Catchword: Language Server Protocol

An Introduction to the Protocol, its Use, and Adoption for Web Modeling Tools

Dominik Bork*,[a], Philip Langer[b]

[a] TU Wien, Business Informatics Group, Vienna, Austria
[b] EclipseSource, Vienna, Austria

Abstract. *With the introduction of the **Language Server Protocol (LSP)**, a fundamental shift has been observed in the development of language editing support for Integrated Development Environments (IDEs), such as VS Code, the traditional Eclipse IDE, or Eclipse Theia. LSP establishes a uniform protocol that standardizes the communication between a language client (e. g., an IDE like Eclipse) and a language server (e. g., for a programming language like Java). The language client only needs to be able to interpret and understand the protocol instead of the specific programming language. Likewise, the language server can focus on language support and does not need to consider the specifics of a respective IDE. This reduces the complexity of realizing language support on different editors and IDEs and enables smooth transitions from one IDE to another. LSP is an open and community-driven protocol that has been developed within the realm of the VS Code community, initiated and driven by Microsoft. The generic concept and architectural pattern of LSP enables widespread applications that go far beyond the realization of editing support for programming languages. This paper provides an introduction to LSP, describes its evolution and core characteristics, and delineates its potential for revolutionizing not only the IDE market but also other software systems, such as modeling tools.*

Keywords. Language Server Protocol • Integrated Development Environment • Graphical Language Server Protocol • Conceptual Modeling • Modeling tools • Software Engineering

Communicated by Peter Fettke. Received 2023-05-23. Accepted after 1 revision on 2023-08-16.

1 Introduction

Traditionally, language editing support, such as code completion and diagnostics for a specific language, had to be implemented for each Integrated Development Environment (IDE) individually. This caused an additional burden to language developers, as they had to get familiar with the extension API of multiple IDEs and integrate their language again and again for each IDE they wanted to support. Furthermore, language developers were required to use the programming language imposed by the respective IDE instead of having the freedom to use any programming language for implementing their language support. As a result, the quality of the language support for specific languages varied significantly across different IDEs and eventually often harmed usability for software developers using the respective language in a specific IDE. Whereas such an IDE-based specialization enables the optimization of the language support by facilitating the specific capabilities offered by a particular IDE, this approach, however, results in redundant development efforts and a lack of interoperability across IDEs.

* Corresponding author.
E-mail. dominik.bork@tuwien.ac.at
The authors like to thank Erich Gamma and Dirk Bäumer from Microsoft for providing valuable feedback on earlier drafts of this manuscript. Their evaluation ensures that LSP and VS Code's historical development is correctly described.

For each programming language L and each IDE I, $L \times I$ language integrations were needed.

With the introduction of the Language Server Protocol (LSP) this issue is significantly mitigated (Gunasinghe and Marcus 2022) by establishing a uniform protocol that standardizes the communication between a language client (e. g., an IDE like Eclipse) and a language server (e. g., for a programming language like Java). The language client only needs to be able to interpret and understand the protocol instead of the specific programming language. Likewise, the language server can focus on language support but doesn't need to consider the specifics of the respective IDE. This reduces the complexity of realizing language support on different IDEs from the beforementioned $L \times I$ to $L + I$ and it furthermore enables smooth transitions from one IDE to another without a change in the quality of the language support. LSP is an open and community-driven protocol that has been developed within the realm of the VS Code community[1] , initiated and driven by Microsoft (Microsoft 2022a), and has evolved into a de-facto standard in the IDE market.

LSP also introduces more flexibility for developers of language servers or language clients by providing *"the freedom to choose the most suitable technology to implement the client editor and the language server independently of each other"* (Rodríguez-Echeverría et al. 2018, p. 371). Moreover, LSP clearly separates the concerns of the language server from those of the language client. While the former is responsible for implementing the *language smarts* (e. g., creating and processing an Abstract Syntax Tree for programming language support), the language client is responsible for *editor smarts* like rendering (e. g., code highlighting), editing, and user interaction.

With the development of LSP, Microsoft has fundamentally disrupted the IDE market and provides VS Code one of the most popular code editors, which is used as an IDE by many developers worldwide[2] . Notably, the generic concept and architectural pattern of LSP enable widespread applications that go far beyond the realization of editing support for programming languages.

This paper provides a coherent introduction to the Language Server Protocol, describes its evolution and core characteristics, and delineates its potential for revolutionizing not only the IDE market but also other software systems, such as modeling tools (Glaser et al. 2022; Rodríguez-Echeverría et al. 2018). In doing so, we *i*) review the historical developments that led to the LSP (Sect. 2), *ii*) describe the specific characteristics of the LSP (Sect. 3), *iii*) showcase how language servers can be developed (Sect. 4), *iv*) report on how LSP-based modeling tools can be realized (Sect. 5), *v*) elaborate future research directions alongside LSP (Sect. 6), and eventually conclude the paper in Sect. 7.

2 Evolution of the Language Server Protocol

The evolution of the Language Server Protocol is heavily linked to the development of VS Code. LSP is today one of the fundamental drivers behind the success of VS Code and its adoption by a very active community of software developers using it and contributing to its further development by means of bug reports, feature requests, and pull requests[3] . In the following, we thus investigate briefly the evolution of LSP in relation to the VS Code code editor.

The rationale and motivation behind building VS Code were to marry the benefits of, at that time, mostly isolated worlds of powerful text editors, which are lightweight, versatile, and fast with the rich functionality offered by powerful IDEs, including sophisticated language support, autocompletion, code navigation, diagnostics, and debugging capabilities. This endeavor should result

[1] LSP GitHub repository: https://github.com/microsoft/language-server-protocol, last accessed: 12.08.2023

[2] https://pypl.github.io/IDE.html last accessed: 12.08.2023

[3] The VS Code Github repository currently lists more than 1,800 contributors: https://github.com/microsoft/vscode/ last accessed: 12.08.2023

in a multi-language, cross-platform, lightweight code editor based on a modern web technology stack, offering rich language support and features across the entire software development process with the help of a broad extension ecosystem provided by the community and software vendors.

Some of the development of VS Code was influenced by the experience of developing the Eclipse platform (Gamma and Beck 2004). Eclipse had—and has until today—a clear extension model, where all extensions run in the same process. While this architecture gives a lot of power to extensions, the Eclipse platform became vulnerable to side effects caused by extensions (e. g., delay in start-up, breaking core functionality, etc.). Also, the Eclipse extension model implied that all extensions had to be written in Java, which hampers realizing language support for different languages, as they are often written in the language for which they provide language support.

This experience led to the design decision of realizing an extension model that safeguards the core platform from misbehaving extensions. From the opposite perspective, this also made sure that changes to the core platform by platform developers did not impact extensions. Therefore, the architecture of VS Code has been designed such that any extension in VS Code runs in a separate process and the communication between extensions and the core platform is only possible via stable and controlled API gateways. This makes work simpler for both platform and extension developers because there is a single API to maintain or use, respectively. Of course, this comes with a cost: extension developers need to realize their functionality within the strict limits of the API imposed by the core platform.

