A General Architecture for Client-Agnostic Hybrid Model Editors as a Service

Liam Walsh
liam.walsh@queensu.ca
Queen's University
Canada

Juergen Dingel
dingel@queensu.ca
Queen's University
Canada

Karim Jahed
jahed@cs.queensu.ca
Queen's University
Canada

ABSTRACT

In this paper, we propose a general architecture for designing language servers for hybrid modeling languages, that is, modeling languages that contain both textual and graphical representations. The architecture consists of a textual language server, a graphical language server, and a client that communicates with the two servers. The servers are implemented using the Language Server Protocol (LSP) and the Graphical Language Server Protocol (GLSP) and are based on a shared abstract syntax of the hybrid language. This means that only static resources need to be common between the graphical and textual language servers. The servers' separation allows each to be developed and maintained independently, while also enabling forward-compatibility with their respective dependencies.

We describe a prototype implementation of our architecture in the form of a hybrid editor for the UML-RT language. The evaluation of the architecture via this prototype gives us useful insight into further generalization of the architecture and the way it is used. We then sketch a suitable extension of the architecture to enable support for multiple diagram types and, thus, multiple graphical views.

CCS CONCEPTS

• **Software and its engineering** → **Software design engineering**; **Domain specific languages**.

KEYWORDS

Hybrid, LSP, UML-RT, Graphical Modeling, Language Server

1 INTRODUCTION

With the increasing amount of complexity in software systems, it is important to be able to express them clearly and consistently. This is a must to ensure effective communication among possibly diverse sets of stakeholders such as designers, developers, and users. Abstracting away implementation-level details is one way to provide clarity to a user. Consistency can be achieved by a definition of rules and patterns. This type of approach to expressing systems allows them to be easily understood by way of familiarity with their domain, rather than programmatic knowledge.

The high-level motivation of our work is to enable better ways to express models of software as used in Model-Driven Development (MDD). The ultimate goal is to facilitate the development of hybrid editors for modeling languages.

We propose a methodology of creating two separate services to equip Integrated Development Environments (IDEs) with the capabilities of hybrid editing. This involves the implementation of two concrete syntaxes, one textual and the other graphical, each providing suitable representations (or views) of the same underlying model and its abstract syntax. While this methodology seems straightforward conceptually, it is challenging to implement in a way that supports cohesion, maintainability and evolvability.

2 BACKGROUND

2.1 Hybrid Modeling Languages

Hybrid modeling languages are characterized by their use of more than one concrete syntax paradigm. In this paper, we will use the term to refer specifically to a modeling language which has both a textual concrete syntax, and a graphical concrete syntax.

2.2 Language Servers

The classical method of implementing support for a language in an IDE usually results in high coupling to the IDE, even though most modern IDEs are functionally similar. Language servers take advantage of these similarities by treating IDEs as clients to be serviced and whose behaviour is dictated by the language's semantics.

2.2.1 Language Server Protocol. The Language Server Protocol allows a client IDE to trade information and instructions with a language server. The language server can run as a background process and be queried by the IDE in order to give the user advanced editing features such as "auto complete", "go to definition/declaration", "find all references", among others [17]. LSP allows for the client IDE to be completely language-agnostic.

2.2.2 Graphical Language Server Protocol (GLSP). The Graphical Language Server Protocol [3, 5], is an open source framework originally built to facilitate displaying SVG graphs and providing editing tools for graphical languages. This is all done in such a way that one implementation would be able to service multiple IDEs compliant with GLSP, hence the similarity in names between it and LSP.

GLSP functions by visiting a model, similar to how any other modeling tool would parse it, and systematically creating graph

objects that reflect each different object in the original model specified by the developer. GLSP also allows the language developer to choose where to persist any elements of the concrete syntax that do not impact the execution semantics of the model (i.e. graph X/Y positioning, colors, shapes, etc.).

2.3 UML for Real-Time

The Unified Modeling Language for Real-Time (UML-RT) is a modeling language that is specified as a UML Profile. The language was created with the properties necessary to be able to describe Real-Time systems [11, 16]. UML-RT's established graphical specifications made it a good candidate for our prototype hybrid editor.

The two main components of a system's architecture are its *structural* and *behavioral* aspects. UML-RT is able to create graphical models of systems that are capable of specifying both these structural and behavioral aspects. Some tools for specifying UML-RT models, such as the open-source eclipse-based Papyrus-RT [7], are able to automatically generate functional code from a given model.

