UNIVERSITÀ DEGLI STUDI DI MILANO

RESEARCH PROJECT			
Applicant's Name	Federico Cristiano Bruzzone	PhD	Computer Science

Project title

Universal Language Server Protocol and Debugger Adapter Protocol for Modular Language Workbenches

Abstract

Domain-specific languages (DSLs) and language-oriented programming have become very successful tools in the development of complex software systems. To best suit their purpose, DSLs are very different from one another, yet many of them share commonalities in either their patterns or their implementation. The goal for language designers would be to spot those similarities and to exploit them in order to improve the reuse of preexisting implementations and minimizing the development from scratch. The most common approach for dealing with this task is the use of product line engineering ideas; this introduces the notion of language product line (LPL) to the DSL development. Nowadays, the research has produced several attempts to the creation of tools for variability management. This project will try to take all of them into account but will focus on the tools applying a bottom-up approach to LPL development, i.e. those in which the application engineering phase is performed before the domain engineering phase. Even among these approaches, the support for multi-dimensional variability is scarce. I propose to study a *formal method* to define a *multi-dimensional variability model* that takes each syntactic and sematic role into consideration using a bottom-up approach. As a case-study to translate the formal method into LPL development tools, I propose Neverlang, a framework for modular DSL definition developed by Università degli Studi di Milano, which also already has its own version of syntax-based LPL development tool by means of AiDE.

Project aims and their relevance in the context of the state of the art

Language Workbench	Modularization Supp.	Precompiled Feature Supp.	Native IDE gen.	LSP Gen.	LSP Mod.
JustAdd	◐	○	○	○	○
Melange	●	○	3rd party (EMF)	○	○
MontiCore	◐	◐	●	○	○
MPS	●	○	●	○	○
Rascal	○	○	●	○	○
Spoofax	●	◐	●	○	○
Xtext	○	◐	●	●	○

Comparison of language workbenches in terms of modularization, precompiled feature support, native IDE generation, LSP generation, and LSP modularization.

The primary aim of this project is to develop a Universal **Language Server Protocol**[1] (LSP) and **Debugger Adapter Protocol**[2] (DAP) for modular language workbenches. This endeavor seeks to address significant gaps and challenges developing LSPs and DAPs in the current landscape of language workbenches, particularly in the areas of modularization, composition, and interoperability. Current language workbenches such as Melange [12], MontiCore [22], Spoofax [19], and MPS [42, 43] have made significant strides in supporting modularization, composition, and IDE integration. The table below provides a comparison of various language workbenches in terms of their support for modularization, precompiled feature support, native IDE generation, LSP generation, and LSP modularization. The ● symbol indicates full support, ○ partial support, and ◐ limited support. However, their approaches are often fragmented and lack a standardized method for LSP and DAP generation and modularization. Neverlang [38, 40], developed at the *ADAPT-Lab*[3] of the Università degli Studi di Milano, being a comprehensive framework for language composition and modularization that supports the development of language product lines [24, 8] (LPLs), is a prime candidate for the implementation of the proposed LSP and DAP. The project will leverage the existing capabilities of Neverlang to develop a universal LSP and DAP that can be used across different programming languages and IDEs. This will enable developers to create external domain-specific languages [15] (DSLs) and general-purpose languages (GPLs) more effectively and efficiently, enhancing the overall development experience and productivity.

The project aims to achieve the following objectives:

Aim 1: Improve IDE and LSP Generation

Integrated Development Environment generation and support for the *Language Server Protocol* are essential for the practical use of domain-specific languages (DSLs). While

[1] https://microsoft.github.io/language-server-protocol
[2] https://microsoft.github.io/debug-adapter-protocol
[3] https://di.unimi.it/it/ricerca/risorse-e-luoghi-della-ricerca/laboratori-di-ricerca/adapt-lab

some language workbenches like Xtext [6] support LSP generation [2], many do not, limiting their usability across different editors and IDEs.
Relevance: By establishing a universal protocol for LSP and DAP, this project aims to bridge the gap, enabling language workbenches to generate IDE support and LSPs more seamlessly. This will ensure that languages developed using these workbenches can be used in any IDE that supports these protocols, enhancing their accessibility and utility.

Aim 2: Facilitate LSP and DAP Modularization
LSP and DAP modularization are not widely supported by current language workbenches [7]. This feature is crucial for allowing different language components to communicate and function cohesively within an IDE.
Relevance: Implementing support for LSP and DAP modularization will allow for better integration and interaction of various language features, thereby improving the overall development experience and capability of language workbenches. This aligns with the needs for more sophisticated and integrated language development tools as highlighted in the contemporary research and development literature.

