.States` (= lense States in fig. 1) is denoted through its GET operation. NMF infers the PUT operation as well as the types and the lense of the left model from the context. The left lense ChildNodes[type="State"] is automatically derived from the generic GLSP graphical model structure.

Next to creating child nodes, another frequent requirement is to generate a label element for each semantic element and synchronize the text shown in this label with an attribute in the semantic model such as e.g., the name.

```
Label(e => e.Name);
```

Listing 3 Synchronizing the name of an element with the text in a label element

Creating a label with a text synchronized with a specific attribute is a frequent use case. This has dedicated support in the DSL. In particular, a minimal implementation is depicted in Listing 3. There are further parameters available to specify guards, fix the position of the label, to provide a custom PUT method or to specify explicitly whether users are allowed to edit the label text.

Frequently, especially in UML, nodes can have compartments. Therefore, compartments have a dedicated support in our DSL: The rules derived from `NodeDescriptor` are responsible for a node and a hierarchy of compartments instead of only a single element. To visually indicate these compartments, we use the `using`-syntax in C#.

```
using (Compartment("comp:header")) {
    Label(e => e.Name);
}
```

Listing 4 Snippet to define a label with the name of an element inside a compartment with the type *comp:header*.

In order to support inheritance in the semantic model, we also introduce the concept of refinements in the isomorphisms. This means, an isomorphism class can refine another if it describes a more concrete semantic element type. In this case, the synchronization blocks of both the refined and the refining isomorphism are used.

4.2. Invocation of the Internal DSL

The DSL integrates with ASP.NET Core: In order to expose a given graphical language to a GLSP server, the language needs to be added to the ASP.NET Core dependency injection container and another call is necessary to expose the GLSP server on a given path. We created an API that integrates with the minimal API provided by ASP.NET Core.

```
1 var builder = WebApplicationBuilder.Create();
2 builder.Services.AddWebSockets(opts => { });
3 builder.Services.AddGlspServer();
4 builder.Services.AddLanguage<StateMachineLanguage
      >();
5 // add more services ...
6 var app = builder.Build();
7 app.UseWebSockets();
8 app.MapGlspWebSocketServer("/glsp");
9 app.Run();
```

Listing 5 Glue code necessary to start the GLSP server

The minimal code to start a GLSP server is depicted in Listing 5. Line 2 loads the components to support web sockets, line 3 loads the general GLSP components, line 4 loads the graphical language. By repeating line 4, the server can also support multiple graphical languages simultaneously. Line 8 declares the entrypoint for the GLSP server as the websocket with address `/glsp`.

4.3. Implementation of the GLSP Protocol

In order to obtain the type hints for GLSP as well as context actions, we extended the DSL classes with the necessary UI information. In order to populate the tool palette, we iterate all the synchronization blocks present in the DSL instance where the target isomorphism is not the identity (which is the case e.g., for labels) and render an action to add an element to this synchronization block. If the class for the target isomorphism defines profiles, a separate action is generated for each profile.

For the type hints, we go through all of the isomorphism classes (i.e., the nested classes inheriting `ElementDescriptor<T>`) and generate a type hint for each of them, provided that the corresponding semantic class is not abstract. Depending on the type of description, this can be a shape type hint or edge hint. The necessary information such as other types that can be contained inside an element of that type can be drawn from the synchronization blocks for this isomorphism.

We also extended the lenses that create labels with an optional validator callback method in order to support a validation of label edits before the user applies the label.

The basic idea of the DSL, however, is that the developer ideally does not even see or need to know the GLSP protocol as it is entirely hidden in the abstractions offered by the DSL.

4.4. Model Manipulation

GLSP already defines operations to manipulate the graphical model: `CreateNodeOperation`, `DeleteNodeOperation`, `ChangeBoundsOperation` and similar operations that directly manipulate the graphical model. For these operations, the online incremental, bidirectional model synchronization makes these very easy to implement: They only need to operate on the graphical model and the model synchronization in the background takes care of the synchronization with the semantic model.

However, GLSP is an extensible protocol and allows the server to define custom operations. These can be used for context actions: the server can propose context actions attached to menu items. When the user clicks on these menu items, the client sends the associated `GetContextAction` to the server. The main use case for such custom operations is to support domain-specific operations, such as for instance toggling whether a state is a start state. To support these, NMF-GLSP contains a generic operation implementation that uses a callback provided through the DSL.

4.5. Undo and Redo

The GLSP protocol provides operations for undo and redo, but implementing an undo and redo functionality can be quite challenging. However, NMF has a builtin generic operation recorder that uses events generated by the code generator to produce a model of changes as they are being applied to the model.

This generic operation recorder simplifies the implementation of custom operations heavily as it does not require developers to provide an undo operation for custom operations. Instead, the change recorder records the changes that the operation introduced to the model, inverts them and applies the inverted changes if necessary. This also means that the redo operation reapplies the exact original changes, such that undo and redo cancel out each other, even if the original operation was not deterministic.