RTPoet is a toolset that we have created for the purposes of working with UML-RT models of different forms [12]. The original use case was to bridge the gap between different UML-RT implementations (specifically Papyrus-RT [7] and RTIST/RSARTE [8, 9]) by model transformation. We have already specified a textual implementation of UML-RT under the RTPoet umbrella. Once the UML-RT hybrid language server prototype is sufficiently mature, we intend to integrate it into the RTPoet suite.

3 PROPOSED ARCHITECTURE

In this section, we identify some key requirements that the architecture must satisfy. Then, the structure of the architecture is described together with sequence diagrams showing the interaction between components to realize aspects of these requirements.

The use of any language in an IDE must include the support of the standard CRUD (create, read, update, delete) operations. However, these are not a necessary point of focus for our proof-of-concept implementation, as these operations are mostly realized by the language definitions contained in the servers themselves. We are more interested in the design of a high-level architecture.

Current best practices in GUI development include the use of the Model-View-Controller (MVC) design pattern. GLSP's design is informed by this pattern, as graphs naturally suggest the use of GUIs. We have chosen to model the textual language server in a similar way. While MVC is not normally applied to text editors, its properties allow us to maintain synchronization between views. In the context of language servers, most of the responsibilities of the "controller" and "view" are carried out by the client, language-specific tasks are delegated to the server.

3.1 Four Core Interface Requirements

Figure 1 shows our proposed general approach to creating the architecture for hybrid language services. There are four main interfaces that must be implemented in order to fulfill the CRUD use cases in each of the textual and graphical views while keeping them synchronized. These four interfaces are denoted by the edges in the figure, and each represent a transformation to, or from, the views and model. Essentially, the model is the abstract syntax,
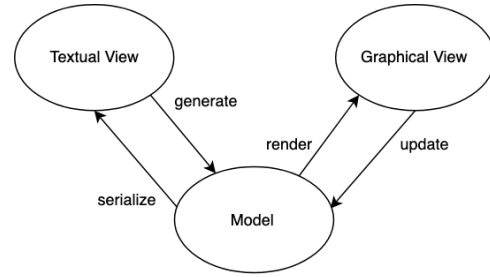

Figure 1: High-Level Relationship Between Graphical and Textual Views

while the graphical and textual views are two different types of concrete syntaxes used to specify and represent it. The four different interfaces will be discussed in more detail below.

3.1.1 Generation. We will use "generation" or "regeneration" to refer to the transformation of the textual view into the model. This is the intended usage of Xtext [4], and will not be any different than the first step of implementing a textual DSL (Domain-Specific Language) using the tool. The realization of this interface requires the specification of a grammar for the language. The textual language server uses the grammar to instruct the language client how to perform this transformation.

3.1.2 Serialization. The model that the textual view generates retains enough information about the original specification that it is still possible to perform a transformation from the model itself back to the textual view. This process is the inverse to the previously explained one, and the client is advised on how to do so by the textual language server.

3.1.3 Rendering. The "rendering" of the graphical view should be realized by specifying a mapping between model objects and graph element objects (predominantly different types of nodes and edges). More granular parts of the language may be left out of this mapping, if they can not be appropriately graphed and can be more effectively handled by the textual view (e.g., action code in state machines).

3.1.4 Updating. Graphical views typically only display a subset of the elements of the model they represent. Lower-level properties may be simplified or excluded altogether, meaning we cannot reliably retrieve these by transforming a graphical view into a model. We can, however, manipulate the model directly by keeping track of the graph elements' source mappings. GLSP allows for this to be done natively. We will refer to this as "updating" the model, as we are applying an update directly to the model objects by proxy of the graphical elements. Models can be kept complete and correct by restricting the operations available in the graphical view (often by using "palettes of operations"). Each time an update is applied, the graphical view must then be rendered anew using the updated model.

3.2 Architecture Diagram

Figure 2 depicts our proposed general architecture for hybrid language servers. The diagram shows the communication between two

language clients, encapsulated together in the "hybrid language client", and two language servers, together in the "hybrid language server". The two pairs of clients and servers communicate on different ports as, despite their similarities, they make use of different protocols.