First aims in releasing a Web IDE, under the name *Visual Studio Online Monaco*, did not yield significant numbers of users, but it brought the browser-based *Monaco editor*[4] to live. To reach a broader adoption, it was decided to pivot to creating a desktop editor that, however, still leverages modern browser technologies, including the Monaco editor, Atom Shell, or later Electron. This shift eventually led to the release of the VS Code editor in 2015. Only in 2020, the circle back to the web was concluded with the release of GitHub Codespaces, a cloud-based development environment that uses VS Code as a code editor entirely running in the browser.

The architecture for extending VS Code with language support for a specific programming language was heavily influenced by the motivation to enable developers to use their language of choice for developing the language smarts of their languages. Thus, the component implementing the language smarts was externalized into a separate process, and communication between the external language component (language server) and the text editor (language client) is only possible via a well-defined protocol – the Language Server Protocol (LSP). LSP was first released in 2015 and already supported, among others the *hover*, *completion*, and *signature help* language smarts, and a *capability model*. It soon became clear that making LSP an open protocol that is even independent of VS Code would yield benefits for language providers and editor providers alike. An open LSP would ease the way to use one language server in arbitrary editors and language providers would avoid depending on a proprietary protocol with their language server and, due to the availability of multiple editor clients, obtain a larger user base. Consequently, the work on developing a generic and open protocol standardizing the communication between a language client and a language server started. In 2016, Microsoft started to collaborate with RedHat and CodeEnvy to create an open protocol for language services. The outcome of that effort was announced at the DevNation conference. The stable maintained version (v3.0.0) of the protocol was released in February 2017. Since then, the protocol is under continuous revision and

[4] https://microsoft.github.io/monaco-editor last accessed: 12.08.2023

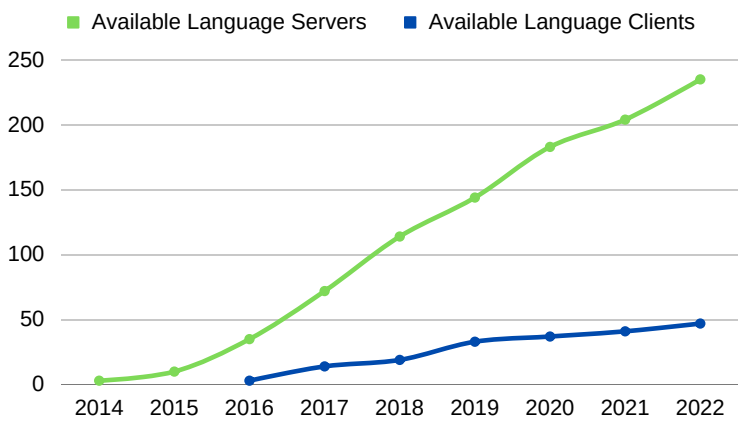

Figure 1: Longitudinal analysis of the available language servers and clients

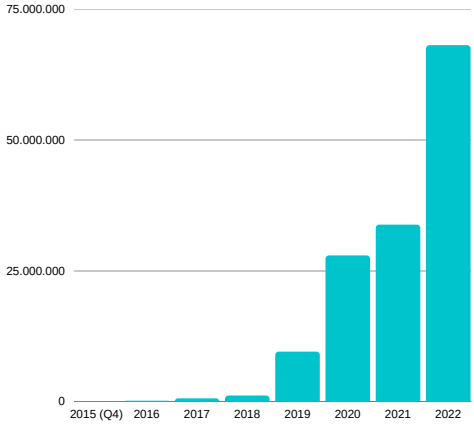

Figure 2: Annual downloads of the vscode-languageserver package in npm[5]

extension by the community[5] . In May 2022, the latest major release, v3.17.0, has been announced.

The continuous growth of available language servers and language clients (see Figure 1 left) shows the disruptive impact of LSP on the software development community.Figure 2 further shows the annual downloads of the VS Code LSP package via npm[6] . Including e.g., CI/CD pipelines, the package has been downloaded via npm 141,253,128 times between October 2015 and December 2022. The figure shows the increasing interest in LSP by the continuous growth of recorded annual downloads. Alone in 2022, it was downloaded more than 68 million times.

3 Language Server Protocol Characteristics

The underlying idea of LSP is to encapsulate the language smarts of a specific programming language, which involves parsing code, analyzing its abstract syntax tree, resolving references, diagnosing errors, etc., into a separate component called a *language server*. With this encapsulation into a separate component, the heavy lifting of providing deep language support for a programming language (language server) is decoupled from the lightweight rendering and direct user interaction in a specific code editor (language client). A language-agnostic client communicates with a language server through JSON-RPC using the *language server protocol* to integrate language-specific editing support for which a deep understanding of the respective programming language is needed. This separation enables the language client to remain utterly agnostic of the programming language. Moreover, the processes in which the language client and the language server are executed can be separated.

3.1 Separation of Concerns

The language server protocol applies an abstraction level centered around a code editor's user interface and defines protocol messages about text documents, character positions, diagnostics, etc. The protocol does not expose information about the intrinsic concepts of a programming language such as the abstract syntax tree or whether a symbol is a class or a method call. Instead, it provides information empowering a client to perform standard features of a code editor for which deep language smarts and analysis are required such as auto-completion, showing syntax errors, or navigating to the definition of a symbol at a specific location. Similarly, the language server protocol

[5] https://github.com/microsoft/language-server-protocol last accessed: 12.08.2023

[6] https://npm-stat.com/charts.html?package=vscode-languageserver&from=2015-10-01&to=2022-12-31