4.6. Layout

In GLSP, the GLSP server is also responsible for creating an initial layout. For this, we integrated the Microsoft Automatic Graph Layout library (MSAGL) (Microsoft 2024b). We allow a graphical language to override the default layouting algorithm but default to a $\frac{\pi}{2}$ rotated Sugiyama layered layout algorithm (Sugiyama et al. 1981) with rectilinear edge routing.

```
public override ILayoutEngine DefaultLayoutEngine
    => new LayeredLayoutService(new()
{
    EdgeRoutingSettings = { EdgeRoutingMode =
    EdgeRoutingMode.StraightLine }
});
```

Listing 6 Changing the default layout engine to straight lines

As an example, Listing 6 shows how to change the layouting strategy to straight lines (and no rotation). As the layout engine is abstracted in an interface, one can also easily exchange the layout library to use something else than the MSAGL.

4.7. Language Composition

Applying the DSL construction from prior work (Hinkel et al. 2017), our language inherits composition functionality. In particular, a language engineer could reuse a `GraphicalLanguage` implementation and override some rules by inheriting from the graphical language root class and inserting a public nested class inheriting from the descriptor the developer wishes to extend or overwrite. As the DSL is an internal DSL in C#, artifacts like reusable graphical language libraries would be technically feasible. However, we have not applied this functionality in a practical use case, this will be subject to future research.

5. A metamodel editor for NMeta

We use the implementation of a graphical editor for the graphical model language NMeta[8] in order to demonstrate how the different bits of our DSL can be applied in an example with real-world complexity.

5.1. The language basics

To create a new graphical language, we first specify the abstract syntax of the graphical model language NMeta via a meta-meta-model language (Ecore or NMeta) and have our framework generate the corresponding meta-model in C#. We then create a class inheriting from `GraphicalLanguage` and override the abstract members, denoting the diagram type and the start isomorphism, called start rule.

```
internal partial class NMetaLanguage :
    GraphicalLanguage {
  public override string DiagramType => "nmeta";
  public override DescriptorBase StartRule
     => Descriptor<RootDescriptor>();
  ...
}
```

Listing 7 Definition of the graphical language

The implementation is depicted in Listing 7. The diagram type is necessary as a single GLSP server is allowed to support multiple languages simultaneously. If the language is integrated into an IDE, this typically matches the file extension. To declare the start rule, we reference it by type. The rest of the class consists of the rules and private helper functions used by multiple rules.

5.2. The root descriptor

We begin with the start rule of the graphical editor. Metamodel editors are like class diagram editors, so the surface consists of nodes for types or child packages (in NMeta called namespaces) and edges for associations (in NMeta called references) and inheritance relations between classes.

[8] NMeta is the meta-metamodel used in NMF and structurally identical to Ecore. In particular, NMF includes a model transformation from Ecore to NMeta (Hinkel 2018b).

```
1  public class RootDescriptor : NodeDescriptor<
       INamespace> {
2     protected override void DeclareLayout() {
3        Nodes(D<TypeDescriptor>(), n => n.Types);
4        Nodes(D<NamespaceDescriptor>(), n => n.
       ChildNamespaces);
5
6        Edges(D<ReferenceDescriptor>(),
7         n => n.Types.OfType<IReferenceType>()
8               .SelectMany(c => c.References)
9               .IgnoreUpdates());
10       Edges(D<ClassDescriptor>(), D<
       ClassDescriptor>(),
11        n => new BaseClassCollection(n))
12          .WithLabel("New Base-Class")
13          .WithType("edge:inheritance");
14    }
15 }
```

Listing 8 The root descriptor implementation

In the DSL, the reference to child nodes is implemented in the synchronization blocks depicted in lines 3 and 4 of Listing 8. The type `RootDescriptor` (definition in line 1) represents the isomorphism $\Phi_{\texttt{Node}\leftrightarrow\texttt{Namespace}}$, as it connects the types `NodeDescriptor` (node type in the generic graphical model) and `INamespace` (name of the C#-interface that has been generated for the meta-model class `Namespace`). The remaining two types for the synchronizations blocks in lines 3 and 4 are derived by the framework from the context.

As the language infers lenses from the lambda expressions, there is a single specification used both to infer the graphical model and to perform updates on the semantical model in response to an operation on the graphical model. In particular, the framework understands that, as a namespace uses types as nodes, the user should be able to create any concrete type and add it to the types lens. Hence, the framework automatically calculates what GLSP calls type shape hints, static information provided by a GLSP server to visually assist a user when creating a node where this node can be created. Further, the framework automatically populates the palette, offering users to add types to a namespace. This is in contrast to the GLSP reference examples where developers have to develop classes for and thus understand each of these GLSP features separately. Large existing editors such as the BIGUML editor, employ their own frameworks to simplify this process but currently cannot hide protocol details such as node creation handlers or type hints from the developer.