The previously described "generation" and "serialization" occur between the "textual client" and the "model" shown in figure 2. Though, the methods that the client uses to do so are given to it by the textual language server. The same is true with respect to the graphical side: "rendering" and "updating" occur between the "graphical client" and "model", and the information on how to carry out these steps is held by the graphical language server.
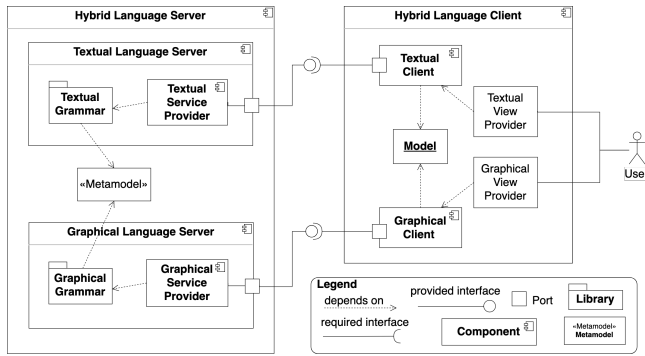

Figure 2: Architecture of Hybrid Language Server

3.3 Sequence Diagrams

In order to better illustrate the interaction between the modules in the proposed general architecture, some sequence diagrams depicting different behaviours have been included below.
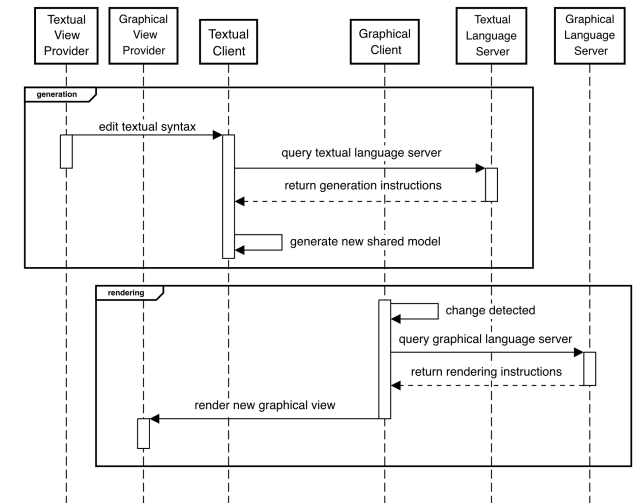

Figure 3: Sequence Diagram: Textual Edit Action

The sequence diagram in figure 3 shows how a change to the textual view would be propagated to the graphical view.
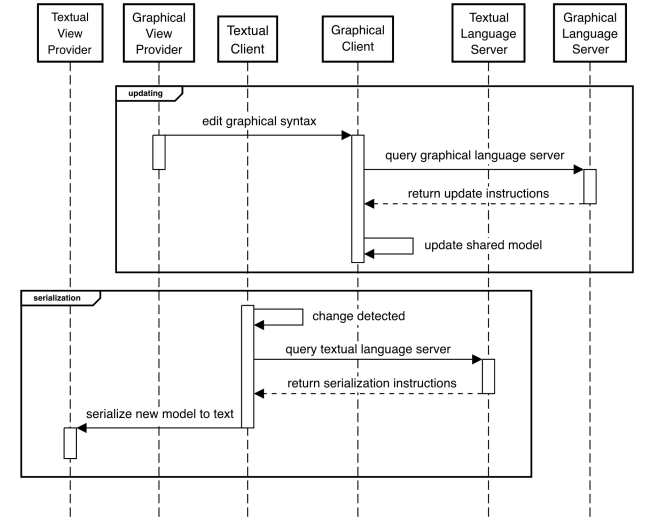

Figure 4: Sequence Diagram: Graphical Edit Action

The sequence diagram in figure 4 shows how a change to the graphical view would affect the textual view. By simply observing the shape of this sequence compared to the previous one, it is clear that they are very similar.

4 PROTOTYPE IMPLEMENTATION

In order to evaluate our proposed architecture, we have created a prototype hybrid language server, which we will refer to as the "RTPoet language", based on UML-RT. Below is a description of our implementation of the four core interfaces described in section 3.1. This is followed by a discussion of our findings.

4.1 Generation

To implement the generation interface, a textual grammar for the RTPoet language has been specified using Xtext.

While most Xtext-based languages often define code generation instructions in addition to a grammar, the code stubs included when creating a new Xtext project are sufficient to allow model files to be generated from the textual view. This is possible because the model file is purely structural.

We have made our Xtext implementation of UML-RT (i.e., the RTPoet language's textual syntax) available as part of the RTPoet tool suite, available on GitHub [13].