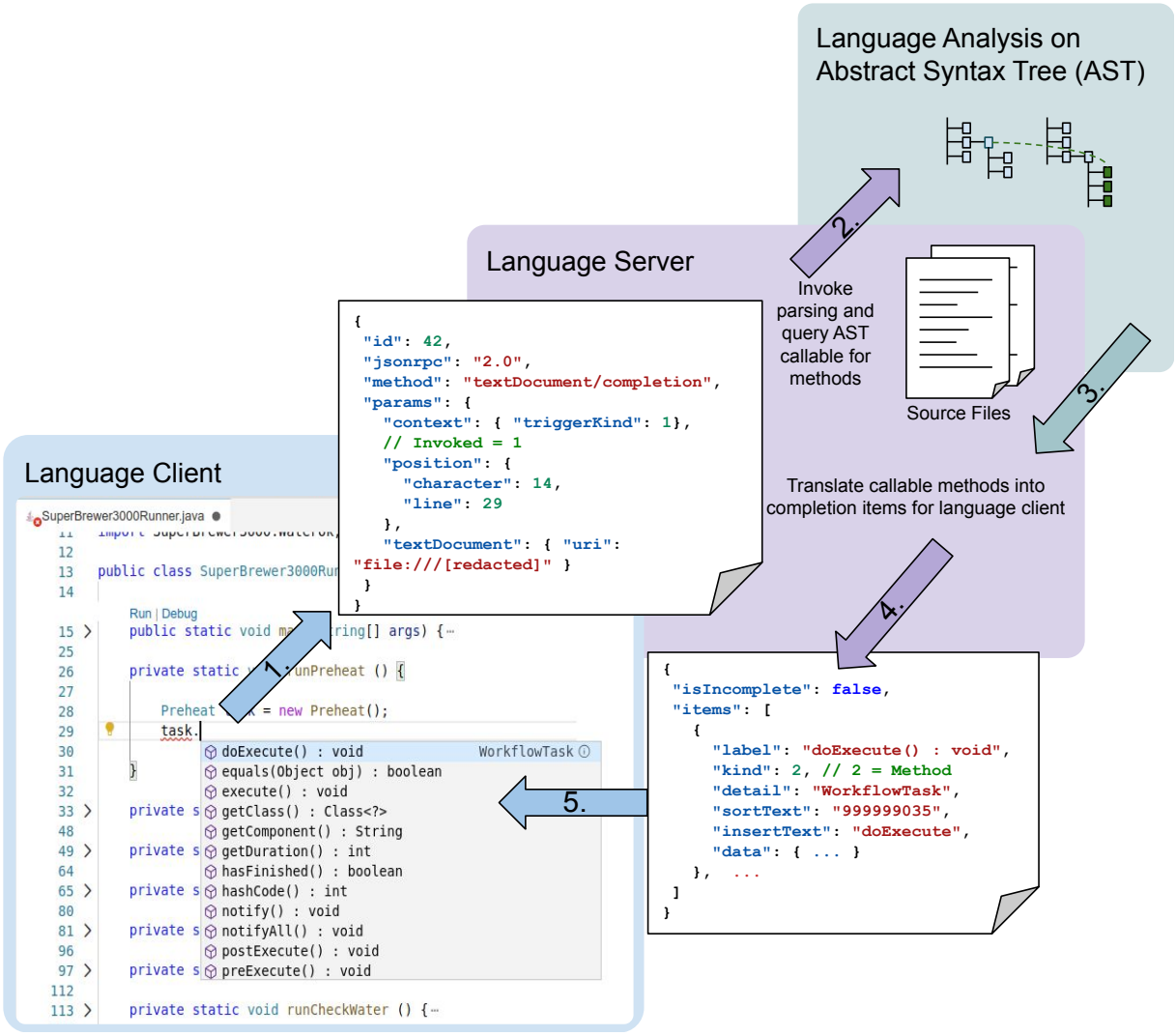

Figure 3: Example interactions between language client and language server for auto-completion

does not include static code editor features, such as syntax highlighting, as this can be achieved on the client based on a static grammar definition. Only more sophisticated highlighting, which requires knowledge about the language – often referred to as semantic highlighting – is handled on the server and thus part of the protocol. This choice of abstraction level makes the protocol versatile and applicable to vastly different types of languages, whether declarative or imperative, object-oriented or functional.

To gain a better overview of the underlying concept, let's consider an example of the interaction of a language client and a language server in Fig. 3. The language client is a language-agnostic component providing generic editing capabilities, such as presenting files, navigating across lines, adding and deleting characters, etc., but has no understanding of the underlying programming language concepts, such as methods of a Java class. If a user, however, triggers an editor feature that requires an understanding of the underlying programming language, such as the auto-completion of a method call for a specific Java object, the language client sends a request to the language server with the required context information, such

as the character position in the text document. The language server, focusing on programming language analysis, such as parsing and querying abstract syntax trees, computes the list of possible completion items based on its internal representation of the text documents' source code. The completion items sent by the server only contain information relevant to the user interface, such as a label and an indicator for the icon and the text operation to be applied, if the user selects the completion item. Based on this information, the editor can then present the completion items and, if invoked, apply the respective text operation of the completion item.

3.2 Protocol Specification

The basic LSP protocol consists of request, response, and notification messages exchanged via the JSON Remote Procedure Call protocol (JSON-RPC[7]). Request messages and notification messages include a *method* identifier to indicate the type of request or notification, as well as additional parameters, which depend on their type. Response messages, in turn, include an identifier pointing to the original request message and a *result* parameter to carry the payload of the response. In addition, specific messages are available to cancel requests and report progress.

The message types of the actual protocol are grouped into categories, such as messages to control the *lifecycle* of a server, to specify and update the *workspace* configuration, to synchronize *text documents* between the client and the server, to publish *diagnostics* identified by the server, and to fulfill certain *language features*, such as *completion*, *signature help*, *formatting*, etc. Alongside well-defined parameter types, most of these messages have a direction specifying whether they are intended to be sent from the client to the server or vice versa.

As an example, Fig. 4 depicts the exchanged messages of communication between a code editor (language client) and a language server. The client manages the life cycle of a server. Therefore, the client sends an *initialize* request to the server providing it with initialization parameters such as the workspace folder(s), the LSP capabilities, the version number, and other initialization options if needed. The server processes this request and responds with an initialization result that composes, among others, the server LSP capabilities and the server version. Once the client is ready to accept further messages, it sends an *initialized* notification back to the server. Based on this message exchange, the client and the server agreed on the mutually available LSP capabilities that form the basis of the subsequent communication. Both the client and the server have read access to the workspace location, but only the client usually requires write access. This enables, for instance, the server to start indexing the workspace directories or recovering a cache for the relevant source files.

Once the client opens a document, it notifies the server with a *didOpen* notification, which passes the document identifier and the initial contents of the document to the server. This message is the starting point of synchronizing the document's in-memory state between the client and the server. Whenever the user performs changes in the opened document, the language client sends *didChange* notifications to ensure that the client and server are operating on the same in-memory state of the document. This synchronization is required, as the client and server are running in separate processes – or even containers – and thus don't share memory. Nevertheless, the server shall be able to, e. g., *publish diagnostics* at any time, even if document changes haven't persisted on disk yet. The client then renders these diagnostics with the provided severity and message at the respective text range in the document.

