IDE as Code: Reifying Language Protocols as First-Class Citizens

Pierre Jeanjean, Benoit Combemale, Olivier Barais

IDE as Code: Reifying Language Protocols as First-Class Citizens

Pierre Jeanjean
Inria, Univ Rennes, CNRS, IRISA
Rennes, France
pierre.jeanjean@inria.fr

Benoit Combemale
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
benoit.combemale@irisa.fr

Olivier Barais
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
olivier.barais@irisa.fr

ABSTRACT

To cope with the ever-growing number of programming languages, manufacturers of Integrated Development Environments (IDE) have recently defined protocols as a way to use and share multiple language services (*e.g.,* auto-completion, type checker, language runtime) in language-agnostic environments (i.e., the user interface provided by the IDE): the most notable are the Language Server Protocol (LSP) for textual editors, and the Debug Adapter Protocol (DAP) for debugging facilities. These protocols rely on a proper specification of the services that are commonly found in the tool support of general-purpose languages, and define a fixed set of capabilities to offer in the IDE. However, new languages appear regularly offering unique constructs (*e.g.,* Domain-Specific Languages), and supported by dedicated services to be offered as new capabilities in IDEs. This trend leads to the multiplication of new protocols, hard to combine and possibly incompatible (*e.g.,* overlap, different technological stacks). Beyond the proposition of specific protocols, the goal of this paper is to stress out the importance of being able to specify language protocols and to offer IDEs to be configured with such protocol specifications. We present our vision by discussing the main concepts for the specification of language protocols, and an approach that can make use of these specifications in order to deploy an IDE as a set of coordinated, individually deployed, language capabilities (*e.g.,* microservice choreography). IDEs went from directly supporting languages to protocols, and we envision in this paper the next step: *IDE as Code*, where language protocols are created or inferred on demand and serve as support of an adaptation loop taking in charge of the (re)configuration of the IDE.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**; **Software as a service orchestration system**.

KEYWORDS

ide, language protocols, iac, microservices, domain specific languages

1 MOTIVATION

While Integrated Development Environments (IDEs) have been initially designed as a way of supporting the various services of a given language (*e.g.,* facilities for editing, debugging, checking, compiling...), modern polyglot developments stressed out the need of integrating the services of various languages in a given specific IDE. To prevent the development of the language services specifically for each existing IDEs, language protocols have become in the recent years a topic of interest in the language engineering community. By communicating through well-defined protocols, the main language services can be reused across the various IDEs supporting the protocols. A direct consequence is that the responsibility to provide a proper support for a specific language is no longer a concern of the IDE manufacturer, but befalls on the language maintainers developing the services independently.

The first protocol, namely the Language Server Protocol (LSP)[1], was proposed by Microsoft in the context of the development of VS Code, to support common editing services of any languages provided in conformance to the protocol. It was designed around a set of services that was extracted from specialized code editors for the most commonly used general purposes languages. LSP, the Debug Adapter Protocol (DAP)[2], and most of the other language protocols we see nowadays specify the structure of the data exchanged between a single client (a UI component of an IDE) and a single server (backend providing the set of services needed by the client), and the requests and events that can be sent from one to the other. Most messages are included in what is referred to as "capabilities". The set of capabilities is set and fixed by the specification of the protocol. The idea is that both clients and servers can choose to implement a subset of the capabilities and notify the other, which should be able to use all the corresponding messages as per the specification.

However, fixing the set of services at the level of the protocol means that it might not fit every use-case. When considering domain specific languages (DSLs), for example, we often find some unusual features that can be tied either to the meta-language approach used to design the language (*e.g.,* generic services [3]), to the language itself (*e.g.,* paradigm, syntax) and even to the program itself (*e.g.,* current state representation). While we could argue that it would be pertinent to add some of these features as new capabilities to the existing protocols, it would be inconceivable to cover all specific use cases inside a single generic protocol. Still, the adoption of DSLs would greatly benefit from support by multiple major IDEs, as they would integrate better in the workflow of the users. Protocols are making this situation possible, although some features would be lost in the process [6].

