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 needs 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 $\mathbf{L} \times \mathbf{E}$ to $\mathbf{L} + \mathbf{E}$. 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 functionality are encapsulated in feature modules, enabling a more compositional and efficient implementation process. Nowadays, *Xtext* [**?**] is one of the few language workbenches that support LSP generation [**?**]. **Neverlang**, developed at the `ADAPT-Lab` of the Universitá degli Studi di Milano, is a language workbench focusing on modularity and composition of language features enhancing their reuse.

By extending its capabilities to support a universal LSP and DAP, reusable, language-agnostic feature modules for LSP and DAP would be feasible.. 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 [**?**]. By applying similar strategies to language support tools, one could achieve similar improvements in language development tools.

Additionally, this project aims to reduce to $\mathbf{L} \times 1$ the number of combinations required to support $\mathbf{L}$ languages. This is important because the number of languages is growing, and the number of editors is

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, ⍛ fine-grained modularization, ⍜ coarse-grained modularization, ✯my expected contribution and ☆ my expected contribution that can be extended to all LWs that support at least component modularization.

limited. By reducing the number of combinations, the effort required to support new languages is reduced, and the quality of language support tools can be improved.

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 Neverlang [**?**], Melange [**?**], MontiCore [**?**], Spoofax [**?**], and MPS [**?**, **?**] have made significant strides in supporting modularization, composition, and IDE integration. The table **??** 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, ⍛ fine-grained modularization, ⍜ coarse-grained modularizationand, ✯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), as can be see in the table **??**, Neverlang is the only LW that supports fine-grained modularization, which is essential for the development of language product lines [**?**, **?**] (LPLs). 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 [**?**]. 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

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

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 **??**, also due to their coarse-grained modularity. Neverlang [**?**, **?**], developed at the `ADAPT-Lab`[3] of the Università degli Studi di Milano, being a comprehensive framework for language composition and fine-grained modularization [**?**, **?**], 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 [**?**] (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 [**?**] support LSP generation [**?**], 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 [**?**]. 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 $\mathbf{L} \times 1$ the number of combinations to support 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 $\mathbf{L}$ languages in $\mathbf{L} \times \mathbf{E}$, where $\mathbf{E}$ is the number of editors. Currently, the number of combinations to support $\mathbf{L}$ languages is $\mathbf{L} + \mathbf{E}$ [**?**], as the Microsoft LSP and DAP are editor-agnostic, as shown in Figure **??**. This project aims to reduce the number of combinations to $\mathbf{L} \times 1$, by developing a universal LSP and DAP that can be used across different programming languages and IDEs.
RQ 3.1: How can the number of combinations required to support multiple languages be reduced to $\mathbf{L} \times 1$?
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?

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

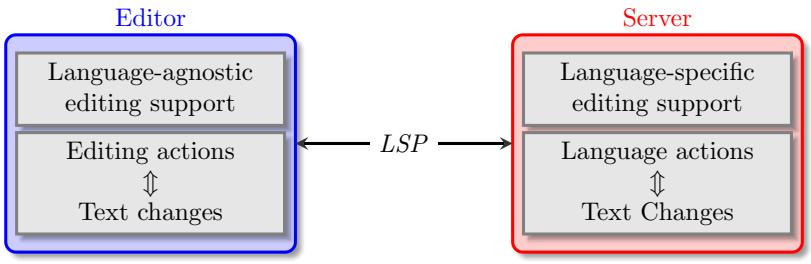

Figure 1: LSP and DAP approach for programming languages (From [**?**]).

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 [**?**], 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 [**?**, **?**], 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 [**?**]. 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 **L** languages and **E** editors, the number of possible combinations is **L** × **E**, 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 [**?**].

In contemporary times, advancements in techniques [**?**] such as the architecture of language infrastructures [**?**, **?**], Language Workbenches (LWBs) [**?**] and the implementation of specific patterns [**?**, **?**, **?**] 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 **L** × **E** to **L** + **E** the number of combinations to be implemented, as it decouples the implementation of the language support from the editor (see Figure **??**). 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 [**?**] 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 [**?**], researchers have started talking about the *Software Product Lines* (SPLs) [**?**, **?**] 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 [**?**] taking the name of *Language Product Lines* (LPLs) [**?**]. LPLs have been successfully used in both GPLs [**?**, **?**, **?**] and DSLs [**?**, **?**, **?**].