Whenever the user performs typical code editor functions, e. g., requesting auto-completion on a specific position in the document or invoking the navigation to the definition of a symbol, the client sends a corresponding request with the current document position to the server to obtain the relevant information for performing this action. In the case of requesting the *definition* of a symbol, the server response contains the document *location*

[7] https://www.jsonrpc.org/, last accessed: 13.08.2023

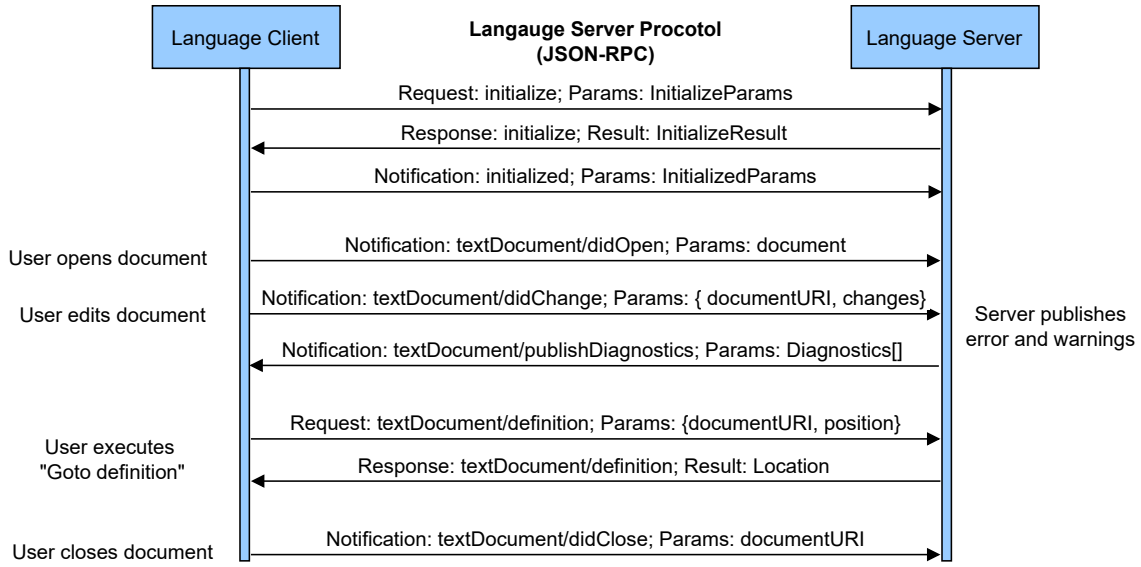

Figure 4: LSP-based communication during a routine editing session – adapted from (Microsoft 2022a)

within the same or another document. Other code editor functions will return different results, ranging from completion suggestions for auto-completion (see Fig. 3) to text edits for rename operations. Once the user closes the document, the client sends a *didClose* notification allowing the server to dispose of the in-memory state of the document and use the persisted state on disk from now on.

Over time, the range of language features supported by LSP kept growing and likely continues to grow in the future. To avoid forcing language server and client implementations to catch up immediately with supporting all of these features to be compatible, LSP introduces the notion of *capabilities*, which allows clients and servers to announce which of those features they support. Therefore the client attaches a *ClientCapabilities* configuration object to the *initialize* request and the server adds a *ServerCapabilities* configuration object to the *initialized* response. These capabilities indicate whether, for instance, the client supports auto-completion or markdown in hover information or whether the server includes a signature help provider or can resolve type definitions. The notion of capabilities is critical to enable incremental but still compatible development of language clients and servers. Moreover, it makes the protocol even more versatile as not all protocol capabilities may make sense for all languages.

4 Developing Language Servers

With the rising adoption of the language server protocol and the increasing number of language servers being developed, also the availability of generic frameworks and Software Development Kits (SDKs) supporting the implementation of language servers increased over the last few years. As of today, the open-source software ecosystem provides more than 200 language server SDKs for a variety of programming languages, including C++, Java, Python, Rust, and JavaScript (cf. Fig. 1). The available SDKs vary in their programming language and their focus. While some of them mainly provide interfaces for the protocol messages and data types, most SDKs also provide a framework, including server infrastructure and JSON-RPC support. These SDKs enable developers to focus on developing the language editing support by implementing the required behavior to server life-cycle events, such as startup, shutdown, and connection, and implementing the respective providers for code completion, hover information,

diagnostics, etc. For a complete overview of available SDKs and frameworks, please refer to the SDKs listed on the language server protocol website (Microsoft 2022b).

The most popular choice for Node.js is *vscode-languageserver-node* (VSCode Language Server - Node 2022), which is driven by the maintainers of the language server protocol specification itself and, thus, may be considered as the reference implementation of the protocol. However, it contains not only the language server protocol definition and data types in TypeScript but also an infrastructure for the underlying JSON-RPC client-server connection and a dedicated framework that can be extended for implementing the language support of a specific language. To create a language server with *vscode-languageserver-node*, developers register callback functions for certain LSP events and requests, such as on initialization, on the opening of documents, or on completion requests (cf. Fig. 3 for an example). These callback functions are then responsible for parsing the file when it is opened and computing completion results for a particular text position.

Similar SDKs are also available for other languages, e.g., *LSP4J* (LSP4J 2021) for Java, *tower-lsp* (tower-lsp 2021) for Rust, and *pygls* (pygls 2021) for Python. They all provide native implementations of the protocol definition and types and a JSON-RPC infrastructure and framework for developing language servers.

Alongside those dedicated LSP SDKs, a few language toolkits for developing domain-specific languages have integrated LSP support. With that, users can not only produce a parser and an abstract syntax tree for a particular domain-specific language but also produce a language server to add language support to VSCode and other LSP-enabled editors. A popular language toolkit with LSP support is *Xtext* (Xtext 2021), which is based on Java and the Eclipse Modeling Framework (Steinberg et al. 2008). Xtext provides a grammar language, similar to the Extended Backus-Naur Form, with dedicated constructs for defining cross-references, attributes, types, etc. Xtext derives the abstract syntax tree from such a grammar definition and generates a parser, a linker, and an extensible language support infrastructure, including a language server, alongside an Eclipse editor integration (cf. (Bünder 2019) for a recent analysis). More recently, a Node.js-based language toolkit named *Langium*[8] (Langium 2021) gains traction, which follows the underlying idea of Xtext but puts a focus on creating language servers based on grammar definitions, see (Giner-Miguelez et al. 2022) for a recent application case report. *Anycode*[9] is another framework for efficiently developing LSP-based language smarts. Anycode is particularly designed to provide language support for environments that don't allow for running actual language services, like https://github.dev or https://vscode.dev.

Since version 3.17 of the language server protocol, the specification entails a meta-model, which eases generating the LSP types and the SDK from that meta-model for specific languages[10] . There are already server implementors that are using the meta-model, e. g., Rust and Python[11] .

4.1 Siblings of the Language Server Protocol

The success of LSP and the benefits of its architecture did not end at "just" becoming a de facto standard for providing language support to code editors. Its underlying idea and architecture have been adopted and adjusted to enable other standard functionalities of development environments and engineering tools. The Debug Adapter Protocol is one such sibling.