With the success of LSP and DAP, we see new language protocols emerging both for new use cases (*e.g.,* Build Server Protocol[3], Test Adapter Protocol[4]) and for specific features not properly supported by the existing ones (*e.g.* using a graphical syntax in LSP [11]). The case of the graphical syntax is particularly interesting, because it led to the definition of two different protocols for the same purpose:

[1] cf. https://microsoft.github.io/language-server-protocol/
[2] cf. https://microsoft.github.io/debug-adapter-protocol/
[3] cf. https://build-server-protocol.github.io/
[4] cf. https://github.com/microsoft/vscode-debugadapter-node/issues/154

the Graphical Language Server Protocol[5] and the Graphical Server Protocol[6]. Another way to extend the features of LSP that was used in works such as [9] and [8] consists in arbitrarily adding support for new messages to a server and a client, and consider that all existing implementations will ignore them if they don't support them. While these two works provide very useful contributions, such an implementation raises concerns of maintainability and interoperability, which furthers motivates our vision.

In the long run, it would be counter-productive to keep making independent protocols for every use-case, as this defeats the purpose of ensuring proper support in all development environments. In a real-world situation, this would shift the challenge on the composability and compatibility of existing protocols, which is currently not addressed. Hence, instead of having a *server* that encompasses all services for a given language, we explore the idea of breaking it down into individual services interacting with each other. With a *client* defined similarly, and every interaction precisely specified, specific parts of the protocols could be better leveraged and reused in multiple services configurations. This also provides more flexibility to the overall architecture: specialized services can be moved or changed depending on the use-case, services can be added or removed at any moment, and it becomes possible to finely control the deployment of each service. With this paper we aim to introduce the need for specifying customized language protocols that precisely capture service interactions, and serve as support to automate the deployment and configuration of the IDE according to a given use case (*e.g.,* user preferences, activities, environment, etc.). We refer to this vision as *IDE as Code*.

To illustrate and motivate our vision, we consider in this paper an environment that provides omniscient debugging capabilities from an execution trace, similar to the one presented in [4]. If we want to access this service from another platform, DAP does have a *stepBack* request that we could implement. However, the backtracking mechanism gives more control than simply stepping back: for example, one can step inside and outside functions/methods. This is not a behavior defined in the specifications of DAP when stepping back. Also, generating and storing the execution trace is an expensive process that should not be enabled all the time, and would benefit from running on a separate host. Ideally, its activation should be controllable throughout a debugging session, and DAP does not have any request to manage that. Also, if we compute the execution trace for a part of the execution, we might as well show it to the user and let him navigate through its states, meaning that a different language protocol is needed to interact with some kind of trace manager client.

The remainder of this paper is structured as follows. In Section 2, we introduce the concept of *IDE as Code* and the specifications required for its implementation. Section 3 puts the emphasis on the need for a higher level specification of language protocols. A possible implementation through microservices orchestration is presented and discussed in Section 4. Some related works are referenced in Section 5. And finally, we propose a research agenda for the questions raised by this paper in Section 6.

[5]cf. https://www.eclipse.org/glsp/
[6]cf. https://obeonetwork.github.io/GraphicalServerProtocol/

2 VISION

Our vision aims at providing more customizable and adaptable IDEs for the end users through language services independent of any client. It is driven both by the need to support domain-specific language services in multiple environments and to contribute in the trend of turning IDEs into web platforms.

When we consider programming languages in the large, including both general-purpose and domain-specific languages, we need to account for an ever-growing number of language services. In contrast, at a given time, an IDE can only manage a subset of these. Since protocols such as LSP and DAP were designed around the features available in Visual Studio Code, it made perfect sense for Microsoft to also fix the set of the capabilities they offer to ensure their full support in their environment. However, this prevents the integration of additional services as provided in support of new (domain-specific) languages.

Instead of fixing a priori the set of services that a user can use in the IDE, we envision a more open approach where all services are made available at any time, and the user can customize the IDE, possibly at runtime, depending on their use case.

To this end, as illustrated by Figure 1, we expect language designers to not only provide their language specifications but also information about the protocol interactions required for the different services. This paper focuses mainly on this part, starting with Section 3.