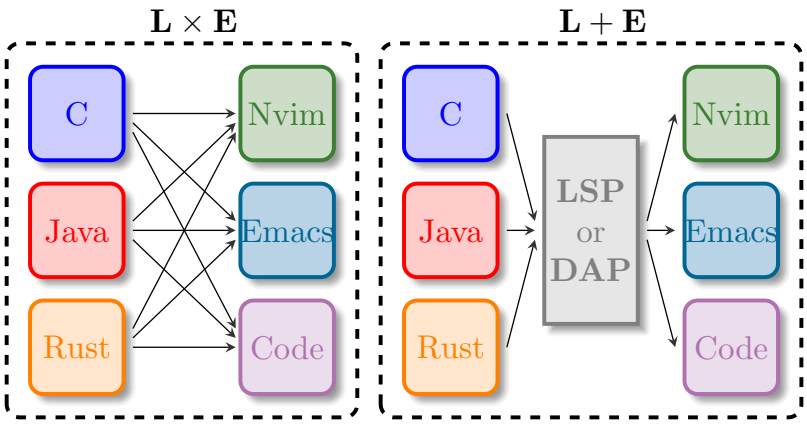

Figure 2: Traditional approach vs LSP/DAP approach to language support.

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 seen as a separate *feature module* [**?**, **?**] splitted across the language artifacts, where each artifacts can be part of one or more *language features* (see Figure **??**). These units can be composed to provide a modular implementation of the LS. This approach is supported by the fact that the LSP and the DAP are *language-agnostic* protocols [**?**, **?**] (see Fig. **??**), 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) [**?**, **?**, **?**], 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) [**?**, **?**, **?**]. Using FOP in language development, a family of languages [**?**] can be defined by composing feature modules [**?**], and a language can be seen as a product of the family. So, by proposing a new modular approach to the implementation of an LS, based also on FOP, I want to extend Neverlang Language Workbench [**?**, **?**] to give support to the implementation of the LS for any artifact of the language, and I will also implement the Neverlang LSP [**?**] 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 $\mathbf{L} + \mathbf{E}$ to $\mathbf{L} \times 1$ 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 that encapsulate different functionality of Language Server Protocol (LSP) and Debug Adapter Protocol (DAP). These functionality include syntax highlighting, code completion, debugging, and documentation support. Each feature module is identified and defined

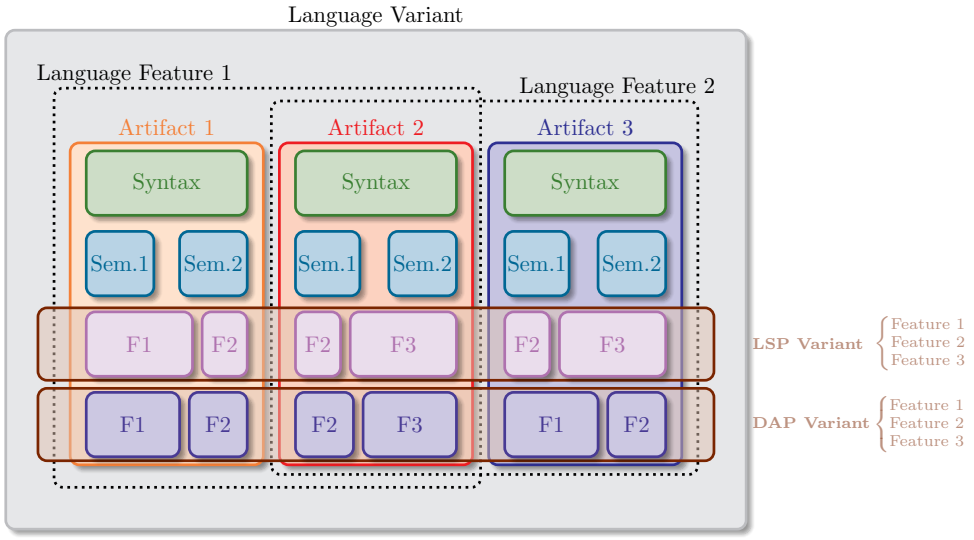

Figure 3: Proposed approach to modular implementation of LSP and DAP.

based on its specific role within the LSP and DAP ecosystem. Following the identification of feature modules, the next phase is to develop 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 important step is to develop sime 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. **??**). 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 $\mathbf{N} \times 1$ combinations, where $\mathbf{N} < \mathbf{L}$. This is possible because the *feature artifacts* are self-contained in a specific language artifact.

To evaluate the effectiveness of Language Servers, their performance and integration within various IDEs and Source Code Editors (SCEs) will be assessed. This evaluation will focus on their performance in real-world development environments and how seamlessly they integrate with existing tools. The final phase involves a comprehensive comparison through metrics, case studies, and user feedback. This includes assessing the **effort and complexity** of the modular approach versus traditional methods. By analyzing the development process, the benefits and challenges of the modular framework can be identified. Additionally, the maintainability and extensibility of the modular approach will be examined by introducing changes to the Language Servers and observing the ease of implementation. The goal is to determine if 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 functionality.
- *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

In support of this project, a paper titled **Code Less to Code More: Streamlining LSP Development for Language Families** (F. C. Bruzzone, W. Cazzola, L. Favalli) is being written for submission to **Jurnal of Systems and Software** [4] (JSS).

Figure **??** shows the proposed timeline for the research project.