When it comes to providing debugging capabilities to development tools, similar challenges arise as for providing language editing support. The necessary means for enabling developers to observe and even interact with the runtime state significantly depends on the respective programming

[8] https://langium.org/ last accessed 12.08.2023
[9] Anycode Github repository: https://github.com/microsoft/vscode-anycode, last accessed: 13.08.2023
[10] LSP meta model: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#metaModel, last accessed: 12.08.2023
[11] https://github.com/microsoft/lsprotocol, last accessed: 13.08.2023

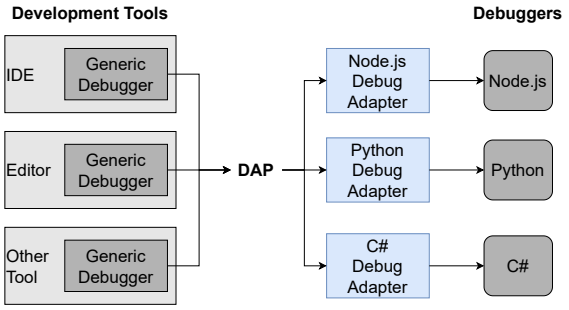

Figure 5: Debug Adapter Protocol Architecture (Microsoft 2022c)

language and runtime system. Thus, the effort to integrate a language-specific debugger has to be repeated for each development tool. To address this issue, the underlying idea of LSP has been co-developed with and applied to debugging by abstracting the interaction between a development tool and a debugger or runtime system into the so-called *Debug Adapter Protocol* (Microsoft 2022c). This protocol enables a generic user interface for debugging that interacts with a debug adapter that maps the protocol to language- or runtime-specific debugging commands (see Fig. 5). Consequently, the generic debugging interface can be used with any runtime system for which a debug adapter is available. The debug adapter protocol supports transferring specific runtime information from the observed runtime to the client, including the runtime state of a process, its suspended stack frames, variable values in the scope of a stack frame, etc. Moreover, the client can instruct the debugger to add or remove breakpoints and perform actions to control the execution, such as stepping over, into, or out of a stack frame.

As of August 2023, nine SDKs for developing debug adaptors, 11 tools supporting the DAP, and 68 debug adapters are registered on the official page of the DAP[12] .

Besides the Debug Adapter Protocol and the Graphical Language Server Protocol (see Sect. 5), many other adoptions of the LSP architectural pattern exist, such as the Trace-Server Protocol (TSP 2022) and the Specification Language Server Protocol Rask et al. (2021). They all encapsulate domain-specific smarts in a server component and introduce a dedicated protocol to transfer this knowledge to a rather domain-independent client on a user interface level.

5 LSP- and Web-based Modeling Tools

Graphical languages, such as UML (France et al. 1998) or BPMN (Chinosi and Trombetta 2012), often impose complex language editing rules, references across single diagram views, and model-wide analysis and validation logic. These complex and expensive language implementations are hardly transferable into a browser-based application due to computational complexity, workspace indexing requirements, and potential re-development costs of existing implementations, similar to the language support for programming languages. These striking parallels between the requirements for programming languages and graphical languages led to the adoption of the architectural pattern of language servers for diagram editors to unlock the benefits of LSP (cf. Sect. 3) for graphical languages (i. e., diagrams).

The open-source project *Eclipse Graphical Language Server Protocol* (Eclipse 2022) (GLSP[13]) provides a protocol specification and generic components for developing browser-based diagram clients and diagram servers in Java or based on Node.js. With the help of re-usable platform integrations, diagram editors made with GLSP can be embedded into multiple tool platforms, such as VS Code (Glaser and Bork 2021; Glaser et al. 2022), Eclipse Theia, and even the traditional Eclipse desktop rich-client platform.

5.1 From LSP to GLSP

While there are several similarities between LSP and GLSP, such as the overall architecture and the JSON-RPC-based client-server communication that follows a protocol abstracting common

[12] DAP web page: https://microsoft.github.io/debug-adapter-protocol/, last accessed: 16.08.2023

[13] https://www.eclipse.org/glsp, last accessed: 12.08.2023

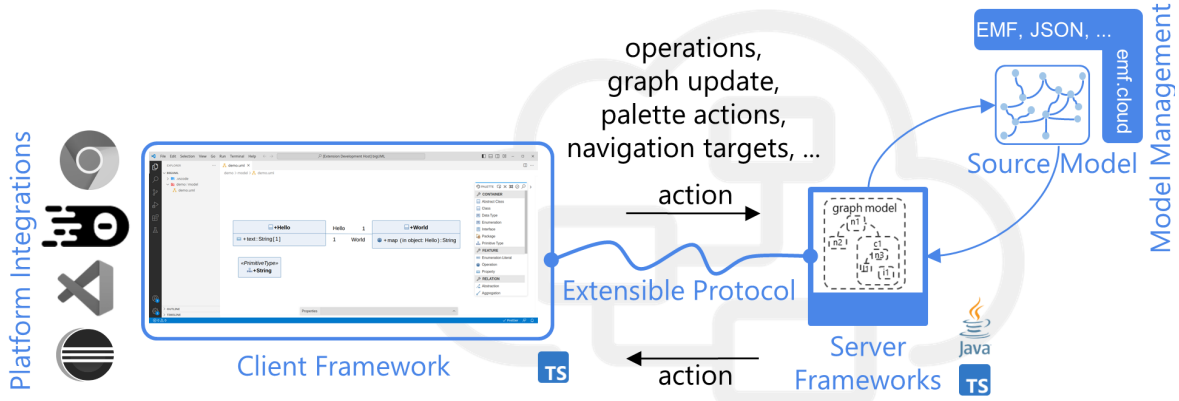

Figure 6: Architecture of GLSP-based modeling tools – adapted from (Eclipse 2022)

language features, there are also specific differences resulting from the different natures of textual languages and graphical languages. The most obvious difference is the common model based on which the client and server communicate. For textual languages, this common model uses files, lines, and characters to communicate text document positions, synchronizing states, etc. For diagrams, however, this common model talks about hierarchically structured nodes and edges and a two-dimensional coordinate system. Therefore, GLSP introduces a common *graphical model* that is shared between client and server and that allows describing the structure and state of a diagram based on an attributed, typed graph instead of plain text files (see Fig. 6). This graphical model is merely a structural description of the diagram, which the server creates from arbitrary source models, such as XMI files, EMF models, databases, etc. Thus, only the server needs to handle reading and writing the underlying model format and define how to translate it into one or more graphical models. The client eventually receives the graphical model and renders that into an SVG-based image using configurable views. Therefore, the specific diagram client is equipped by the diagram developer with specific SVG views that define how a particular node or edge type is translated into SVG. This enables complete flexibility for rendering, which is crucially required, as different diagram languages demand different shapes and styles—a challenge that isn't prevalent for textual languages.

Another notable difference between LSP and GLSP concerns editing. While editing textual languages always works precisely the same by adding or removing characters, independently of the textual language, graphical languages often impose strict editing constraints, such as the available types of nodes that can be created, allowing to connect only certain types of nodes with specific types of edges, etc. Therefore GLSP introduces dedicated protocol messages that allow clients to request editing operations for a particular diagram in a specific context and delegate performing the operations on the model to the server instead of applying changes directly to the client. This way, the server can constrain the available edit operations and control how the underlying source model, be it an XMI file, EMF model, or database, is eventually being modified for an edit operation that the client requested. Once the modification is performed, the server updates the graphical model on the client to reflect the changes graphically in the diagram on the client.