From there, language service packages can be obtained, and users would be able to create an IDE configuration that includes the ones they need. The deployment of these packages would also be finely controlled to run some language services on suitable platforms, *e.g.*, with vast amounts of memory for execution logging or with powerful CPUs for compilation. Default IDE configuration and deployment specifications might also be provided by language designers for some use cases.

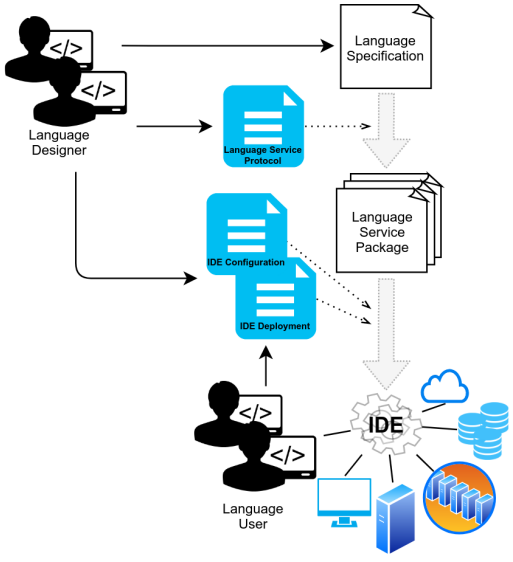

Figure 1: Vision Overview for *IDE as Code*

3 SPECIFYING LANGUAGE PROTOCOLS

The first step before attaining an *IDE as Code* is to unify the specifications of language protocols.

Current language protocols are basically provided as bi-directional APIs. While the data-flow between servers and clients is precisely specified, the control-flow is not explicit and can only be deduced from processing the documentation available in natural language. As a consequence, even though maintainers for servers and clients should be free of any specific implementation, they end up relying on Visual Studio Code as a reference.[7] Here, we propose that the control-flow between language services should be formalized and become a part of the protocol specification. The implementation of language service packages needs to separate this concern from the actual implementation.

As illustrated in Figure 2, individual language service packages should have their interactions properly specified in a language protocol. The left side of the figure represents the current practices in language engineering, where language services and their implementations can be automatically derived from a language specification, and the right side shows a conceptual metamodel of the specifications required to obtain language service packages. Language services are provided as "language capabilities", and the corresponding UI components as "UI services". In addition, mandatory packages are reified as first-order concepts, like the notion of workspace for example. We make the assumption that language services can retrieve their implementations from a dependency to the language specification (e.g., similarly to [7]), as their actual implementation is not the focus of this paper. From a given language protocol specification, a generative approach supports the automatic creation and deployment of the language packages.

For customization purposes, language service packages can be switched to more specialized implementations depending on the use-case, and services can be dynamically deployed, and then added or removed from the protocol specification. They might also specify particular hardware requirements to later drive the deployment.

[7] https://www.reddit.com/r/vim/comments/b3yzq4

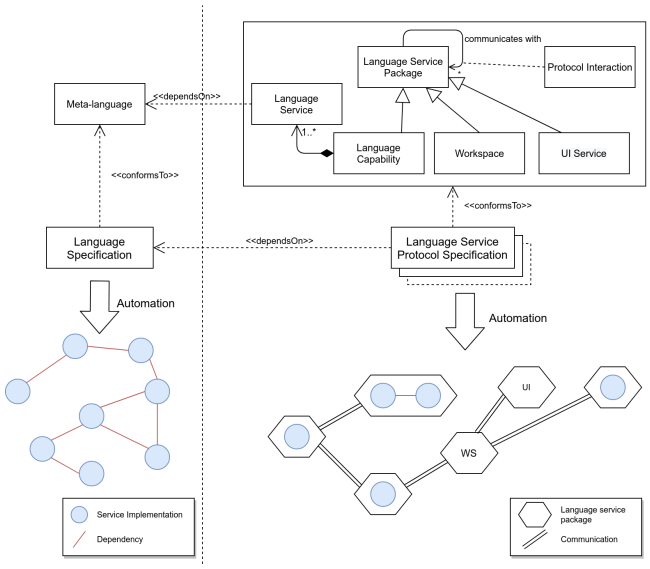

Figure 2: Protocol Specification for *IDE as Code*

4 PRELIMINARY RESULTS

In this section, we present some implementation choices we are currently exploring, and preliminary results to illustrate our vision. We present a specific IDE architecture, using the scenario of adding an omniscient debugging.

