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

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 [**?**], MontiCore [**?**], Spoofax [**?**], and MPS [**?**, **?**] have made significant strides in supporting modularization, composition, and IDE integration. However, their approaches are often fragmented and lack a standardized method for LSP and DAP generation and modularization. Neverlang [**?**, **?**], 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 [**?**, **?**] (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 [**?**] (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 some language workbenches like Xtext [**?**] support LSP generation [**?**], 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 [**?**]. 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

[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

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 $\mathcal{O}(\mathcal{L} + \mathcal{E})$ [**?**], 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 [**?**], 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

Finding and working with the right abstractions for describing a problem or its solution is one of the central pillars of software engineering [**?**]

DSLs are programming languages with either a textual or graphical interface that, in contrast with general-purpose languages (GPLs), are designed to describe and solve problems in a specific domain. The main advantages of using a DSL over a GPL come from a communication and ease of use standpoint: many of the failures in software projects (large ones in particular) come from the difficulty to translate requirements into specifications and specifications into implementations, due to the lack of a common vocabulary. DSLs can make it easier to communicate with domain experts, providing both a description and a solution for the problem [**?**]. It is also much easier and faster to train the developers on how to use a DSL rather than a GPL while at the same time ensuring the whole team is aligned on the same style and project-specific conventions, improving maintainability. On the other hand, DLSs developing skills are hardly transferable and difficult to design and develop.

All languages, both DLSs and GPLs, tend to be designed as monolotic pieces of software with very little opportunity for reuse. More and more interest has been given to reusability in language development during the recent years. The solution suggested by Schwerdfeger and Van Wyk is a modular approach to parse table definitions: each module represents a grammar fragment that can be precompiled and distributed as a standalone product [**?**]. A complete parse table can than be obtained by combining grammar extensions into an host language. This approach leverages on the shared language concepts between different languages (such as loops, conditionals, assignments, etc.): these concepts always share the same semantics, but with a different syntax. Some of the frameworks embracing this approach are: JastAdd [**?**], xText [**?**], MontiCore [**?**] and Neverlang [**?**], each addressing this problem in a unique way.

Modular DSL development frameworks are best suited to be used along with the feature-oriented programming paradigm (FOP). FOP is a programming paradigm for generation of software product lines (SPLs), a term referring to a set of techniques and engineering methods inspired by industrial production and marketing, in which product lining is the process of offering several related products for sale individually. SPLs, in the same way, deal with software variability of products that share the same code base: by splitting a product in the features it provides, a set of slightly different but related products can be generated by a composition-based mechanism applied on the constituent features of the SPL. The same concept applied to language development goes under the name of language product line (LPL). The LPL development can be faced either in a top-down or a bottom-up approach: the former builds the feature model first by performing a variability analysis during the design phase; the latter starts from a predefined base language, decomposes it in its parts and builds the feature model starting from there. For this project, I propose the use of a bottom-up approach, to achieve better maintainability and extendibility of the language families. Neverlang is a good case-study for this task because, differently from most of the other frameworks, which have been tested applying a top-down approach, such as Spoofax [**?**], LansGems [**?**] and the aforementioned Monticore, already has its own tool for bottom-up LPL development in the form of AiDE [**?**, **?**, **?**]. AiDE currently builds the language variability model on a syntactic level, evaluating the dependencies between the Neverlang modules based on the provided and required nonterminals of their grammar fragments. AiDE can then be used to produce a language definition by activating the desired features in the variability model.

The aim of this project would be to demonstrate that the best approach to LPL development is the *multi-dimensional variability modeling*; in this approach the variability tree is replaced by a variability forest (or tree-set) in which each tree represents a different dimension of variability (or an implementation concern) [**?**]. On a high level, those dimensions can be sumarized into three groups.

- *Abstract syntax variability*: providing abstract syntax variability lets the user tune the complexity of the DSL changing amount of constructs that will be part of the language configuration. Including additional (not required) constructs to the language causes a needless increase in complexity. Constructs are usually grouped in features to facilitate their selection.

- *Concrete syntax variability*: given the same abstract syntax, the choice of a particular concrete syntax is still relevant. Depending on context and type of user, for instance, a localized version of the DSL could be needed, or a graphical interface could be more suitable than a textual one.

- *Semantic variability*: semantic variability refers to the capability of supporting different interpretations for the same construct. Some frameworks apply a role-based strategy, in which each role gives a different interpretation over the same abstract syntax.

In this project, the goal would be to produce a forest in which each tree represents either the abstract syntax, the concrete syntax or a semantic role. The concrete language feature implementation would be derived by applying a compositional algorithm over the set of selected features in the multi-dimensional variability model. In order for this to be doable, the framework must support a modularization approach in which each implementation concern is defined in a different language module. In every other case, multi-dimensional variability is just not possible.

