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

The rapid evolution of software development necessitates efficient tools for creating and integrating programming languages. *Integrated Development Environments* (IDEs) and *source-code editors* (SCEs) offer vital support features like syntax highlighting, code completion, and debugging, but their development is often complex and labor-intensive. **Language Server Protocol** (LSP) and **Debugger Adapter Protocol** (DAP) were introduced to simplify this process by providing a standardized API, decoupling language support implementation from specific editors decreasing the number of combinations from $\mathcal{L} \times \mathcal{E}$ to $\mathcal{L} + \mathcal{E}$. Despite these advancements, the integration of LSP and DAP remains challenging due to fragmented and inconsistent approaches, also caused by it is difficult to reuse IDE implementations across multiple languages. Modern language workbenches have made strides in modularization, composition, and IDE integration. However, their methods for LSP and DAP generation often lack a standardized and cohesive framework, resulting in increased complexity and reduced efficiency. By leveraging techniques like feature-oriented programming and software product lines (SPLs), there is potential to enhance modularity and reusability in language server development. This approach promotes a *bottom-up* methodology where LSP and DAP functionalities are encapsulated in feature modules, enabling a more compositional and efficient implementation process. Nowadays, *Xtext* [6] is one of the few language workbenches that support LSP generation [2]. **Neverlang**, developed at the `ADAPT-Lab` of the Università degli Studi di Milano, being a framework for language composition and modularization, presents a promising solution. By extending its capabilities to support a universal LSP and DAP, reusable, language-agnostic feature modules can be created. This approach aims to reduce development effort and complexity compared to traditional *top-down* methods. Empirical evidence suggests that a modular framework could significantly improve maintainability, extensibility, and productivity in software [42]. By applying similar strategies to language support tools, one could achieve similar improvements in language

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

Table 1: Comparison of language workbenches in terms of modularization, precompiled feature support, native IDE generation, LSP generation, and LSP modularization. The ● symbol indicates full support, ○ no support, ◐ limited support, ✯my expected contribution and ✰ my expected contribution that can be extended to all LWs that support at least component modularization.

development tools. Additionally, this project aims to reduce to $\mathcal{L} \times 1$ the number of combinations required to support $\mathcal{L}$ languages.

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

The primary goal of this project is to develop a Universal **Language Server Protocol**[1] (LSP) and **Debugger Adapter Protocol**[2] (DAP) for modular language workbenches (LWs). 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 [23], Spoofax [20], and MPS [47, 48] have made significant strides in supporting modularization, composition, and IDE integration. The table 1 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, ○ no support, ◐ limited support and ✯my expected contribution, which can be extended to all LWs that support at least component modularization (identified by ✰). The second column indicates the level of support for modularization of artifacts and language features (more detail in Project Description section). The third column indicates the level of support for precompiled features, the importance of this feature lies in the fact that an artifact can be used by several features being compiled once, and that one feature can be used among several projects without the recompilation step [28]. The fourth column indicates the level of support for native IDE generation, this is because many LWs are supported by the existence of some IDE and thus allow IDE generation for languages developed for the IDE that hosts them. The generation and modularizazion of LSP and DAP is trivially shown by the fifth and sixth columns, respectively. However, their approaches are often fragmented and lack a standardized method for LSP and DAP generation and modularization, as shown in Table 1.

[1]https://microsoft.github.io/language-server-protocol
[2]https://microsoft.github.io/debug-adapter-protocol

Neverlang [43, 45], 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 [25, 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. Moreover, the reusability of modules across different languages is a key feature of some LWs. By developing a core reusable base for LSP and DAP, the project will establish a foundation for creating new languages and features more efficiently. This will enable developers to leverage existing modules and components to build LSPs and DAPs for new languages more quickly and effectively, reducing development time and effort significantly.

In the following, the **Research Objectives** (RO) are outlined, along with their **relevance** in the context of the state of the art. The relevance of each RO is discussed in terms of **Research Questions** (RQs) that will guide the investigation and development of the proposed universal LSP and DAP for modular language workbenches.

RO 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 some language workbenches like Xtext [6] support LSP generation [2], many do not, limiting their usability across different editors and IDEs.
RQ 1.1: How can IDE generation be improved to support LSP and DAP?
RQ 1.2: What are the key challenges in generating LSP and DAP for different programming languages?
RQ 1.3: How can a universal LSP and DAP be developed to support multiple languages and IDEs?