Aim 3: Reduce to $\mathcal{O}(\mathcal{L})$ the number of combinations to support $\mathcal{L}$ languages
Before the advent of LSP and DAP, developers had to implement language support for each editor separately, having the number of combinations to support ℓ languages in $\mathcal{O}(\mathcal{L} \times \mathcal{E})$, where $\mathcal{E}$ is the number of editors. Currently, the number of combinations to support $\mathcal{L}$ languages is $\mathcal{O}(\mathcal{L} + \mathcal{E})$ [37], as the Microsoft LSP and DAP are editor-agnostic. This project aims to reduce the number of combinations to $\mathcal{O}(\mathcal{L})$, by developing a universal LSP and DAP that can be used across different programming languages and IDEs.
Relevance: Reducing the number of combinations required to support multiple languages will simplify the development process and make it more efficient. This will enable developers to create language support more quickly and effectively, enhancing the overall productivity and usability of language workbenches.

Aim 4: Leverage Neverlang for LSP and DAP LPL Development
Neverlang's capabilities for language composition and modularization make it an ideal platform for developing a universal LSP and DAP that caters to a variety of language needs. By leveraging Neverlang's LPL development features [14], the project will establish a reusable core for LSP and DAP functionalities, allowing for the creation of product line variations tailored to specific programming language requirements. This will significantly reduce development time and effort for creating LSPs and DAPs for new languages within the product line.
Relevance: Developing a core reusable base for LSP and DAP functionalities through Neverlang's LPL features will streamline the creation of new language support. This fosters a more efficient and scalable approach to LSP and DAP development, aligning perfectly with the core principles of software product lines.

Project description

Software languages, crucial not only in software engineering but also in various

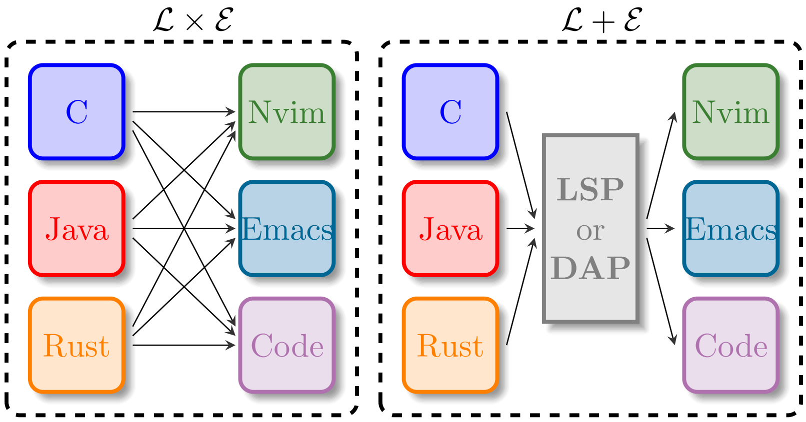

Figure 1: Traditional approach to language support in editors.

other fields [32, 10], require effective editing support for optimal use. This applies to both general-purpose languages (GPLs) and domain-specific languages (DSLs). To aid in this accomplishment, modern *Integrated Development Environments* (IDEs) and *source-code editors* (SCEs) provide a wide range of editing support (e.g., syntax and semantic highlighting, intelligent code completion, debugging, and show documentation on hovering over a primitive), but the development of such support is a complex and time-consuming task [36]. The reduction of efforts in implementing this support has paved the way for an advantageous strategy for programming language developers and maintainers, as well as those developing integration tools, when an IDE would have provided the implementation for their language and vice-versa. Then, given $\mathcal{L}$ languages and $\mathcal{E}$ editors, the number of possible combinations is $\mathcal{L} \times \mathcal{E}$ for both LSP and DAP implementations, which is a large number. It means that the development of a new language or editor would require a large amount of effort to provide support for all possible combinations, with a significant amount of duplicated work and the risk of introducing inconsistencies [34].

In contemporary times, advancements in techniques [35] such as the architecture of language infrastructures [28, 41], Language Workbenches (LWBs) [13] and the implementation of specific patterns [3, 29, 31] have been made to address this issue.