The references to child nodes manifest themselves in a semantic model reference (`Types` and `ChildNamespaces`), hence NMF Expressions is able to automatically turn the specification into a lens. We do not need to provide a PUT method. For the edges, i.e. the references of classes contained in the namespace, this is different. Therefore, NMF cannot infer an operation that adds an element to the query result. However, this is not necessary as the `References` reference is an opposite of the `DeclaringType` reference which will be used to map the source of the edge. Hence, it is safe to ignore updates to the collection, implemented in lines 6–9 in Listing 8.

```
public BaseClassCollection(INamespace ns)
 : base(ns.Types.OfType<IClass>().SelectMany(
    c => c.BaseTypes,
    (c,baseClass) => ValueTuple.Create(c,
    baseClass)))
{}
```

Listing 9 The query expression to obtain inheritance links in the metamodel

For the base classes of a class, this is different as there is no opposite reference that we can use. Therefore, we need to implement a custom collection class. As NMF is able to infer change propagation, this essentially means that we need to implement the methods `Add`, `Remove` and `Clear`. The incrementalization of the query expression can be done by NMF, by passing the query expression to the constructor, depicted in Listing 9.

With the help of this custom collection, we can specify the synchronization block to create inheritance edges as depicted in lines 10–13 of Listing 8: inheritance edges connect classes to classes and we use the custom collection to denote between which classes an inheritance edge should be drawn. Creating such an edge from tools such as the palette should have the label *New Base-Class* and the created edge should have the type `edge:inheritance`. The advantage here is that the developer stays in the semantic model, specifying what it means if an inheritance link as a tuple of two classes is added or removed. Unlike GLSP reference implementations, the developer does not have to bother with lower level protocol details of GLSP. The only leftovers from the protocol are those that are required by the frontend, in this case the type and display name of the generated edge.

5.3. Classes

In NMeta, classes are a special kind of reference type. Classes appear in class diagrams as boxes with three compartments, one for the name, one for attributes and one for operations. These compartments are typically laid out in a vertical box.

```
1  Refines(D<ReferenceTypeDescriptor>());
2  Layout(LayoutStrategy.Vbox);
3  using (Compartment("comp:header")) {
4      Label(e => e.Name).Validate(ClassNameRegex(),
        "Not a valid class name!");
5  }
6  using (Compartment("comp:divider")) { }
7  using (Compartment("comp:comp")) {
8      Labels(D<AttributeDescriptor>(), c => c.
       Attributes);
9  }
10 using (Compartment("comp:divider")) { }
11 using (Compartment("comp:comp")) {
12     Labels(D<OperationDescriptor>(), c => c.
       Operations);
13 }
```

Listing 10 Rule to describe graphical contents for a class

In the DSL, the implementation of this specification is depicted in Listing 10. Line 1 of this listing specifies that classes are a kind of reference type. As `ReferenceTypeDescriptor` similarly refines `TypeDescriptor`, the framework will populate the palette with a command to add a class to a namespace. Line 2 specifies that the children of classes are to be laid out using a Vbox algorithm. Lines 3–5 specify that the first compartment should contain the name of the class, using a regular expression to check the validity of a name. Lines 6 and 10 add compartment dividers. Lines 7–9 specify that the second compartment should consist of the attributes that are to be rendered using the `AttributeDescriptor` class. Similarly, lines 11–13 specify that the next compartment should consist of the operations.

As before, labels are specified as lenses such that no further specification is required to support label edits, again in contrast to the GLSP reference implementation. In case additional information is required for GLSP, this extra information is added explicitly and in terms of the abstract syntax rather than GLSP protocol classes. As the use of regular expressions is a frequent and very convenient way of validating label contents, we created a convenience overload to accept a regular expression and pass it directly.[9]

5.4. Attributes

An attribute in a class diagram appears as a label, therefore the DSL class `AttributeDescriptor` inherits from the base class `LabelDescriptor<IAttribute>`. The main responsibility of a label descriptor consists in the specification of how the text of the label is constructed and set. The label for an attribute is constructed from multiple properties of an attribute and encodes the name, the type and bounds.[10]

```
1 Label(a => a.Name + (a.Type != null ? (" : " + a.
      Type.Name) : string.Empty)
2           + " [" + GetBounds.Evaluate(a) + "]"
3           + (a.DefaultValue != null ? " = " + a.
      DefaultValue : ""))
4     .Validate(AttributeRegex(), "not a valid
      attribute")
5     .WithSetter(SetAttributeFromString);
```

Listing 11 `DefineLayout` implementation for `AttributeDescriptor`

This encoding is depicted in Lines 1–3 of Listing 11. As the rendering of the bounds string is used multiple times in the NMeta editor, it is implemented as a reusable method. In order to make the structure of the method visible to the incrementalization system NMF Expression, the lambda expression is passed as a code model rather than a delegate. This way, NMF can automatically infer when the label text needs to be updated and hence, the developer does not need to specify this. As NMF can only infer an incrementalization, but not an inversion of generic string operations, we need to specify a method to invert the encoding in Line 5. Otherwise, the framework does not allow to edit the text of the label. Line 4 specifies that updated values for an attribute should be checked against the specified regular expression.