The **literature review** phase will be carried out in the first six months. I will start by expanding my knowledge on LSP and DAP in general, and then I will perform a deeper study of all the most important approaches currently available in the 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 to find their shared aspects. This process will lead to the drafting of a survey on feature-oriented programming [**?**] 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 **DSLs 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* [**?**]. RQ 2.1 will be answered at the end of this phase.

One of the biggest challenges, in the next six months, will be to **compose feature modules**. This will be done by implementing a *composition algorithm* that will take as input the split feature modules and the

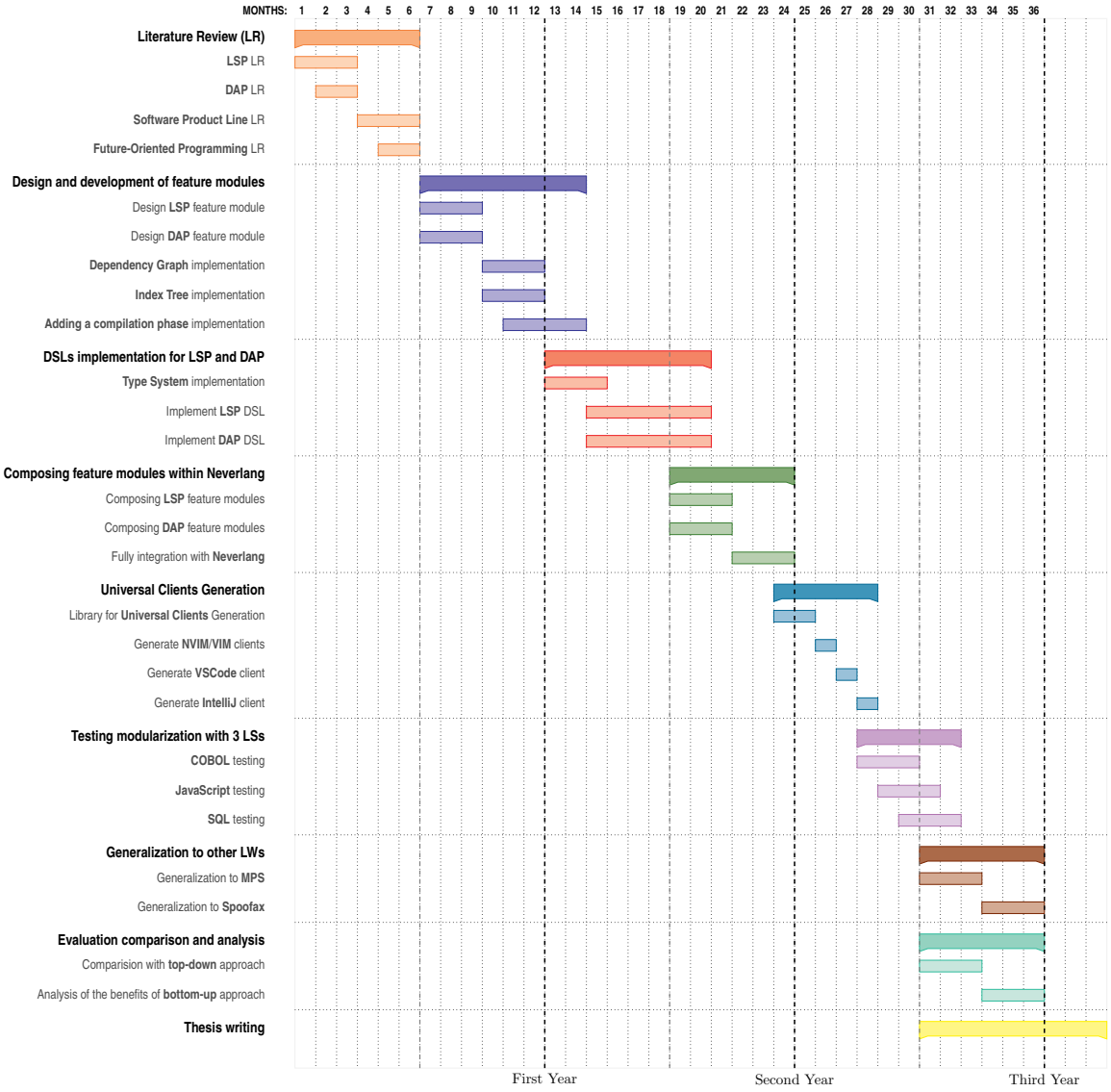

Figure 4: Proposed timeline for the research project.

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 client 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 significantly reduces the complexity and effort of developing Language Servers. By decomposing LS functionality into reusable and composable feature modules, this method enhances maintainability, extensibility, and efficiency in creating language support tools. This project promises valuable contributions to programming language development and integration, especially for DSLs. The modular framework for implementing Language Servers aims to streamline development, increase reusability, and improve the maintainability and extensibility of language support tools.

References