In this context, Microsoft in 2016 proposed the *Language Server Protocol* and the *Debugger Adapter Protocol* for Visual Studio Code as a promising solution to this problem, reducing from $\mathcal{L} \times \mathcal{E}$ to $\mathcal{L} + \mathcal{E}$ the number of combinations to be implemented, as it decouples the implementation of the language support from the editor (see Figure 1). Detailing, the LSP and DAP are protocols that describes a common *Application Programming Interface* (API) that the **language server** (LS) should implement, with the benefit of having only one implementation of the LS and multiple clients (IDEs and SCEs) that can consume it, essentially establishing a *client-server* relationship through a communication channel (e.g., *pipes* or *sockets*). owever, the implementation of an LS and its integration with an IDE/SCEs is still a complex task, as it requires the knowledge of the LSP specification and the implementation of the language support. The imple-

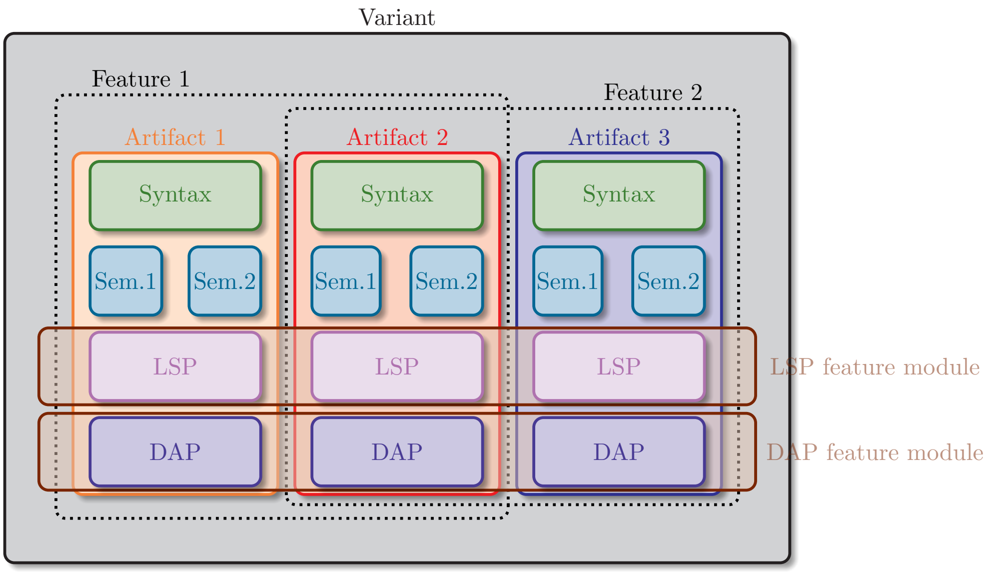

Figure 2: Proposed approach to modular implementation of LSP and DAP.

mentation [16] of an LS is done entirely manually and it is a *top-down* activity, where most of the time is spent on the design and implementation data structures and algorithms. Recently, researchers have started talking about the *Software Product Lines* (SPLs) [5, 14] to move towards a more modular world, where the implementation of a software system can be done in a compositional way, by composing the features of the SPL. When a SPL is applied to the implementation of a programming language, each product corresponds to a language variant [24] taking the name of *Language Product Lines* (LPLs) [24]. LPLs have been successfully used in both GPLs [9, 23, 24] and DSLs [17, 27, 39, 44, 45].

What I want to prove with this project is that the implementation of an LS could be a *bottom-up* activity, where each LSP or DAP functionality can be see as a separate *feature module* [4, 18] splitted across the language artifacts, where each artifacts can be part of one or more *features* (see Figure 2). These units can be composed to provide a modular implementation of the LS. This approach is supported by the fact that the LSP and DAP are *language-agnostic* protocol [30, 36], which means that it does not impose any restrictions on the implementation of the LS, as long as it respects the specification of the protocol. In *feature-oriented programming* (FOP) [1, 11, 33], a feature module is a unit of composition that encapsulates a specific functionality, and it is a first-class entity that can be composed with other feature modules to form a software system; similar to an aspect module that encapsulates a crosscutting concern in *aspect-oriented programming* (AOP) [20, 21, 26]. So, proponing a new modular approach to the implementation of an LS, based also on FOP, I want to extend Neverlang Language Workbench [38, 40] in order to give support to the implementation of the LS for each semantic action of the language, and I will also implement the Neverlang LSP [25] and DAP to support the composition of the LS feature modules. In this way, the implementation of the LS is a *bottom-up* activity, where each semantic action has attached a feature module that implements the LS functionality for that action, and these units can be composed to provide a modular implementation of the LS. Each feature module is written using a DSL, developed in the context of the Neverlang framework, that is specific for the implementation of the LS, and it is independent from the language for which the LS is being implemented. Furthermore, with this approach, we want to prove that it is possible to reduce the number of combinations from $\mathcal{L} + \mathcal{E}$ to $\mathcal{L} \times 1$ by generating client implementations.