RO 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.
RQ 2.1: How can LSP and DAP modularization be facilitated in language workbenches?
RQ 2.2: What are the key challenges in modularizing LSP and DAP for different programming languages?
RQ 2.3: How can LSP and DAP modularization be integrated with existing language composition and modularization features in language workbenches?

RO 3: Reduce to $\mathcal{L} \times 1$ 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 $\mathcal{L}$ languages in $\mathcal{L} \times \mathcal{E}$, where $\mathcal{E}$ is the number of editors. Currently, the number of combinations to support $\mathcal{L}$ languages is $\mathcal{L} + \mathcal{E}$ [40], as the Microsoft LSP and DAP are editor-agnostic, as shown in Figure 1. This project aims to reduce the number of combinations to $\mathcal{L} \times 1$, by developing a universal LSP and DAP that can be used across different programming languages and IDEs.

[3]https://di.unimi.it/it/ricerca/risorse-e-luoghi-della-ricerca/laboratori-di-ricerca/adapt-lab

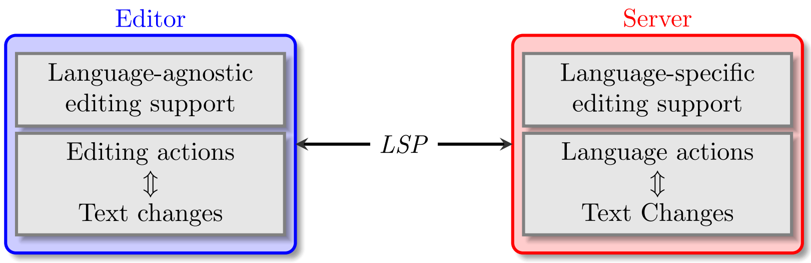
Editor
Language-agnostic editing support
Editing actions ⇕ Text changes
LSP
Server
Language-specific editing support
Language actions ⇕ Text Changes

Figure 1: LSP and DAP approach for programming languages (From [40]).

RQ 3.1: How can the number of combinations required to support multiple languages be reduced to $\mathcal{L} \times 1$?
RQ 3.2: In what ways does simplifying the development process for language support enhance efficiency?
RQ 3.3: How does reducing combinations impact the speed and effectiveness of creating language support?

RO 4: Leverage Neverlang for LSP and DAP in 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 variants 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.
RQ 4.1: How can Neverlang's LPL development features be leveraged for creating a reusable core for LSP and DAP functionalities?
RQ 4.2: What are the key benefits of using Neverlang for LSP and DAP development in the context of LPLs?
RQ 4.3: How does leveraging Neverlang's LPL features enhance the scalability and efficiency of LSP and DAP development?

Project description

Software languages, crucial not only in software engineering but also in various other fields [34, 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 [39]. 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 $\mathcal{L} \times \mathcal{E}$, which is a large number. It means that the development of a new language or

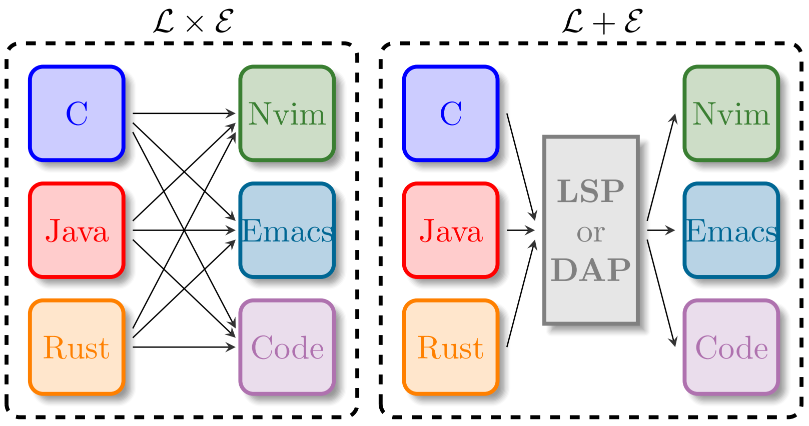

Figure 2: Traditional approach vs LSP/DAP approach to language support.

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 [37].

In contemporary times, advancements in techniques [38] such as the architecture of language infrastructures [30, 46], Language Workbenches (LWBs) [13] and the implementation of specific patterns [3, 31, 33] have been made to address this issue.