4.2 Serialization

In order to be able to serialize a model back into the textual view, specific mappings to code templates must be specified for each of the model objects. Whereas the aforementioned generation process would typically yield "markup language" string representations of objects as their target (such as EMF/XML models), this type of transformation would need to yield complete and correct "code". The templates can consist of predefined code blocks with different types of values and identifiers substituted into the appropriate places. A traversal of the model's abstract syntax tree allows the string templates to be correctly ordered. This is similar to how an

Xtext language might normally be equipped to generate code, but it easier to implement considering we already have access to the target language's grammar and metamodel.

4.3 Rendering

The metamodel resulting from the specification of the Xtext grammar was used to automatically create a full Java class model for the RTPoet language. The structure of these classes exactly reflects the scope of models specifiable in the textual syntax. This collection of Java classes was used as a basis for the implementation of the graphical language server. The collection's format is identical to the metamodel. The next step was to map each metamodel type to an appropriate GLSP graphical type. GLSP contains its own classes for graph elements, and assigns them SVG properties at runtime.

Next, a model visitor was created, based on one of GLSP's extension points, that converts any given model into a collection of graphical objects based on the mappings. The graph resulting from our rendering operation is held directly in memory. The source mappings are kept track of by GLSP. Since we have opted to not store any additional graphical properties, we chose to implement automatic arrangement of the elements.

4.4 Updating

Some basic update instructions were added to the graphical language server, such as the ability to add a new simple state to a state machine. However, the implementation of the prototype was halted here, as there were architectural incompatibilities that needed to be addressed. This will be discussed in detail below.

5 NEW PROPOSED ARCHITECTURE

We found that, given our current approach, the functionality of our implementation of UML-RT was not up to par with existing editors. In order to implement hybrid languages in a more convincing, user-friendly way, single views are insufficient. In particular, single views offer poor support for the following modeling concepts and activities.

5.1 Containment

A concept that is present in many modeling languages is that of object containment. If an object owns a collection of other objects, it may make sense to render the collection of objects within the boundaries of their owner. However, as object ownership depth increases, this becomes harder to manage. A better solution is to render only objects with a direct containment relation. This would mean that in order to interact with elements of greater depth, we would need the ability to drill down into elements using new views.

5.2 Projection

Models often contain many elements, complicating their user-friendly graphical display. Projection, that is, the selective display of only specific subsets of model elements, can mitigate this problem.

These projections can be based on language concepts, such as types or other fixed properties, and they can be further influenced by user input. For instance, some elements in a UML-RT model represent structure, while some other represent behaviour (as seen in Figure 5). While displaying different projections simultaneously

in a single view can be useful, it can also cause confusion, and support for different views seems preferable in general.

5.3 Static Analysis

A static analysis often involves using the model as an input and, after some additional processing, returning an output whose syntax and semantics may well be outside the scope of the modeling language itself. For example, we might want to see a full traversal of a state machine. This would include duplicate states, something not possible using only the vanilla model view. Therefore, rendering graphs for these types of operations is intrinsically incompatible with a single model visualization.
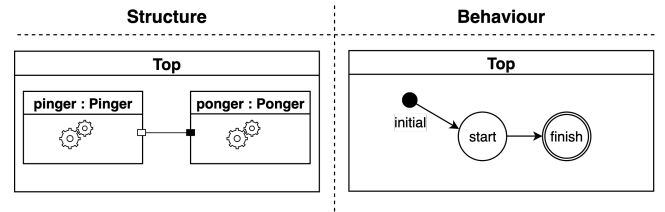
Figure 5: Structure vs Behaviour in UML-RT Model

In summary, an architecture for hybrid language servers should enable support for different views. Only then can different parts of the model, as well as results of different projections and analyses, be displayed most appropriately.

By using multiple different graphical views, we may represent languages' structure more accurately while also providing an extension point for supplemental graph functionality.

5.4 Modified Architecture

Our architecture can be modified to support multiple views. This modification is minimally invasive and, from the point of view of the client, it is the same as before. The new proposed architecture can be seen in figure 6.
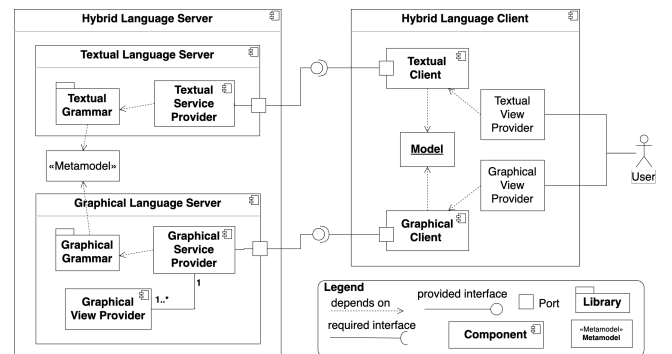
Figure 6: New Architecture of Hybrid Language Server