Domain-specific language designers need to provide services to support specific language capabilities. The debug adapter protocol, a protocol that should address every debugging concern, is lacking in features in regard to omniscient debugging as introduced by [4]: it requires support for execution traces management, and a backtracking interface that makes the distinction between stepping into and stepping over statements.

However, there is no need to reinvent the wheel. DAP is a very valuable and well needed contribution to unify the interfaces of different debuggers. For most of the capabilities it offers, it is in fact adapted for the different services needed to debug DSLs. The main issue is that it has missing features, and the fact that it does not offer any extension mechanism. So if one wanted to extend this protocol, one would have to define yet another protocol that would end up being very specific to their use-case, without anyone else adopting it. Ideally, it should be possible to import a debug adapter as a language service, and specify its interactions with other services to deploy it as a language service package. We could argue that such a language service could, and should, be refined into smaller packages, but the re-usability of existing language servers is a major concern at this stage.

Thus, we propose an extensible architecture that can support existing language server implementations, as illustrated by Figure 3. In order to support resource scaling for web IDEs that need to

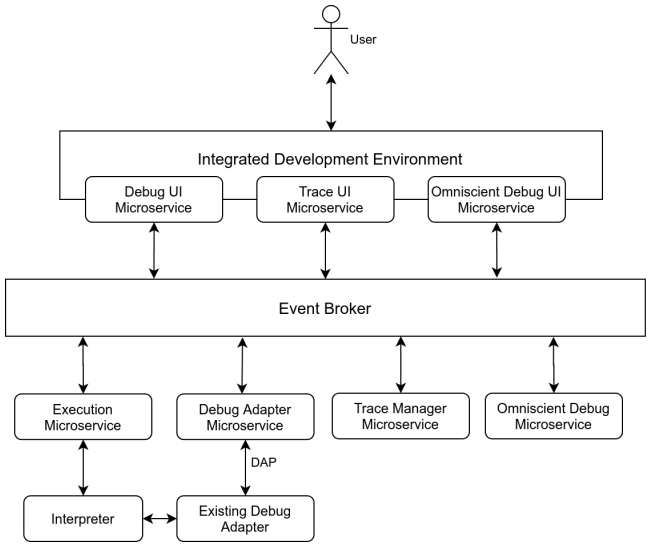

Figure 3: Example of a Microservice Choreography for Omniscient Debugging

serve multiple clients at the same time, the concept of language service packages is implemented as microservices. An event broker lets both language microservices and UI microservices interact as part of a choreography. In the figure, UI microservices are all included in the same platform, but technically they could also be deployed and integrated into others: the service displaying the execution trace could very well be embedded inside a completely independent web page, or as part of a Jupyter Notebook. Other implementation approaches such as OSGi plugins or Visual Studio Code extensions could also be considered, but comparing them is out of the scope of this paper. Here, we also made the decision to use JSON serialization and websockets for the transport in order to stay close to LSP implementations, but we do not argue that these are the most efficient nor the best choices.

The debug adapter microservice serves as a bridge between an existing debug adapter and the rest of the microservices. Its main goal is to turn the requests of DAP into events exchanges, in order to interact with the debug UI microservice. Implementing debugging language services in this manner might look like a weird approach, but it is a purely technical decision to show that this architecture can include existing protocols.

The trace manager microservice receives events about the data changes during the program execution, and builds the execution trace. In order to notify it, there is also an execution microservice that has direct access to the program interpreter. The omniscient debug microservice uses this trace to reset the execution state whenever the user needs to step back. The trace UI microservice provides feedback about the execution trace and the current state of the program.