In 2016, Microsoft 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 2). 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*). However, 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 implementation [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. Since 1990s [18], 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 [25] taking the name of *Language Product Lines* (LPLs) [25]. LPLs have been successfully used in both GPLs [9, 24, 25] and DSLs [17, 44, 50].

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, 19] splitted across the language artifacts, where each artifacts can be part of one or more *language features* (see Figure 3). 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 [32, 39] (see Fig. 1), 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)

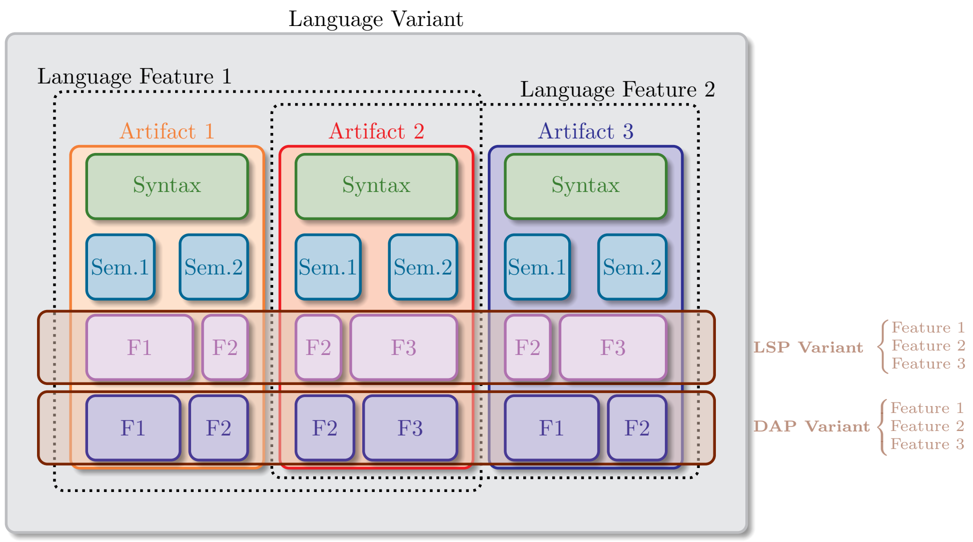

Figure 3: Proposed approach to modular implementation of LSP and DAP.

[1, 11, 36], 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) [21, 22, 27]. Using FOP in language development, a family of languages [29] can be defined by composing feature modules [49], and a language can be seen as a product of the family. So, proposing a new modular approach to the implementation of an LS, based also on FOP, I want to extend Neverlang Language Workbench [43, 45] in order to give support to the implementation of the LS for any artifact of the language, and I will also implement the Neverlang LSP [26] 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 artifact has attached a part of LSP and DAP feature module that implements the LS functionality for that *language fragment*, and these units can be composed to provide a modular implementation having **variants** of the LS. I will also make it possibile write feature modules using DSLs. These DSLs will be developed in the context of the Neverlang framework, and will be specific for the implementation of the LS, and trivially they will be independent from the language for which the LS is being implemented. Furthermore, with this approach, I 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; this will be done by implementing a *client generator* that will take as input the LS feature module and will produce the client implementation. This will be supported by the implementation of a *client language* that will allow the developer to specify which client to generate.

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 a **modular framework** within the Neverlang Language Workbench to support the implementation of these feature modules. This framework will provide the necessary infrastructure for creating, composing, and managing the feature modules effectively. This phase will involve designing and implementing the necessary data structures and innovative algorithms to support the composition. An imporant step 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 (**LSP/DAP variant** in Fig. 3). 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. One very important aspect that would have been possible to guess is that it is actually possible to **further reduce** the number of combinations. Due to the modularization and *splitting* of the feature implementation between **artifacts** it is possible to get down to $\mathcal{N} \times 1$ combinations, where $\mathcal{N} < \mathcal{L}$. This is possible because the *freature artifacts* are self-contained in a specific language artifact. 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. The final phase of the methodology involves a comprehensive comparison and analysis through a several metrics, case studies and user feedbacks. 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 Design**: An abstract comprehensive framework within the Neverlang Language Workbench that supports the modular development of Language Servers.
- **The Implementation of the Modular Framework**: A concrete implementation of the modular framework that enables the development and composition of feature modules for 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.

- **Evaluation and Comparison**: A comprehensive evaluation and comparison of the modular approach with traditional top-down methods, highlighting the benefits and challenges of each approach.
- **Generalization to other Language Workbenches and Research Areas**: A generalization of the modular approach to other Language Workbenches and research areas, demonstrating its applicability and effectiveness in various contexts.