Methodology

The first step involves defining feature modules, which are essential components that encapsulate different functionalities of Language Server Protocol (LSP) and Debug Adapter Protocol (DAP). These functionalities include syntax highlighting, code completion, debugging, and documentation support. Each feature module is identified and defined based on its specific role within the LSP and DAP ecosystem.

Following the identification of feature modules, the next phase is developing domain-specific languages (DSLs) within Neverlang. These DSLs are tailored to facilitate the development and composition of the feature modules, providing a structured and efficient way to create and manage them.

Once the feature modules are defined and the DSLs are developed, the next step is to implement a system within Neverlang that allows for the composition of these feature modules. This system enables the integration of various feature modules into a complete and functional Language Server.

With the modular framework in place, the next phase involves developing Language Servers for multiple programming languages. This step demonstrates the reuse and compositional capabilities of the feature modules. By leveraging the modular design, Language Servers for different languages can be developed more efficiently and with greater consistency.

To ensure the effectiveness of these Language Servers, their performance and integration within different Integrated Development Environments (IDEs) and Source Code Editors (SCEs) will be evaluated. This evaluation will focus on how well the Language Servers perform in real-world development environments and how seamlessly they integrate with existing tools. Comparison and Analysis

The final phase of the methodology involves a comprehensive comparison and analysis. This includes evaluating the effort and complexity involved in the modular approach compared to traditional top-down methods. By analyzing the development process, the benefits and challenges of using a modular framework can be assessed.

Additionally, the maintainability and extensibility of the modular approach will be scrutinized. This involves introducing changes and enhancements to the Language Servers and observing how easily these modifications can be implemented. The goal is to determine whether the modular approach offers superior maintainability and extensibility compared to traditional methods.

Expected Contributions

- **A Modular Framework for Language Server Development**: A comprehensive framework within the Neverlang Language Workbench that supports the modular development of Language Servers.

- **Reduction in Development Effort**: Empirical evidence demonstrating a reduction in the development effort and complexity associated with implementng Language Servers.
- **Reusable and Language-Agnostic Modules**: A library of reusable, language-agnostic feature modules for common LSP and DAP functionalities.
- **Case Studies and Practical Applications**: Detailed case studies showcasing the practical applications of the modular approach across different programming languages and development environments.

Timeline of 36 months

- **Months 1-6**: Literature review on Language Servers, Language Server Protocol, Debug Adapter Protocol, Feature-Oriented Programming, and Software Product Lines.
- **Months 7-12**: Design and development of feature modules for LSP and DAP functionalities within the Neverlang framework.
- **Months 13-18**: Implementation of DSLs for LSP and DAP development, enabling the creation of feature modules.
- **Months 19-24**: Development of a modular system for composing feature modules and generating Language Servers.
- **Months 25-30**: Implementation of Language Servers for multiple programming languages using the modular framework.
- **Months 31-36**: Evaluation of the modular approach, comparison with traditional methods, and analysis of the benefits and challenges.

Conclusion

The proposed modular approach to implementing Language Servers via feature-oriented programming within the Neverlang Language Workbench represents a significant advancement in reducing the complexity and effort associated with developing Language Servers. By decomposing the LS functionalities into reusable and composable feature modules, this approach promises to enhance maintainability, extensibility, and overall efficiency in the development of language support tools. This research will contribute valuable insights and practical solutions to the field of programming language implementation and development environment integration.

References

[1] Sven Apel, Alexander von Thein, Philipp Wendler, Armin Größlinger, and Firk Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In Betty H. Chang and Klaus Pohl, editors, *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, pages 482–491, San Francisco, CA, USA, may 2013. IEEE.

[2] Djonathan Barros, Sven Peldszus, Wesley K. G. Assunção, and Thorsten Berger. Editing Support for Software Languages: Implementation Practices in Language Server Protocols. In Manuel Wimmer, editor, *Proceedings of the 25th International Conference on Model Driven Engineering Langauges and Systems (MoDELS'22)*, pages 232–243, Montréal, Canada, October 2022. ACM.