5.2 Developing GLSP-based Modeling Tools

GLSP has been under active development by the community since 2017, with the next major release v2.0.0 (expected September 2023). In its current version, GLSP provides four types of components

for realizing modern web-based modeling tools (see Fig. 6):

- **Server framework:** GLSP provides a server framework one can use to build particular diagram servers for e. g., UML or a domain-specific graphical modeling language on top. Initially, GLSP was focused on supporting the Eclipse Modeling Framework (EMF), based on which many modeling languages and their language-specific logic are already implemented and GLSP servers have been mainly written in Java. In the meantime, this support opened up to arbitrary model management frameworks, whether it is EMF, a JSON file, a database, or a remote REST service. More recently, GLSP also added a framework that enables developing GLSP servers with TypeScript[14] .
- **Client framework:** GLSP also provides a client framework. Similarly to the server framework, one can build a particular graphical modeling language client including the definition of the rendering with SVG, styling, and user interaction on top of the provided GLSP client framework. As the rendering and user interaction may heavily differ between one graphical modeling language and another, the client framework allows users to take full control over the SVG view implementations for rendering and enable their customization, as well as adding additional editing tools to control user interactions.
- **Protocol:** The messages that can be exchanged between the GLSP clients and servers are specified in a flexible and extensible GLSP protocol which standardizes, on a language-agnostic abstraction level, the communication between arbitrary clients and servers.
- **Platform integration:** GLSP provides platform integrations—reusable components that take an implemented GLSP diagram client and integrate it seamlessly into platforms such as Eclipse RCP, Eclipse Theia, or VS Code. These components provide the clue code necessary to register an editor to a certain file type or some other commands specific to the integrated platform. With that, GLSP aims to enable the integration of GLSP editors into multiple tool platforms and applications with maximum reuse.

[14] https://github.com/eclipse-glsp/glsp-server-node, last accessed: 12.08.2023

5.3 Comparison to the Use of Traditional Metamodeling Platforms

Although a comprehensive and systematic comparison between GLSP-based tool development and established traditional metamodeling platforms is not within the scope of this paper, we want to briefly discuss our experiences with working with GLSP and contrast this to our experience of developing tools in the past with traditional metamodeling platforms. Our comparison is primarily focusing on two essential aspects: the tool architecture (Sect. 5.3.1), and the tool development process (Sect. 5.3.2).

5.3.1 Tool Architecture

Traditionally, modeling tool development with metamodeling platforms like ADOxx, EMF, MetaEdit+, and Microsoft Modeling SDK follows more or less the architectural pattern depicted in Fig. 7. The developers of the metamodeling platforms define a platform-specific meta-metamodel and a generic platform-specific metamodel. On this level of genericity, the developers realize manifold functionality like user-, model-, access-, and data management, the visualization and (de-)serialization of models, and also some model processing functionality like queries, simulations, or code generators. Modeling tool developers can then inherit from the generic platform-specific metamodel and functionality, thereby introducing their domain-specific concrete metamodel and the corresponding domain- and language-specific functionality. Through this inheritance, the domain-specific tools automatically inherit the rich, extensible platform features developed on the generic level—the major driver of efficiency of these traditional tool development platforms.

In contrast, GLSP follows the architecture depicted in Fig. 6 and is not tightly embedded within

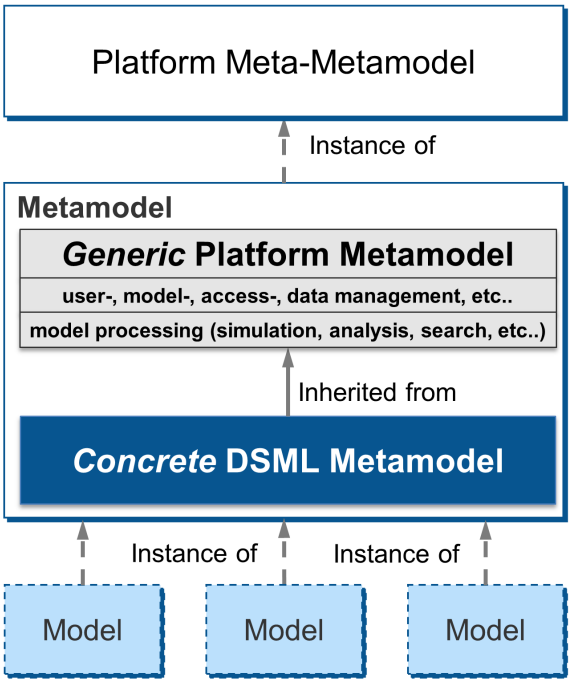

Figure 7: Architecture of traditional metamodeling platforms

a specific modeling framework. Instead, its core generic functionality revolves around diagram editor capabilities, such as rendering a diagram, providing editing tools for a diagram, performing operations on a diagram, and showing, for instance, validation errors on top of it. At its core, GLSP is agnostic to the actual model management including the underlying modeling framework, and externalizes the mapping of an underlying model to a diagram to the developer adopting GLSP. Thus, the core API of GLSP expects an implementation to load, save, and manipulate the underlying model, as well as to provide a mapping of an underlying model to GLSP's graphical metamodel used to represent diagrams, which is basically a Labelled Property Graph composed of nodes and edges. This entails many advantages, as GLSP can be used with arbitrary meta modeling frameworks or data sources and decouples many diagram-editor-related functionalities, such as diagram rendering and editing tools, from other parts, such as model manipulation and model management.

At the same time, however, this entails more effort with respect to the integration of a specific metamodel (i. e., the abstract syntax)—the *'heart'* of any modeling tool, as this integration has to be provided by the developers in terms of a transformation of an underlying model into the graphical model, as well as hooking up certain other functionalities that a metamodeling framework may provide, such as validation. To mitigate—but certainly not remove—this additional effort, the community around GLSP has provided generic integrations for certain modeling frameworks, such as EMF. Those integrations provide generic implementations mapping from the GLSP API to the respective functionalities of the modeling framework. This, as an example, allows for the reduction of custom code for loading and saving models and allows the use of the native API of the modeling framework to implement model manipulations (e. g. EMF commands) instead of implementing them based on the GLSP API.

A huge benefit, we see from GLSP-based modeling tools is their support for *blended modeling* (Ciccozzi et al. 2019; David et al. 2023; Glaser and Bork 2021) i. e., the simultaneous use of multiple concrete syntaxes based on one abstract syntax. Due to the loose coupling of the individual GLSP frameworks (see Fig. 6), one GLSP server can easily be used by several GLSP clients, each of which representing (a different subset of) the entire model with a separate concrete syntax (e. g., one graphical, one textual, one forms-based). The GLSP architecture thereby naturally supports decomposing one overarching abstract syntax into different viewpoints, each of which is provided by *projectional* views on the core metamodel (Voelter and Lisson 2014). The same is also true the other way around: as GLSP is not tied to a specific modeling framework, a single GLSP editor can represent multiple underlying models, which may even be managed in different modeling frameworks or data sources on the GLSP server.