Timeline

Figure 4 shows the proposed timeline for the research project. The project is divided into seven main phases:

- Literature Review
- Design and development of feature modules
- DSLs implementation for LSP and DAP
- Composing fature modules within Neverlang
- Universal Clients Generation
- Testing modularization with 3 LSs
- Evaluation comparison and analysis
- Generalization to other Language Workbenches and Research Areas

The **literature review** phase will be carried out in the first six months. I will start by expanding my knowledge LSP and DAP in general, and then I will perform a deeper study of all the most important approaches currently available in literature to elaborate on their pros and cons and lay a groundwork for my research work. Great attention would be given to the study of *bottom-up* and *top-down* approaches, in order to find their shared aspects. This process will lead to the drafting of a survey on feature-oriented programming [35] and software product lines.

In the following eight months, I will **design** and **develop the feature modules** for the LS, and I will extend the Neverlang framework to support the implementation of the LS feature modules. This will be supported by the implementation of generic data structures, such as *Indexed Trees*, *Dependency Graphs*, and *Symbol Tables*, that will be populated by any given language artifact not known *a priori*. An additional compilation step will be added to the Neverlang framework to generate the feature modules from the language artifacts. At the end of this phase, it should be possible answer the RQ 1.1, RQ 1.2 and RQ 1.3 research questions, so achieving the first research objective (**RO 1**).

During the last two months of *design and development* and for 8 months, I will implement the **DLSs for the LSP and DAP**, and I will extend the Neverlang framework to support the composition of the LS feature modules through the DLSs. This will be supported by the implementation of a *multi-dimensional variability model* [41]. RQ 2.1 will be answered at the end of this phase.