[3] Bas Basten, Jeroen van den Bos, Mark Hills, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen Vinju. Modular Language Implementation in Rascal—Experience Report. *Science of Computer Programming*, 114:7–19, December 2015.

[4] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004.

[5] Francesco Bertolotti, Walter Cazzola, and Luca Favalli. SP⅃LꟼƧ: Software Product Lines Extraction Driven by Language Server Protocol. *Journal of Systems and Software*, 205, November 2023.

[6] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. PACKT Publishing Ltd, August 2013.

[7] Hendrik Bünder. Decoupling language and editor-the impact of the language server protocol on textual domain-specific languages. In *MODELSWARD*, pages 129–140, 2019.

[8] Walter Cazzola and Luca Favalli. Towards a Recipe for Language Decomposition: Quality Assessment of Language Product Lines. *Empirical Software Engineering*, 27(4), April 2022.

[9] Walter Cazzola and Diego Mathias Olivares. Gradually Learning Programming Supported by a Growable Programming Language. *IEEE Transactions on Emerging Topics in Computing*, 4(3):404–415, September 2016. Special Issue on Emerging Trends in Education.

[10] Alain Colmerauer. An introduction to prolog iii. *Communications of the ACM*, 33(7):69–90, 1990.

[11] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Using Feature Models. In David Weiss and Rob van Ommering, editors, *Proceedings of the 3rd International Conference on Software Product-Line (SPLC'04)*, Lecture Notes in Computer Science 3154, pages 266–283, Boston, MA, USA, August-September 2004. Springer.

[12] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: a Meta-Language for Modular and Reusable Development of DSLs. In Davide Di Ruscio and Markus Völter, editors, *Proceedings of the 8th International Conference on Software Language Engineering (SLE'15)*, pages 25–36, Pittsburgh, PA, USA, October 2015. ACM.

[13] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerrtsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen

Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, and Eelco Visser. The State of the Art in Language Workbenches. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Proceedings of the 6th International Conference on Software Language Engineering (SLE'13)*, Lecture Notes on Computer Science 8225, pages 197–217, Indianapolis, USA, October 2013. Springer.

[14] Luca Favalli, Thomas Kühn, and Walter Cazzola. Neverlang and FeatureIDE Just Married: Integrated Language Product Line Development Environment. In Philippe Collet and Sarah Nadi, editors, *Proceedings of the 24th International Software Product Line Conference (SPLC'20)*, pages 285–295, Montréal, Canada, 19th-23rd of October 2020. ACM.

[15] Martin Fowler and Rebecca Parsons. *Domain Specific Languages*. Addison Wesley, September 2010.

[16] Nadeeshaan Gunasinghe and Nipuna Marcus. *Language server protocol and implementation : supporting language-smart editing and programming tools*. Apress, Berkeley, Calif, 2022.

[17] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. Adding Standardized Variability to Domain Specific Languages. In Klaus Pohl and Birgit Geppert, editors, *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*, pages 139–148, Limerick, Ireland, September 2008. IEEE.

[18] Christian Kästner, Sven Apel, and Klaus Ostermann. The Road to Feature Modularity? In Ina Schäfer, Isabel John, and Klaus Schmid, editors, *Proceedings of the 3rd Workshop on Feature-Oriented Software Development (FOSD'11)*, Münich, Germany, August 2006. ACM.

[19] Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In Martin Rinard, Kevin J. Sullivan, and Daniel H. Steinberg, editors, *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*, pages 444–463, Reno, Nevada, USA, October 2010. ACM.

[20] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeff Palm, and Bill Griswold. An Overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, LNCS 2072, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[21] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th European Conference on Object Oriented Programming (ECOOP'97)*, Lecture Notes in Computer Science 1241, pages 220–242, Helsinki, Finland, June 1997. Springer-Verlag.

[22] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, September 2010.

[23] Thomas Kühn and Walter Cazzola. Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines. In Rick Rabiser and Bing Xie, editors, *Proceedings of the 20th International Software Product Line Conference (SPLC'16)*, pages 50–59, Beijing, China, 19th-23rd of September 2016. ACM.

[24] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. Choosy and Picky: Configuration of Language Product Lines. In Goetz Botterweck and Jules White, editors, *Proceedings of the 19th International Software Product Line Conference (SPLC'15)*, pages 71–80, Nashville, TN, USA, 20th-24th of July 2015. ACM.