A key motivation for this approach in GLSP—similar to LSP—is to prioritize decoupled, flexible diagram editing capabilities that can be reused and shared across multiple tool platforms, modeling

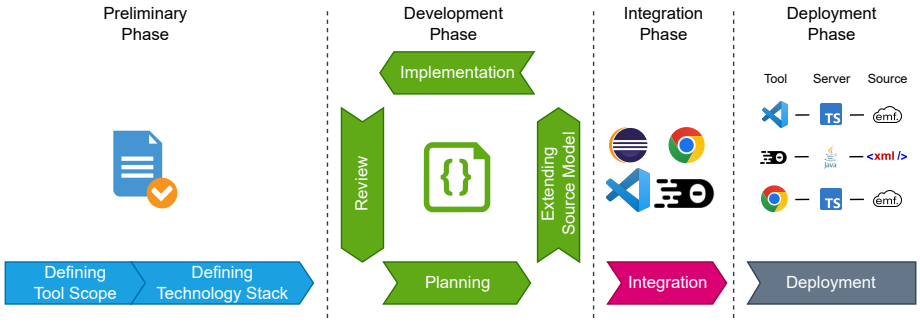

Figure 8: Development and operation process for GLSP-based web modeling tools (Metin and Bork 2023)

languages, and even modeling frameworks, rather than focusing on the integration of a single tool platform or modeling framework. Such integrations may, if needed, be put on top of the GLSP API, as has been done with tool platforms, such as VS Code and Eclipse Theia, or modeling frameworks, such as EMF. Another problem we faced in the past with traditional metamodeling platforms is their limited support to natively support the reuse of existing metamodels or parts thereof—a requirement stressed a long time in research which remains recent (Emerson and Sztipanovits 2006; Lédeczi et al. 2001; Mora Segura et al. 2023; Pfeiffer et al. 2023). The separation of concerns and the loose coupling of the individual GLSP frameworks greatly fosters the reuse of (parts of) developed modeling tools.

5.3.2 Tool Development Process

Aside from the previously discussed differences regarding the tool architecture, one can also identify significant differences with respect to the process and the tasks involved when developing modeling tools with GLSP in comparison to traditional metamodeling platforms.

Tool development with traditional metamodeling platforms often follows an iterative approach that centers the metamodel. Tool developers first introduce their domain-specific metamodel by inheriting from the generic platform-specific metamodel. Afterward, the concrete syntax, i. e., the visual representation of the metamodel concepts needs to be specified and the functionality enabled by the modeling tool needs to be developed. Finally, and optionally, integrations to other tools and applications can be realized.

When developing GLSP-based modeling tools (see Fig. 8), a developer needs to first think about the intended integration and deployment scenarios of the tool as these decisions influence the technology choices adhering to the selection of the appropriate GLSP server frameworks (e. g. Java-based vs. node-based server), which may certainly entail additional complexity to an inexperienced adopter of GLSP. For most traditional metamodeling platforms these considerations are inapplicable, anyway, as they enforce a specific technology stack and mostly often constrain the tool deployment to desktop client applications.

6 LSP Research Opportunities

LSP opens various directions for research by offering opportunities for the information systems engineering and conceptual modeling research communities that heavily develop and use modeling tools (Frank et al. 2014; Sandkuhl et al. 2018). Using the LSP and the GLSP yields opportunities for building highly flexible and interoperable modeling editors that can be accessed via the browser or through open platforms, such as VS Code or Eclipse Theia (Bork et al. 2023). An Entity Relationship modeling tool that has been released as an extension through the VS Code ecosystems shows the feasibility of facilitating LSP to elevate modeling tools to the web and to efficiently reach a broad audience (Glaser and Bork 2021; Glaser

et al. 2022). Other GLSP-based modeling tools for UML[15] and BPMN[16] have been released recently.

From a software engineering point of view, LSP brings the opportunity to develop other language servers for upcoming programming languages. Instead of developing specialized support for the diverse IDEs, engineers can develop one language server that can be used directly by all supporting LSP clients (cf. Fig. 1). Moreover, research may focus on applying the generic characteristics of LSP to support further purposes similar to the Debug Adapter Protocol and the Graphical Language Server Platform.

Another research opportunity emerges from the disruptive changes LSP causes in the tool development and IDE markets. Whole ecosystems like the one surrounding VS Code emerge rapidly. LSP's effects on software engineers and other ecosystems should be investigated. LSP changes the way software engineers develop software systems. Empirical software engineering research should thus investigate the effects standardized protocols like LSP have on, e. g., developer productivity and loyalty. Initial empirical insights into how software engineers realize language servers, based on the analysis of 30 publicly available language servers, are reported in (Barros et al. 2022). LSP and its adaptations for modeling like GLSP may also ease the development of much-needed support for e. g., collaborative modeling (Zweihoff and Steffen 2021), model execution (Khorram et al. 2022; Leroy et al. 2020), model simulation (Liboni and Deantoni 2020), or device-specific advanced model representation and interaction (Bork and Carlo 2023; Carlo et al. 2022) including AR/VR (Muff and Fill 2021; Yigitbas et al. 2022) for domain-specific modeling languages.

[15] BIGUML modeling tool: https://marketplace.visualstudio.com/items?itemName=BIGModelingTools.umldiagram, last accessed: 16.08.2023

[16] Open-BPMN modeling toolhttps://marketplace.visualstudio.com/items?itemName=open-bpmn.open-bpmn-vscode-extension, alast accessed: 16.08.2023

7 Conclusion

In this paper, we introduced the history, the core characteristics, and the concepts composing the Language Server Protocol (LSP). We showed, how new language servers can be developed and further, how different siblings and spin-offs from LSP extend its generic characteristics to cover additional purposes like the support of debugging. An emphasis was then put on the Eclipse Graphical Language Server Platform (GLSP) that adopts and extends the LSP protocol for graphical languages (i. e., models or diagrams) and enables the development of highly flexible and interoperable web-based modeling tools. The paper closed with critical reflections on LSP and opportunities for future research.