The signature of the method `SetAttributeFromString` is depicted in Listing 12. As the DSL incrementalizes the label expression, this method is only responsible for updating the semantic model based on the input string.

[9] The reason that the regular expression is returned from a method in Listing 10 is that we rely on compile-time source code generation from a regular expression in .NET.

[10] Similar to Ecore but unlike UML, NMeta does not contain a definition of access modifiers.

```
private static void SetAttributeFromString(
    IAttribute attribute, string attributeString)
```

Listing 12 Signature for the `SetAttributeFromString` method

This lens construction yields a very powerful editing capability as the user may change the name, bounds, the type and the default value with a single label change. During editing, the GLSP server is constantly asked for feedback whether the current attribute string is valid. After the user has entered the string, it is automatically formatted correctly as the server actually uses the provided PUT operation to set the properties in the semantic model. This results in an incremental update of the GET operation.

5.5. References

References in NMeta are special as they have to be rendered as edges that correspond to either one of two semantic model elements: A reference and its opposite (if it exists) are rendered as a single edge. However, if a reference has an opposite, we want both the name and bounds of the reference and its opposite to be shown.

```
1  Label(r => r.Name)
2      .At(0.5, EdgeSide.Top, offset: 10)
3      .Validate(IdentifierRegex(), "not a valid
       identifier");
4  Label(r => r.Opposite.Name)
5      .If(r => r.Opposite != null)
6      .At(0.5, EdgeSide.Bottom, offset: 10)
7      .Validate(IdentifierRegex(), "not a valid
       identifier");
8  Label(r => GetBoundsString.Evaluate(r))
9      .At(0.9, EdgeSide.Top, offset: 10)
10     .Validate(BoundsRegex(), "not a valid bounds
       string")
11     .WithSetter(UpdateBounds);
12 Label(r => GetBoundsString.Evaluate(r.Opposite))
13     .If(r => r.Opposite != null)
14     .At(0.1, EdgeSide.Top, offset: 10)
15     .Validate(BoundsRegex(), "not a valid bounds
       string")
16     .WithSetter(UpdateOppositeBounds);
17
18 SourceNode(D<ReferenceTypeDescriptor>(), r => r.
       DeclaringType);
19 TargetNode(D<ReferenceTypeDescriptor>(), r => r.
       ReferenceType);
20
21 Forward("renderEndArrow", r => r.Opposite == null
       );
22 Forward("renderComposition", r => r.IsContainment
       );
23
24 Profile(Bidirectional);
25 Profile(Containment);
26
27 Operation("Toggle Containment", (r,_) => r.
       IsContainment = !r.IsContainment);
```

Listing 13 `DefineLayout` implementation for `ReferenceDescriptor`

Listing 13 depicts the `DefineLayout` implementation for the `ReferenceDescriptor`. Lines 1–3 specifiy that the name of the reference should be visible on top of the middle of the reference with an offset of 10 pixels to the edge. If the user edits the label, it should validate against a regular expression. Lines 4–7 create a label for the opposite reference, but only if the opposite reference is actually present. Similarly, lines 8–11 render the bounds of the reference near the reference target whereas lines 12–16 render the bounds of the opposite reference, if present.

Lines 18 and 19 define the synchronization blocks that connect an edge to its source node and target node. Both source and target are set to simple property accesses. NMF can infer a PUT operation in this case and the DSL enables the user to reconnect the edges.

Lines 21 and 22 forward two query results to the client: Is the edge bidirectional (i.e. is there an opposite present)? Is the edge a composition? The client decides on the basis of this information whether to render an arrow symbol or a composition diamond.

Next, lines 24 and 25 define two additional profiles for references, encoding the profile with a simple string. With these profiles, the framework provides three different actions for the creation of a reference – a standard reference, a bidirectional reference and a containment reference. When the user chooses one of these actions, the `CreateElement` method of the `ReferenceDescriptor` class is used to instantiate the reference.

Lastly, line 27 declares a custom toggle operation (containment: true/false).

```
1  public override IReference CreateElement(string
       profile, object parent) {
2    var reference = new Reference {
3      Name = "NewReference",
4      IsContainment = profile == Containment,
5      Opposite = profile == Bidirectional ? new
       Reference { Name = "Opposite" } : null,
6    };
7
8    SetHooks(reference);
9    SetHooks(reference.Opposite);
10
11   return reference;
12 }
```

Listing 14 Implementation of `CreateElement` for the `ReferenceDescriptor`

The implementation of `CreateElement` is shown in Listing 14. A new reference is created. Depending on the profile parameter, it is a containment reference and optionally has an opposite. Note that the `CreateElement` method does neither set the source or target of the edge, as these will be set using the synchronization blocks from lines 18 and 19 of Listing 13. However, for a bidirectional reference, the type of the reference is always the declaring type of the opposite and vice versa. This consistency relation is not defined in the metamodel and therefore should be ensured by the editor. Setting these hooks is done in lines 8 and 9 of Listing 14.