While the omniscient debug microservice relies on the trace manager, the configuration of microservices presented here is not fixed. As a choreography, every microservice is aware of the addition or removal of the others, and is able to react if there is an impact on their workflow. So it is possible to only provide debugging services from the debug adapter depending on the available resources.

A domain-specific language would help in defining and maintaining the specifications of these microservices. We are aware that a language to specify a service-oriented architecture is nothing new, but one dedicated to manage language services is yet to exist and would help tremendously in the adoption of such an architecture. As such, we are considering a metalanguage designed to specify language services and their protocols, that provides constructs specific to IDEs, like those of workspaces, development resources (files), and run configurations for example. It also drives the choreography by letting the microservices define workflows in their protocol speficiation.

Figure 4 shows an instance of such a language. Data structures can be specified, and used as arguments for events. By separating events into different blocks, multiple communication channels can be managed inside the event broker that will deliver them. The different packages can explicitly declare that they require other packages in order to be relevant, through a mechanism of dependencies. Then, their workflow when receiving events is explicitly defined, and the tasks can consist in waiting for other events, sending events, or calling methods of imported language services.

```
data {
    TraceState {
        backInto: Step
        backOver: Step
        backOut: Step
    }
    Step { ... }
}

events {
    trace {
        state
        stateResult(TraceState)
    }
    stepBackIn
    stepBackInDone
    setState(Step)
    setStateDone
}

packages {
    tracemanager depends on execution listens trace {
        recv step -> call updateState(Step)
        recv trace/state -> call getStateResult(result)
            -> send trace/stateResult(result)
    }
    omniscientdebug listens trace
    depends on execution, debugadapter, tracemanager {
        recv stepBackIn -> send trace/state
            -> recv trace/stateResult(traceState)
            -> send setState(traceState.backInto)
            -> recv setStateDone
            -> send stepBackInDone
    }
}
```

Figure 4: Protocol Specification for *Step Back In* Service

5 RELATED WORK

Providing a language for specifying protocols and interactions and providing a compliance relationship to that protocol is not necessarily a new idea. Indeed, in the field of components based software engineering, Plasil *et al* [10] provides within their architecture description language a way to specify this behavioral contract of each component. Software designers can define component's behavior. The paper defines a protocol conformance relation. Using these concepts, the designer can check the adherence of a component's implementation to its specification at runtime, while the correctness of refining the specification can be verified at design time. In the multi-agents community, several agent-oriented programming languages such as JADEL [2] provide abstractions to define agents interaction protocols. In the networking domain, Burgy *et al.* [5] proposes a new language-based approach to developing protocol-handling layers, to improve their robustness without compromising their performance. The approach is based on the use of a domain-specific language to specify the protocol-handling layer of network applications that use textual HTTP-like application protocols.

An earlier definition of a meta-protocol can be found in [1]. The approach aimed at providing a higher level of flexibility in protocol-based communications. Multiple protocol specifications are available inside a repository, and two parties can decide on which one to use during a negotiation phase. From the agreed upon specification, the actual implementation of the protocol is generated on the fly for the programming language used to define the component. While we are aiming for a generative approach, the abilities to provide different protocol specifications through a centralized repository, and to automatically derive different implementations

depending on the use-case are ideas we would like to explore. The main difference in our approach aims at thinking in the specific domain of language protocols to facilitate the definition of modular and distributed IDE architectures. By focusing on this domain, it will be interesting to integrate business concepts as first-class entities directly into the protocol description language.

As for our implementation proposal, defining IDE language components as microservices has already been done in [7]. However, the focus of the presented work was mainly to analyze the trade-offs in modularizing and distributing these services, and categorize them based on these observations. Both of these works are, as such, complementary: our vision consists in making the specifications of language services explicit and executable, with their possible deployments and the protocols they use. Giving inputs to decide on the best architecture and platforms to use is out of the scope of our paper.

6 RESEARCH AGENDA

Our vision, summarized as *IDE as Code*, places the language protocol specification in the adaptation loop of the IDE, where language services are packaged and deployed dynamically, and provided as new capabilities to the user. From our preliminary results, we identify and discuss concrete challenges in the following.