References

Barros D., Peldszus S., Assunção W. K. G., Berger T. (2022) Editing support for software languages: implementation practices in language server protocols. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, Montreal, Quebec, Canada, October 23-28, 2022. ACM, pp. 232–243

Bork D., Carlo G. D. (2023) An extended taxonomy of advanced information visualization and interaction in conceptual modeling. In: Data & Knowledge Engineering

Bork D., Langer P., Ortmayr T. (2023) A Vision for Flexibile GLSP-based Web Modeling Tools. In: 16th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modeling (PoEM 2023). in press

Bünder H. (2019) Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages. In: Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, February 20-22, 2019. SciTePress, pp. 129–140

Carlo G. D., Langer P., Bork D. (2022) Rethinking Model Representation - A Taxonomy of Advanced Information Visualization in Conceptual Modeling. In: Conceptual Modeling - 41st International Conference Vol. 13607. Springer, pp. 35–51

Chinosi M., Trombetta A. (2012) BPMN: An introduction to the standard. In: Comput. Stand. Interfaces 34(1), pp. 124–134

Ciccozzi F., Tichy M., Vangheluwe H., Weyns D. (2019) Blended Modelling - What, Why and How. In: 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019. IEEE, pp. 425–430

David I., Latifaj M., Pietron J., Zhang W., Ciccozzi F., Malavolta I., Raschke A., Steghöfer J., Hebig R. (2023) Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study. In: Softw. Syst. Model. 22(1), pp. 415–447

Eclipse. https://www.eclipse.org/glsp/. Last Access: last accessed: 22.05.2023

Emerson M., Sztipanovits J. (2006) Techniques for metamodel composition. In: OOPSLA–6th Workshop on Domain Specific Modeling, pp. 123–139

France R. B., Evans A., Lano K., Rumpe B. (1998) The UML as a formal modeling notation. In: Comput. Stand. Interfaces 19(7), pp. 325–334

Frank U., Strecker S., Fettke P., vom Brocke J., Becker J., Sinz E. J. (2014) The Research Field "Modeling Business Information Systems" - Current Challenges and Elements of a Future Research Agenda. In: Bus. Inf. Syst. Eng. 6(1), pp. 39–43

Gamma E., Beck K. L. (2004) Contributing to Eclipse - principles, patterns, and plug-ins. The Eclipse series. Addison-Wesley

Giner-Miguelez J., Gómez A., Cabot J. (2022) DescribeML: A Tool for Describing Machine Learning Datasets. In: ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion). ACM

Glaser P.-L., Bork D. (2021) The BIGER Tool – Hybrid Textual and Graphical Modeling of Entity Relationships in VS Code. In: 25th IEEE International Enterprise Distributed Object Computing Workshop, EDOC Workshops 2021, pp. 337–340

Glaser P.-L., Hammerschmied G., Hnatiuk V., Bork D. (2022) The BIGER Modeling Tool. In: Proceedings of the ER Forum and PhD Symposium 2022 co-located with 41st International Conference on Conceptual Modeling (ER 2022) Vol. 3211. CEUR-WS.org

Gunasinghe N., Marcus N. (2022) Language Server Protocol and Implementation. Springer

Khorram F., Bousse E., Mottu J., Sunyé G., Gómez-Abajo P., Cañizares P. C., Guerra E., de Lara J. (2022) Automatic test amplification for executable models. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022, Montreal, Quebec, Canada, October 23-28, 2022. ACM, pp. 109–120

Langium. https://github.com/langium/langium. Last Access: last accessed: 22.05.2023

Lédeczi Á., Nordstrom G., Karsai G., Volgyesi P., Maroti M. (2001) On metamodel composition. In: Proceedings of the 2001 IEEE International Conference on Control Applications (CCA'01)(Cat. No. 01CH37204). IEEE, pp. 756–760

Leroy D., Bousse E., Wimmer M., Mayerhofer T., Combemale B., Schwinger W. (2020) Behavioral interfaces for executable DSLs. In: Softw. Syst. Model. 19(4), pp. 1015–1043

Liboni G., Deantoni J. (2020) A Semantic-Aware, Accurate and Efficient API for (Co-)Simulation of CPS. In: Software Engineering and Formal Methods. SEFM 2020 Collocated Workshops. Springer, pp. 280–294

LSP4J. https://github.com/eclipse/lsp4j. Last Access: last accessed: 22.05.2023

Metin H., Bork D. (2023) On Developing and Operating GLSP-based Web Modeling Tools: Lessons Learned from bigUML. In: Proceedings of the 26th International Conference on Model Driven Engineering Languages and Systems, MODELS 2023. IEEE

Microsoft. https://microsoft.github.io/language-server-protocol/overviews/lsp/overview. Last Access: last accessed: 22.05.2023

Microsoft. https://microsoft.github.io/language-server-protocol/implementors/sdks. Last Access: last accessed: 22.05.2023

Microsoft. https://microsoft.github.io/debug-adapter-protocol/overview. Last Access: last accessed: 22.05.2023

Mora Segura A., de Lara J., Wimmer M. (2023) Modelling assistants based on information reuse: a user evaluation for language engineering. In: Software and Systems Modeling

Muff F., Fill H.-G. (2021) Initial Concepts for Augmented and Virtual Reality-based Enterprise Modeling.. In: ER Demos/Posters, pp. 49–54

Pfeiffer J., Rumpe B., Schmalzing D., Wortmann A. (2023) Composition operators for modeling languages: A literature review. In: Journal of Computer Languages

pygls. https://github.com/openlawlibrary/pygls. Last Access: last accessed: 22.05.2023

Rask J. K., Madsen F. P., Battle N., Macedo H. D., Larsen P. G. (2021) The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions. In: F-IDE2021, in press

Rodríguez-Echeverría R., Izquierdo J. L. C., Wimmer M., Cabot J. (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, pp. 370–380

Sandkuhl K., Fill H., Hoppenbrouwers S., Krogstie J., Matthes F., Opdahl A. L., Schwabe G., Uludag Ö., Winter R. (2018) From Expert Discipline to Common Practice: A Vision and Research Agenda for Extending the Reach of Enterprise Modeling. In: Bus. Inf. Syst. Eng. 60(1), pp. 69–80

Steinberg D., Budinsky F., Merks E., Paternostro M. (2008) EMF: eclipse modeling framework. Pearson Education

tower-lsp. https://github.com/ebkalderon/tower-lsp. Last Access: last accessed: 22.05.2023

TSP. https://github.com/theia-ide/trace-server-protocol. Last Access: last accessed: 22.05.2023

Voelter M., Lisson S. (2014) Supporting Diverse Notations in MPS' Projectional Editor. In: Proceedings of the 2nd International Workshop on The Globalization of Modeling Languages, GEMOC@Models 2014. CEUR Workshop Proceedings Vol. 1236. CEUR-WS.org, pp. 7–16

VSCode Language Server - Node. https://github.com/Microsoft/vscode-languageserver-node. Last Access: last accessed: 22.05.2023

Xtext. https://github.com/eclipse/xtext-core. Last Access: last accessed: 22.05.2023

Yigitbas E., Gorissen S., Weidmann N., Engels G. (2022) Design and evaluation of a collaborative UML modeling environment in virtual reality. In: Software and Systems Modeling, pp. 1–29

Zweihoff P., Steffen B. (2021) A Generative Approach for User-Centered, Collaborative, Domain-Specific Modeling Environments. In: CoRR abs/2104.09948