5.6. Evaluation

Fig. 4 shows a screenshot of the NMeta editor: a metamodel for finite state machines is being edited. The palette on the right side of the screenshot is populated based on the synchronization blocks specified in the DSL. Palette entries are grouped by the name of their semantic enclosing type by default. In particular, the tool palette options for the creation of a new class, a new enumeration or a new namespace are derived from lines 3 and 4 of Listing 8 and the fact that enumerations and classes are the only non-abstract implementations of types. The three options for the creation of a new reference are a consequence of the synchronization block in lines 6–9 of the same listing. There are three different entries, as the `ReferenceDescriptor` declares two additional profiles (lines 24 and 25 of Listing 13). The option for the addition of a new base class is a consequence of lines 10–13 in Listing 8. Lastly, the context menu for a reference is populated based on the operations provided for references, i.e. line 27 of Listing 13. The context menu entry to delete the element is currently generically available.

Although the editor is only a prototype, it has full support for undo and redo operations (cf. Sec. 4.5), as NMF uses a generic operation recorder to record changes performed on the model. The implementation of the editor is available online[11].

6. Discussion

This section describes the limitations of the prototype and discusses potential next steps:

- The DSL restricts edges to start and end at nodes[12]. The corresponding assumption in GLSP is mainly due to a similar assumption in Sprotty. As soon as Sprotty supports edges starting at edges, we can extend our DSL.
- Regarding custom operations, we currently provide all operations that are bound to one selected element and allow users to declare these operations in the rules to describe the visual appearance of these elements. Operations that require multiple inputs are not supported yet. This can be implemented as an extension to the DSL.
- We have not applied the framework to a diagram type with a semantic notion of ports. Mainly, ports do not impact the graph model and therefore hardly influence the DSL as most of the changes affect the frontend, only. However, the layouting engine needs to know that a child graphical element is to be rendered as a port in order to get the initial layout right.

All discussed changes to the existing DSL can be achieved as extensions, i.e. without impact on the existing API.

Creating the NMeta editor, we also noticed that although the DSL reduces it, there is still quite some manual code necessary, especially when processing label changes. This supports our design choice for an internal DSL as such a DSL typically makes it easy to integrate code of the host language. However, it also

[11] The full language implementation is available at https://github.com/NMFCode/NMF/blob/main/Glsp/NMetaGlspEditor/NMetaGlspEditor.Server/Language/NMetaLanguage.cs, the frontend implementation is available at https://github.com/NMFCode/nmf-web-client/tree/main/packages/nmeta-glsp-client.

[12] Association classes in UML class diagrams and reference refinements in NMeta are two examples for situations in which an edge is visullay connected to another edge.

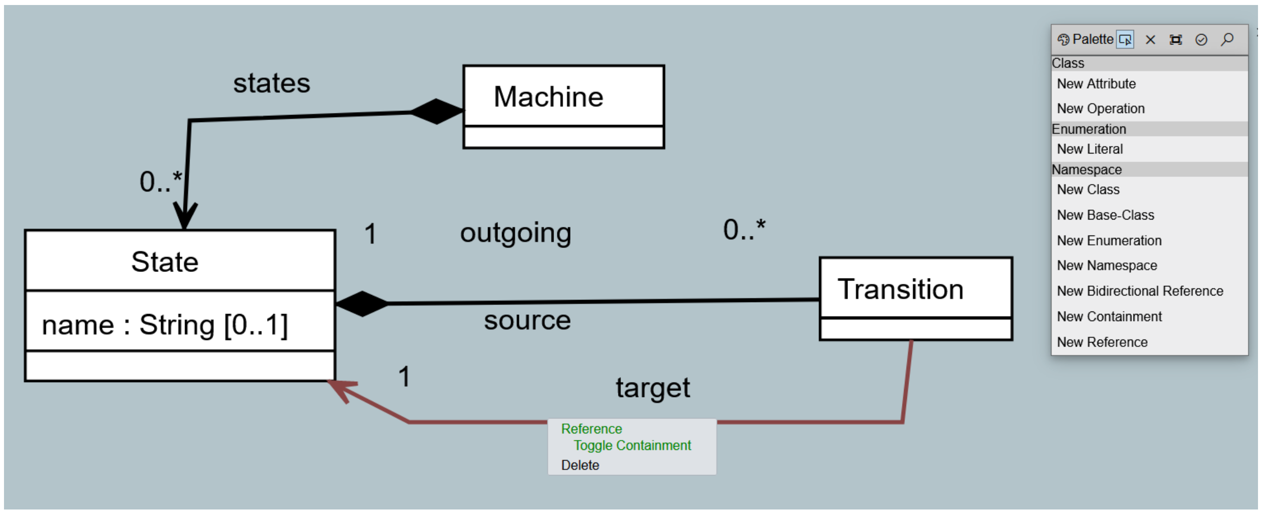

Figure 4 Screenshot of the resulting editor creating a metamodel for finite state machines

limits the degree to which our approach can be generalized to other programming languages as the expression tree feature of C#, that we use (through NMF Expressions) to obtain incremental change propagations for lambda expressions, is rather rare. If such a syntax feature is missing, such as in Java, one can either go for an external DSL or use a framework such as Active Operations (Calvar et al. 2019) to achieve a similar syntax.