5.4.1 Switching Views. A short sequence diagram depicting the switching of graphical views can be seen in figure 7. The initial action could come from simply clicking on an element. In this case,

the client would send the click with a reference to the clicked element along with necessary metadata. We can also see that the latter half of the sequence is the same as the previous graph rendering sequence in figure 3. Unless we explore the local scope of the Graphical Language Server, the process will not appear much different.
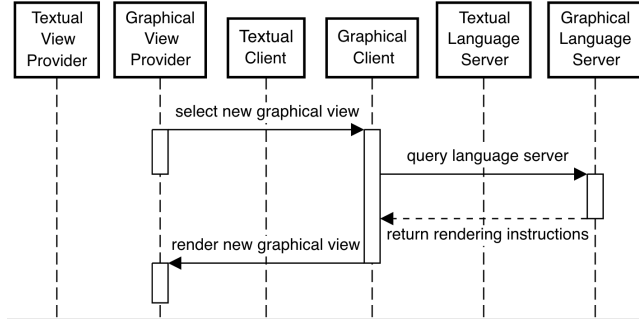

Figure 7: Sequence: Graphical View Switching

6 RELATED WORK

Passing Graphical Information Using LSP. Papers [14] and [15] discuss the proposal and implementation of a custom graphical editing solution for EMF languages, using LSP without any extensions. Their solution involves converting the EMF model into a JSON-based format in order to exchange it with the server across LSP. The JSON-based equivalent of the whole EMF model is what is transmitted, including pure graph properties to render the graph as an SVG (e.g. `width:100`, `height:50`, `shape:square`). The usage of JSON-based messages makes this effectively an implementation of a textual language server. They also define actions that the frontend client is able to perform and manually map them to specific LSP messages, such as creating or removing edges or nodes. In the earlier of the their two papers, they propose the solution as completely compliant with LSP as a prototype. In the later paper, they discuss the possibility of extend LSP or defining a brand new protocol, but ultimately decide to remain compliant with LSP. Since our work separates the two editing mediums' implementations, we will not need to evaluate problems such as protocol extensions.

UML Profile Hybrid Editors. Addazi et al. [1] propose a methodology of implementing hybrid languages based on UML by extending it (i.e. creating UML profiles). In concept, their approach is similar to ours, as it makes use of a shared abstract syntax between the two different views of a given language. Similar to our proposed architecture, the source of their shared metamodel is based on Xtext's grammar specifications. The authors use UML as a basis for their specification of Xtext grammars, allowing them to leverage its existing graphing libraries. Our approach does not rely on UML, but it could be extended in the same way (i.e. used as a basis for typing in the Xtext grammar specification). However, reusing any of the graphical editor implementation designs in this work is not possible in the context of language servers. The UML graphing libraries' dependence on IDE-specific features render them incompatible with our proposed architecture both in concept and in practice. That being said, the stability of UML could motivate the creation of similar

UML-based graphing libraries that operate within the boundaries of graphical language servers.

bigER Modeling Language. Glaser et al. have created the "bigER" tool [6] which, as of yet, seems to be the most closely related to our own work. The bigER tool is an example of a functional hybrid language server that also makes use of Xtext. The approach to the design of the bigER tool mirrors one of our prior proposed architectures for hybrid language servers [19, 20]. At a high level: graphical operations are mapped to code snippets which are injected directly into to the textual model. This is similar to the code templating required for defining the "serialization" interface, but controlled by a palette of operations similar to that required by the "updating" interface (as seen in figure 1). The textual model can then regenerate the graphical model, displaying the changes graphically. We did not pursue this style of architecture for several reasons. Most notably, this solution to synchronization of the two editors is more difficult to specify and maintain as the size of the language increases. For more complex languages, static analysis might need to be done on the code in order properly contextualize the code snippets. Our current proposed architecture uses the language's structure to guarantee completeness and correctness of graphical edits. Language specification errors in our implementation can more easily be detected at compile time, whereas errors using this style of implementation may only be clear at runtime.

Graphical Viewspoints. In [2], domain-specific modeling languages with multiple "viewpoints" are discussed. The work describes the nuances of the relationship between model and graphs. The authors present a methodology for designing multi-view DSVLs (Domain-Specific Visual Languages) in which they note that models can be projected into submodels in different "viewpoints" (which are analogous to the "views" described in this work). The concept of viewpoints closely mirrors our proposed implementation of views. While our work was not inspired by this paper, it corroborates our findings well. The authors' work predates LSP, meaning that it evidently does not make use of it. This is a key differentiating factor relative to our work.