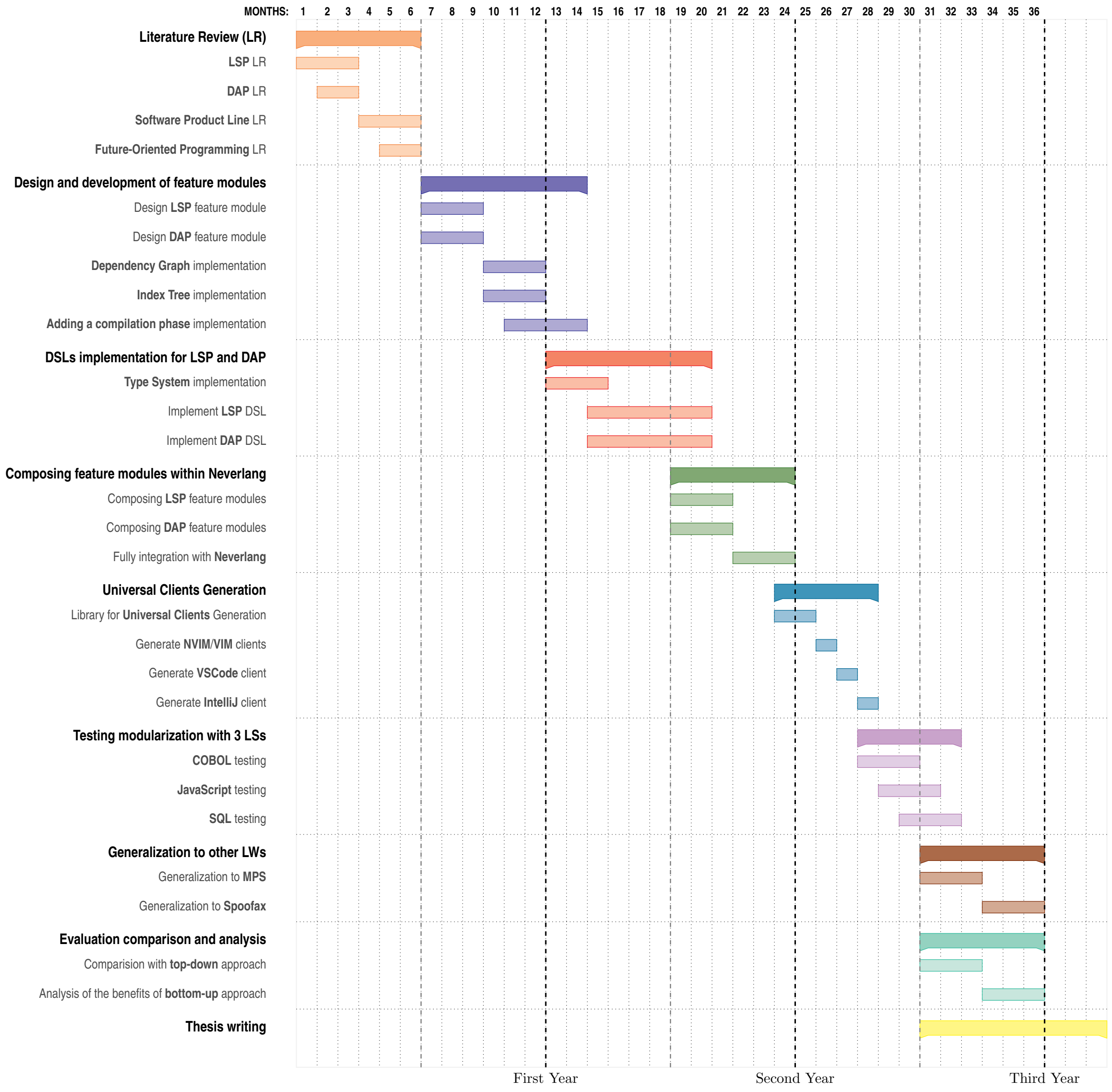

Figure 4: Proposed timeline for the research project.

One of the biggest challenges, in the next six months, will be to **compose the feature modules**. This will be done by implementing a *composition algorithm* that will take as input the splitted feature modules and the language artifacts and will produce the LS feature module. The second research objective (**RO 2**) will be achieved by answering the RQ 2.2 and RQ 2.3 research questions.

The following six months will be dedicated to the **universal clients generation**. This will be done by implementing a *client generator* that will take as input the LS feature module and will produce the client implementation; answering the RQ 3.1 research question. This will be supported by the implementation of a *client language* that will allow the developer to specify which client to generate and will be extensible to support new clients; answering the RQ 3.2. It will be implemented the client generator for different IDEs and SCEs, such as Visual Studio Code, Vim/Nvim and IntelliJ IDEA.

Then, I will test the modularization with three LSs, evaluating the feasibility of the approach. This will be done by **implementing the LS for three different languages** and by generating the client implementations. The third research objective (**RO 3**) will be achieved by answering the RQ 3.3 research question.

In the last months, the **evaluation** will focus on the effort and complexity involved in the development of the LS, the maintainability and extensibility of the LS, and the integration of the LS with the existing tools. The evaluation will also include a comparison with the traditional approach to LS development, to assess the benefits and challenges of using the modular framework. Simultaneously, I will work on the **generalization** of the modular approach to other Language Workbenches and research areas, demonstrating its applicability and effectiveness in various contexts. This will be done by implementing the modular framework in other Language Workbenches and by applying it to other research areas, such as software product lines and software architecture.

Regarding the fourth research objective (**RO 4**), it will be achieved by answering the relative research questions (RQ 4.1, RQ 4.2, RQ 4.3), in the course of whole project.

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. Considering the potential impact of this research, I am confident that the proposed project will yield valuable contributions to the field of programming language development and integration, especially DSLs. By providing a modular framework for the implementation of Language Servers, this research has the potential to improve the way language support tools are developed and maintained. The reduction in development effort, the increased reusability of feature modules, and the improved maintainability and extensibility of Language Servers are just a few of the benefits that this research aims to deliver.

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] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, November 1990.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] Manuel Leduc, Thomas Degueule, Eric Van Wyk, and Benoît Combemale. The Software Language Extension Problem. *Software and Systems Modeling*, 19(2):263–267, January 2020.

[29] 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.

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

[31] 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.

[32] 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.

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

[34] 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.

[35] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, Lecture Notes in Computer Science 1241, pages 419–443, Helsinki, Finland, June 1997. Springer.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] Marko Rosenmüller, Norbert Siegmund, Thüm, and Gunter Saake. Multi-Dimensional Variability Modeling. In Krzysztof Czarnecki and Ulrich W. Eisenecker, editors, *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS'11)*, pages 11–20, Namur, Belgium, January 2011. ACM.

[42] Hongyi Sun, Waileung Ha, Pei-Lee Teh, and Jianglin Huang. A case study on implementing modularity in software development. *Journal of Computer Information Systems*, 57(2):130–138, 2017.

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

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.