7. Related Work

We divide related work into GLSP implementations, graphical editors using model transformation technology and graphical editors using web technologies.

7.1. GLSP Implementations

The BIGUML modeling tool is one of the first UML editors based on GLSP and the reference architecture for GLSP (Metin & Bork 2023). BIGUML is under development and provides currently basic support for several UML diagram types. In contrast to our approach, a high proportion of the code is devoted to the backend, as domain specific actions and extensions to the graph model need to be implemented by the language engineer. We believe that the language engineer has to have a much better understanding of the GLSP protocol than with our DSL.

7.2. Graphical editors using transformation technologies

Our approach is not the first to utilize model transformation technology to develop graphical editors. With EuGENia (Kolovos et al. 2010), Kolovos et al. used model transformations specified in ETL to simplify the development of graphical editors using the GMF framework. However, these transformations are non-incremental and are run at compile-time. The novelty of our approach is that the model synchronization is executed at run-time and incrementally, in order to continuously keep the graphical model consistent with the semantic model.

Similarly, Picto (Kolovos et al. 2020) allows to create graphical views of a model through a model-to-text transformation. With Picto Web (Yohannis et al. 2024), there is also a tool that creates a web application rather than an Eclipse plugin from such a specification. However, both Picto and Picto Web are limited to view capabilities where the GLSP protocol adds editing capabilities exposed by our implementation.

Closest to our approach, Rath et al. propose to use bidirectional, online incremental model synchronizations (called live transformations) to synchronize the concrete syntax with the abstract syntax of a model (Ráth et al. 2010). This is also what we do in our approach, but we use a dedicated internal transformation DSL instead of a general transformation language (cf. Section 3.2). We believe that this DSL could be more declarative than the transformation rules presented in this paper as our DSL allows the developer to specify correspondences instead of triggers, but such a comparison will be subject of future work.

7.3. Graphical editors using web technologies

Several technologies exist to simplify the process of creating graphical domain specific languages (Eclipse 2024; Martínez-Lasaca et al. 2023; Rocco et al. 2023; Zweihoff et al. 2019; Corley et al. 2016). As the connection between the frontend that displays diagrams to the user and the backend that stores the semantic model is proprietary in such tools, the deployment options of the resulting editors are limited as the user typically has to use the entire platform. Integrating these editors into existing editors like Visual Studio Code or entirely tailored editing environments such as those based on Eclipse Theia can therefore be rather difficult.

The Sirius Web project (Eclipse 2024) facilitates the development of graphical editors based on web technologies by letting developers model their editor. Whereas our DSL is an internal DSL that makes it easy for developers to insert custom code, Sirius Web uses a tree editor for a formal viewer model, relying on OCL-like expressions for navigation. Sirius Web lacks an abstraction (like e.g. lenses) from model features. This leads for many features to a higher degree of coupling and is therefore less flexible. Similarly, Dandelion (Martínez-Lasaca et al. 2023) also creates a view model with text-based model navigation expressions, but focuses more on support for diverse modeling frameworks.

jjodel (Rocco et al. 2023) uses TSX as the technology that developers use to specify the visuals of a graphical editor, similar to the Eclipse GLSP client that uses TSX to create custom SVG elements. In the paper, the authors focus on collaborative modeling. This important feature is implicit for GLSP-based editors as of the client-server architecture, but unlike jjodel, the underlying protocol is standardized.

8. Conclusion

In this paper, we have shown how model synchronization technology can be used for the development of graphical editors, in particular for the heterogeneous model synchronization between a graphical model and a semantic model. We suggest to use the paradigm of model synchronization via synchronization blocks. This paradigm ensures basic properties, like correctness and hippocraticness, and helps to separate the concerns in the development of graphical editors. In order to tailor the model synchronization technology for this task, we suggest to use a dedicated DSL and introduced a candidate. The application of this DSL to a class-diagram-like editor with real-world complexity has shown the flexibility and conciseness that can be achieved using such a DSL.

References

Anjorin, A., Buchmann, T., Westfechtel, B., Diskin, Z., Ko, H., Eramo, R., . . . Zündorf, A. (2020). Benchmarking bidirectional transformations: theory, implementation, application, and assessment. *Softw. Syst. Model.*, *19*(3), 647–691. doi: 10.1007/s10270-019-00752-x

Bork, D., Langer, P., & Ortmayr, T. (2024). A vision for flexible glsp-based web modeling tools. In J. P. A. Almeida, M. Kaczmarek-Heß, A. Koschmider, & H. A. Proper (Eds.), *The practice of enterprise modeling* (pp. 109–124). Cham: Springer Nature Switzerland.