A toolchain that supports this kind of development cycle is ASF+SDF+FeatureHouse, that uses ASF+SDF for modular language design and FeatureHouse as a languages variability management framework [**?**], but, to my knowledge, no framework or toolchain implements a bottom-up multi-dimensional variability model building algorithm based on semantic language features. I suggest that finding an euristic that reliably applies this strategy over a set of language artifacts would greatly increase the reliability and the productivity of the LPL: in the top-down approach, the feature model is built by the variability analysis performed by the human brain, which behaves on a functional (semantic) level; even though a variability model developed by a top-down approach is way harder to expand and maintain and for this reason used mainly for product lines with a few products and a limited set of features, it has the advantage of usually providing a coherent and human-readable structure. On the other hand, a syntax-based bottom-up approach scales better but has little to no knowledge of the underlying semantic features. A semantic-based variability model would be built with the goal of emulating the decision process performed by a human designer, achieving the best of both worlds.

Neverlang does not implement abstract syntax explicitly, but provides a module construct in which it is possible to define grammar fragments (that in Neverlang's syntax go under the *reference syntax* section) and, in case, one or more semantic roles. Neither the reference syntax nor the roles are considered concrete unless they are imported by a *slice*. This can be used (with surprising levels of flexibility) to greatly increase the degree of variability of the LPL: each module-defined reference syntax can both serve the use of abstract syntax and concrete syntax.

As a result, activating a feature in the abstract syntax tree would enable the possibility to activate all the compliant features in the concrete syntax tree (including the chosen abstract syntax, thing that would be impossible if the distinction between abstract and concrete syntaxes was clearly defined) and in turn the semantic roles in the relevant trees. The concrete language artifacts (slices) can be generated automatically by joining a concrete syntax definition with all the semantic roles compliant with that syntax. On a side note, the constraints of the feature model ensure every language configuration of the LPL is feasible.

To increase this flexibility even further, the mechanism would be extended to provide a translation mechanism between apparently incompatible syntax definitions sharing the same semantics: usually this result can be achieved simply by designing language artifacts with a finer grained level of modularity, an additional semantic role and/or a remapping in the order of the nonterminals in the grammar fragments, but some developers may find the possibility to translate a syntax into another more intuitive.

I propose a general roadmap to organize the resarch in order to achieve the project goals:

- In the first six months I would expand my knowledge on LPLs and language development in general, I would 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 could lead to the drafting of a survey on language product lines.

- In the next year I would define a formal method to extrapolate features from the language artifacts. The challenge would be to rely on the artifact's semantics rather than its syntax elements. This method would be used to elaborate a general algorithm to build a multi-dimensional variability model based on the semantic language features. I would than proceed expanding the Neverlang framework to implement all the required features: first ad foremost a way to determine if two or more syntax definitions are compliant will be implemented and, later on, the focus will go towards other required utilities such as syntax translation or syntax inheritance. I would develop a system to automatically generate language slices from language artifacts. Each new feature should be subject to unit testing in order to ensure the reliability of the framework.
- In the next year I would implement, test and perfect the Neverlang version of the algorithm, studying its applicability in real world scenarios. This study would first be focused on DSL development from scratch and than to DSL expansion. The applicability of the method in both scenarios would be evaluated and compared to that of other LPL development frameworks. I would also try to improve the algorithm flexibility introducing a mixed approach, so that a top-down and a bottom-up approaches could be used in conjunction: the main disadvantage of the bottom-up approach is the lack of control over the variability model structure from the developer standpoint; a mixed approach would permit to perform a variability analysis during the design process to set part of the variability model structure while the rest of the features would be generated using the bottom-up approach. The fixed section of the variability model would not be subject to any changes upon updating the set language artifacts.
- In the last six months of the project I would proceed testing the applicability of the method by mixing and matching features from different languages, in order to evaluate the feasibility of a development cycle that does not include any production of language artifacts from scratch. I would also test the algorithm with at least one other framework to check how much the chosen paradigm and modularization strategies affect the semantic evaluation of the language artifacts.

References

[1] 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.

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

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

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

[5] 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.

[6] 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.

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

[8] Görel Hedin and Eva Magnusson. JastAdd — An Aspect-Oriented Compiler Construction System. *Science of Computer Programming*, 47(1):37–58, April 2003.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] David Méndez-Acuña, José A. Galindo, Thomas Degueule, Benoît Combemale, and Benoît Baudry. Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. *Computer Languages, Systems & Structures*, 46:206–235, November 2016.

[15] 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.

[16] August C. Schwerdfeger and Eric R. Van Wyk. Verifiable Parse Table Composition for Deterministic Parsing. In Mark G. J. van den Brand, Dragan Gasevic, and Jeffrey G. Gray, editors, *Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09)*, LNCS 5969, pages 184–203, Dublin, Ireland, June 2009. Springer.

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

[18] 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.

[19] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoît Combemale. Variability Support in Domain-Specific Language Development. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Proceedings of 6th International Conference on Software Language Engineering (SLE'13)*, Lecture Notes on Computer Science 8225, pages 76–95, Indianapolis, USA, 27th-28th of October 2013. Springer.

[20] 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.

[21] 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.

[22] 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.

[23] 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.