7 CONCLUSION AND FUTURE WORK

We have proposed and evaluated a methodology for designing and implementing hybrid language servers, in the form of a general architecture, that offers a cohesive approach, simplifies maintainability, and facilitates language evolution.

We anticipate that there is much work to be done in the study of hybrid language design and usability. This may be inspired by existing research on best practices for textual and graphical mediums alike [10, 18]. We hope that our findings may be foundational to the accessibility of hybrid modeling languages, such that we may facilitate future work in studying them as a whole, and not only relative to language servers.

REFERENCES
[1] Lorenzo Addazi, Federico Ciccozzi, Philip Langer, and Ernesto Posse. 2017. Towards Seamless Hybrid Graphical–Textual Modelling for UML and Profiles. 20–33. https://doi.org/10.1007/978-3-319-61482-3_2
[2] Francisco Andrés, Juan Lara, and Esther Guerra. 2007. Domain Specific Languages with Graphical and Textual Views, Vol. 5088. 82–97. https://doi.org/10.1007/978-3-540-89020-1_7

[3] Eclipsesource. 2022. Graphical Language Server Protocol. https://github.com/eclipse-glsp/glsp

[4] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. ACM, New York, NY, USA, 307–309. https://doi.org/10.1145/1869542.1869625

[5] Luca Forstner. 2022. Integrating GLSP based Tooling into Visual Studio Code. Bachelor Thesis.

[6] Philipp-Lorenz Glaser and Dominik Bork. 2021. The bigER Tool - Hybrid Textual and Graphical Modeling of Entity Relationships in VS Code. In *2021 IEEE 25th International Enterprise Distributed Object Computing Workshop (EDOCW)*. 337–340. https://doi.org/10.1109/EDOCW52865.2021.00066

[7] Christopher Guindon. 2015. Papyrus-RT - Overview: The Eclipse Foundation. https://www.eclipse.org/papyrus-rt/content/overview.php

[8] Mattias Mohlin. 2018. Modeling Real-Time Applications in RTist. https://rtist.hcldoc.com/help/topic/com.ibm.xtools.rsarte.webdoc/pdf/RTist%20Concepts.pdf

[9] Mattias Mohlin. 2020. Modeling Real-Time Applications in RSARTE. https://rsarte.hcldoc.com/help/topic/com.ibm.xtools.rsarte.webdoc/pdf/RSARTE%20Concepts.pdf

[10] Daniel Moody. 2009. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering* 35, 6 (2009), 756–779. https://doi.org/10.1109/TSE.2009.67

[11] Ernesto Posse and Juergen Dingel. 2016. An executable formal semantics for UML-RT. *Software & Systems Modeling* 15, 1 (2016), 179–217.

[12] Queen's University MASE Lab. 2022. RTPoet. https://github.com/qumase/rtpoet

[13] Queen's University MASE Lab. 2022. RTPoet DSL. https://github.com/qumase/rtpoet-dsl

[14] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. 2018. An LSP infrastructure to build EMF language servers for web-deployable model editors. In *MODELS Workshops*.

[15] Roberto Rodríguez-Echeverría, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. 2018. Towards a Language Server Protocol Infrastructure for Graphical Modeling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (Copenhagen, Denmark) *(MODELS '18)*. ACM, New York, NY, USA, 370–380. https://doi.org/10.1145/3239372.3239383

[16] B. Selic. 2002. The real-time UML standard: definition and application. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*. 770–772. https://doi.org/10.1109/DATE.2002.998385

[17] Sourcegraph. 2022. Langserver.org A community-driven source of knowledge for Language Server Protocol implementations. https://langserver.org/

[18] Edward R Tufte, Susan R McKay, Wolfgang Christian, and James R Matey. 1998. Visual explanations: Images and quantities, evidence and narrative.

[19] Liam Walsh. 2020. Slide Deck: Toward Client-Agnostic Hybrid Model Editor Tools as a Service. https://msdl.uantwerpen.be/conferences/MPM4CPS/2020/wp-content/uploads/2020/11/Liam_Walsh_MPM4CPS_2020.pdf

[20] Liam Walsh, Juergen Dingel, and Karim Jahed. 2020. Toward client-agnostic hybrid model editor tools as a service. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 1–1. https://doi.org/10.1145/3417990.3421440