Boronat, A. (2021). Incremental execution of rule-based model transformation. *Int. J. Softw. Tools Technol. Transf.*, *23*(3), 289–311. doi: 10.1007/S10009-020-00583-Y

Buchmann, T., Bank, M., & Westfechtel, B. (2022). Bxtenddsl: A layered framework for bidirectional model transformations combining a declarative and an imperative language. *Journal of Systems and Software*, *189*, 111288. doi: https://doi.org/10.1016/j.jss.2022.111288

Bäumer, D. (2023). *LSP specification 3.17.0.* https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification. ([Online; accessed 2024-11-26])

Calvar, T. L., Jouault, F., Chhel, F., & Clavreul, M. (2019, July). Efficient ATL incremental transformations. *Journal of Object Technology*, *18*(3), 2:1-17. Retrieved from http://www.jot.fm/contents/issue_2019_03/article2.html (The 12th International Conference on Model Transformations) doi: 10.5381/jot.2019.18.3.a2

Corley, J., Syriani, E., & Ergin, H. (2016). Evaluating the cloud architecture of AToMPM. In S. Hammoudi, L. F. Pires, B. Selic, & P. Desfray (Eds.), *MODELSWARD 2016 - proceedings of the 4rd international conference on model-driven engineering and software development, rome, italy, 19-21 february, 2016* (pp. 339–346). SciTePress. doi: 10.5220/0005776903390346

Eclipse, F. (2023). *Sprotty.* https://sprotty.org. ([Online; accessed 2024-11-26])

Eclipse, F. (2024). *Sirius Web.* https://eclipse.dev/sirius/sirius-web.html. ([Online; accessed 2024-11-26])

Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., & Schmitt, A. (2007, May). Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *29*(3). doi: 10.1145/1232420.1232424

Hinkel, G. (2015a). Change Propagation in an Internal Model Transformation Language. In D. Kolovos & M. Wimmer (Eds.), *Theory and Practice of Model Transformations: 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings* (pp. 3–17). Cham: Springer International Publishing. doi: 10.1007/978-3-319-21155-8_1

Hinkel, G. (2015b, 7 24). An NMF Solution to the Java Refactoring Case. In L. Rose, T. Horn, & F. Krikava (Eds.), *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences* (Vol. 1524, pp. 95–99). CEUR-WS.org. Retrieved from http://ceur-ws.org/Vol-1524/paper9.pdf

Hinkel, G. (2017, July 21). An NMF solution to the Smart Grid Case at the TTC 2017. In A. Garcia-Dominguez, G. Hinkel, & F. Krikava (Eds.), *Proceedings of the 10th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2017) federation of conferences.* CEUR-WS.org.

Hinkel, G. (2018a). *Implicit incremental model analyses and transformations* (Doctoral dissertation, Karlsruhe Institute of Technology, Germany). Retrieved from https://publikationen.bibliothek.kit.edu/1000084464

Hinkel, G. (2018b). NMF: A multi-platform modeling framework. In A. Rensink & J. Sánchez Cuadrado (Eds.), *Theory and practice of model transformation* (pp. 184–194). Cham: Springer International Publishing.

Hinkel, G. (2020). An NMF solution to the TTC 2020 roundtrip engineering case. In A. Boronat, A. García-Domínguez, & G. Hinkel (Eds.), *TTC 2020/2021 - Joint Proceedings of the 13th and 14th Tool Transformation Contests. The TTC pandemic proceedings with CEUR-WS co-located with Software Technologies: Applications and Foundations (STAF 2021), Virtual Event, Bergen, Norway, July 17, 2020 and June 25, 2021* (Vol. 3089). CEUR-WS.org. Retrieved from http://ceur-ws.org/Vol-3089/ttc20_paper4_Hinkel.pdf

Hinkel, G. (2021). An NMF solution to the TTC2021 incremental recompilation of laboratory workflows case [ttc]. In A. Boronat, A. García-Domínguez, & G. Hinkel (Eds.), *TTC 2020/2021 - Joint Proceedings of the 13th and 14th Tool Transformation Contests. The TTC pandemic proceedings with CEUR-WS co-located with Software Technologies:*

Applications and Foundations (STAF 2021), Virtual Event, Bergen, Norway, July 17, 2020 and June 25, 2021 (Vol. 3089). CEUR-WS.org. Retrieved from http://ceur-ws.org/Vol-3089/ttc21_paper9_labflow_Hinkel_solution.pdf

Hinkel, G., & Burger, E. (2019). Change propagation and bidirectionality in internal transformation DSLs. *Softw. Syst. Model.*, *18*(1), 249–278. doi: 10.1007/s10270-017-0617-6

Hinkel, G., Goldschmidt, T., Burger, E., & Reussner, R. (2017). Using Internal Domain-Specific Languages to Inherit Tool Support and Modularity for Model Transformations. *Software & Systems Modeling*, 1–27. doi: 10.1007/s10270-017-0578-9