Unified representation of language services specifications. As mentioned in Section 3, this paper focuses on the specification and deployment of language packages, and their interactions. We do not target a specific representation of the specifications of languages and their services. Which means that we either need to ensure that we only use generic concepts, or settle on a technology that doesn't rule out some language designs.

Domain-specific IDEs concepts. This paper proposes a first categorization of the IDE-specific concepts. In practice, this needs to fit all development platforms (e.g., notebooks, TUIs...) and properly introduce them as first-class constructs to specify language protocols.

Stateless vs stateful trade-off. Microservice architectures efficiently support resources scaling. But in practice, it is unlikely that every language package will be stateless. For instance, a package to manage the workspace requires a concept of session. To support web-based IDEs in practice, it is necessary to abstract the states of packages and still be able to scale with stateful microservices through synchronization mechanisms.

Automation of the deployment. Our vision revolves around an automatic deployment of the language service packages and their integration into IDEs. This requires either a generative approach, or an interpreter, possibly with an adaptation loop, for processing the protocol specifications.

Measuring the impact on performances. A direct consequence of microservicizing the different language services will be a multiplication of the number of messages exchanged to achieve any action, and numerous asynchronous waits. Beyond the flexibility, scale and collaboration, at the scale of a fully-featured IDE, this could hit in performances. A proper study of the possible impacts of this approach to the user experience is required. Alternatives (e.g. approaches to automatically compile different language service packages, possibly dynamically, back into bigger components when they are deployed on the same machine) and other technical choices for the serialization format (e.g., xml, protobuf) and the transport layer (e.g., pipes, WebRTC) are also worth investigating.

REFERENCES

[1] Ibrahim S. Abdullah and Daniel A. Menascé. 2013. The Meta-Protocol framework. *Journal of Systems and Software* 86, 11 (2013), 2711 – 2724. https://doi.org/10.1016/j.jss.2013.05.096

[2] Federico Bergenti, Eleonora Iotti, Stefania Monica, and Agostino Poggi. 2016. Interaction Protocols in the JADEL Programming Language. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Amsterdam, Netherlands) *(AGERE 2016)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/3001886.3001888

[3] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. 2016. Execution Framework of the GEMOC Studio (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering* (Amsterdam, Netherlands) *(SLE 2016)*. Association for Computing Machinery, New York, NY, USA, 84–89. https://doi.org/10.1145/2997364.2997384

[4] Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. 2018. Omniscient Debugging for Executable DSLs. *Journal of Systems and Software* 137 (March 2018), 261–288. https://doi.org/10.1016/j.jss.2017.11.025

[5] L. Burgy, L. Reveillere, J. Lawall, and G. Muller. 2011. Zebu: A Language-Based Approach for Network Protocol Message Processing. *IEEE Transactions on Software Engineering* 37, 4 (2011), 575–591.

[6] Hendrik Bünder. 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 - Volume 1: MODELSWARD,*. INSTICC, SciTePress, 131–142. https://doi.org/10.5220/0007556301310142

[7] Fabien Coulon, Alex Auvolat, Benoit Combemale, Yérom-David Bromberg, François Taïani, Olivier Barais, and Noël Plouzeau. 2020. Modular and Distributed IDE. In *SLE 2020 - 13th ACM SIGPLAN International Conference on Software Language Engineering*. Virtual, United States. https://doi.org/10.1145/3426425.3426947

[8] Jan Koehnlein. 2020. Beyond LSP: Getting Your Language into Theia and VS Code. (2020). https://www.eclipsecon.org/2020/sessions/beyond-lsp-getting-your-language-theia-and-vs-code EclipseCon 2020.

[9] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. 2020. Example-Based Live Programming for Everyone: Building Language-Agnostic Tools for Live Programming with LSP and GraalVM. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Virtual, USA) *(Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3426428.3426919

[10] F. Plasil and S. Visnovsky. 2002. Behavior protocols for software components. *IEEE Transactions on Software Engineering* 28, 11 (2002), 1056–1076.

[11] Roberto Rodriguez-Echeverria, 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)*. Association for Computing Machinery, New York, NY, USA, 370–380. https://doi.org/10.1145/3239372.3239383