[25] Thomas Kühn, Walter Cazzola, Nicola Pirritano Giampietro, and Massimiliano Poggi. Piggyback IDE Support for Language Product Lines. In Thomas Thüm and Laurence Duchien, editors, *Proceedings of the 23rd International Software Product Line Conference (SPLC'19)*, pages 131–142, Paris, France, 9th-13th of September 2019. ACM.

[26] Ramnivas Laddad. *AspectJ in Action: Pratical Aspect-Oriented Programming*. Manning Pubblications Company, 2003.

[27] Jörg Liebig, Rolf Daniel, and Sven Apel. Feature-Oriented Language Families: A Case Study. In Philippe Collet and Klaus Schmid, editors, *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13)*, Pisa, Italy, January 2013. ACM.

[28] Ralf Lämmel. *Software Languages*. 01 2018.

[29] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain Specific Languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[30] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. 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*, Onward! 2020, page 1–17, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 1st edition, 2009.

[32] Christine Paulin-Mohring. Inductive definitions in the system coq rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, pages 328–345. Springer, 1993.

[33] Christian Prehofer. Feature-Oriented Programming: A New Way of Object Composition. *Concurency and Computation: Practice and Experience*, 13(6):465–501, May 2001.

[34] Jonas Kjær Rask, Frederik Palludan Madsen, Nick Battle, Hugo Daniel Macedo, and Peter Gorm Larsen. The specification language server protocol: A proposal for

standardised LSP extensions. In José Proença and Andrei Paskevich, editors, *Proceedings of the 6th Workshop on Formal Integrated Development Environment, F-IDE@NFM 2021, Held online, 24-25th May 2021*, volume 338 of *EPTCS*, pages 3–18, 2021.

[35] Jonas Kjær Rask, Frederik Palludan Madsen, Nick Battle, Hugo Daniel Macedo, and Peter Gorm Larsen. Visual studio code vdm support. In *Proceedings of the 18th International Overture Workshop*, pages 35–49, 2021.

[36] Roberto Rodriguez-Echeverría, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. An LSP Infrastructure to Build EMF Language Servers for Web-Deployable Model Editors. In Regina Hebig and Thorsten Berger, editors, *Proceedings of the 2nd International Workshop on Model-Driven Engineering Tools (MDE-Tools'18)*, pages 1–10, Copenhage, Denmark, October 2018. CEUR.

[37] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. 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*, MODELS '18, page 370–380, New York, NY, USA, 2018. Association for Computing Machinery.

[38] Edoardo Vacchi and Walter Cazzola. Neverlang: A Framework for Feature-Oriented Language Development. *Computer Languages, Systems & Structures*, 43(3):1–40, October 2015.

[39] Edoardo Vacchi, Walter Cazzola, Benoît Combemale, and Mathieu Acher. Automating Variability Model Inference for Component-Based Language Implementations. In Patrick Heymans and Julia Rubin, editors, *Proceedings of the 18th International Software Product Line Conference (SPLC'14)*, pages 167–176, Florence, Italy, 15th-19th of September 2014. ACM.

[40] Edoardo Vacchi, Diego Mathias Olivares, Albert Shaqiri, and Walter Cazzola. Neverlang 2: A Framework for Modular Language Implementation. In *Proceedings of the 13th International Conference on Modularity (Modularity'14)*, pages 23–26, Lugano, Switzerland, 22nd-25th of April 2014. ACM.

[41] Markus Voelter. *DSL Engineering*. CreateSpace Independent Publishing, January 2013.

[42] Markus Völter. Language and IDE Modularization and Composition with MPS. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Proceedings of the 4th International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'11)*, Lecture Notes in Computer Science 7680, pages 383–430, Braga, Portugal, July 2011. Springer.

[43] Markus Völter and Vaclav Pech. Language Modularity with the MPS Language Workbench. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pages 1449–1450, Zürich, Switzerland, June 2012. IEEE.

[44] Christian Wende, Nils Thieme, and Steffen Zschaler. A Role-Based Approach towards Modular Language Engineering. In Mark van den Brand, Dragan Gašević,

and Jeff Gray, editors, *Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09)*, Lecture Notes in Computer Science 5969, pages 254–273, Denver, CO, USA, October 2009. Springer.

[45] Jules White, James H. Hill, Jeff Gray, Sumant Tambe, Aniruddha Gokhale, and Douglas C. Schmidt. Improving Domain-specific Language Reuse with Software Product-Line Configuration Techniques. *IEEE Software*, 26(4):47–53, July-August 2009.