Hinkel, G., Heinrich, R., & Reussner, R. (2019, Jan 29). An extensible approach to implicit incremental model analyses. *Software & Systems Modeling*. doi: 10.1007/s10270-019-00719-y

JSON-RPC Working Group. (2013). *JSON-RPC 2.0 Specification.* https://www.jsonrpc.org/specification. ([Online; accessed 2024-11-26])

Juarez, S. (2016). *Anders Hejlsberg on Modern Compiler Construction.* https://learn.microsoft.com/en-us/shows/seth-juarez/anders-hejlsberg-on-modern-compiler-construction. ([Online; accessed 2024-11-26])

Knopfel, A., Grone, B., & Tabeling, P. (2006). *Fundamental Modeling Concepts: Effective communication of IT systems*. Hoboken, New Jersey, USA: Wiley.

Kolovos, D. S., de la Vega, A., & Cooper, J. C. (2020). Efficient generation of graphical model views via lazy model-to-text transformation. In E. Syriani, H. A. Sahraoui, J. de Lara, & S. Abrahão (Eds.), *Models '20: ACM/IEEE 23rd international conference on model driven engineering languages and systems, virtual event, canada, 18-23 october, 2020* (pp. 12–23). ACM. doi: 10.1145/3365438.3410943

Kolovos, D. S., Rose, L. M., bin Abid, S., Paige, R. F., Polack, F. A. C., & Botterweck, G. (2010). Taming EMF and GMF using model transformation. In D. C. Petriu, N. Rouquette, & Ø. Haugen (Eds.), *Model driven engineering languages and systems - 13th international conference, MODELS 2010, oslo, norway, october 3-8, 2010, proceedings, part I* (Vol. 6394, pp. 211–225). Springer. doi: 10.1007/978-3-642-16145-2_15

Martínez-Lasaca, F., Díez, P., Guerra, E., & de Lara, J. (2023). Engineering low-code modelling environments with dandelion. In *ACM/IEEE international conference on model driven engineering languages and systems, MODELS 2023 companion, västerås, sweden, october 1-6, 2023* (pp. 14–18). IEEE. doi: 10.1109/MODELS-C59198.2023.00011

Metin, H., & Bork, D. (2023). On Developing and Operating GLSP-based Web Modeling Tools: Lessons Learned from BIGUML. In *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)* (p. 129-139). doi: 10.1109/MODELS58315.2023.00031

Microsoft. (2007). *The .NET Standard Query Operators.* http://msdn.microsoft.com/en-us/library/bb394939.aspx. ([Online; accessed 2024-11-26])

Microsoft. (2024a). *LSP Language Servers.* https://microsoft.github.io/language-server-protocol/implementors/servers. ([Online; accessed 2024-11-26])

Microsoft. (2024b). *Microsoft Automatic Graph Layout.* https://github.com/microsoft/automatic-graph-layout. ([Online; accessed 2024-11-26])

Microsoft. (2024c). *Monaco Editor.* https://microsoft.github.io/monaco-editor/. ([Online; accessed 2024-11-26])

Ráth, I., Ökrös, A., & Varró, D. (2010). Synchronization of abstract and concrete syntax in domain-specific modeling languages - by mapping models and live transformations. *Softw. Syst. Model.*, *9*(4), 453–471. doi: 10.1007/S10270-009-0122-7

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

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

Rubel, D., Wren, J., & Clayberg, E. (2011). *The eclipse graphical editing framework (gef).* Addison-Wesley Professional.

Sugiyama, K., Tagawa, S., & Toda, M. (1981). Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics*, *11*(2), 109-125. doi: 10.1109/TSMC.1981.4308636

Yohannis, A. R., Kolovos, D. S., & García-Domínguez, A. (2024). Exploring complex models with picto web. *Sci. Comput. Program.*, *232*, 103037. doi: 10.1016/J.SCICO.2023.103037

Zweihoff, P., Naujokat, S., & Steffen, B. (2019). Pyro: Generating domain-specific collaborative online modeling environments. In R. Hähnle & W. M. P. van der Aalst (Eds.), *Fundamental approaches to software engineering - 22nd international conference, FASE 2019, held as part of the european joint conferences on theory and practice of software, ETAPS 2019, prague, czech republic, april 6-11, 2019, proceedings* (Vol. 11424, pp. 101–115). Springer. doi: 10.1007/978-3-030-16722-6_6

About the authors

Georg Hinkel is professor at the RheinMain University of Applied Sciences (Germany) and maintainer of NMF. His research focuses on MDE topics, incremental change propagation and laboratory automation You can contact the author at georg.hinkel@hs-rm.de.

Bodo Igler is professor at the RheinMain University of Applied Sciences (Germany). His research focuses on the application of formal methods for software quality and DSLs embeddings.You can contact the author at bodo.igler@hs-rm.de.