

Generalized Software Product Line Extraction

Federico Bruzzone ^a, Walter Cazzola ^{a,*}, Luca Favalli ^a

^aUniversità degli Studi di Milano, Computer Science Department, Milan, Italy

Abstract

Software product line (SPL) engineering has been successfully applied to software development by obtaining software systems as compositions of modular features. Existing approaches to SPL engineering, however, are typically bound to a specific technological space (such as, a programming language and a composer) and *integrated development environment* (IDE), and rely on extraction mechanisms that make strong assumptions on the underlying technological space. This tight coupling hinders reuse, evolution, and adoption of heterogeneous development environments. We propose a general, workbench-agnostic protocol for extracting feature models from existing software artifacts and for configuring and deriving software products. The protocol follows a bottom-up approach based on lightweight dependency units called “atoms”, and organizes the extraction and configuration process around an SPL server (workbench-independent) and an SPL client with a workbench-specific backend and a generic frontend. The protocol makes few assumptions on the underlying software artifacts and is therefore applicable to varied SPLs. The applicability of this approach is presented through a prototypical implementation of the architecture in which several subsystems interact and can be swapped freely without affecting the others. In particular, we focus on the application of such a protocol in the context of language product lines (LPLs), demonstrating its applicability to concrete scenarios while preserving workbench-agnosticism. From bottom to top, the implementation comprises: Neverlang language artifacts, a Java SPL client backend, an agnostic and reusable SPL server written in Go and Prolog, and a JavaScript SPL client frontend.

Keywords: Programming Language Engineering, Software Product Lines, Language Product Lines, Workbench-Agnostic Architectures and Language Workbenches

1. Introduction

Context. Modern software systems are expected to meet the demands of a continuously expanding user base. Given the inherent diversity in user requirements, software must be both highly customizable and easily reconfigurable. To address this, researchers and practitioners have increasingly focused on software product lines (SPLs), which draw inspiration from industrial product line practices to support the development of highly variable systems.




Among software systems, programming languages constitute a particularly interesting case: despite the large variety of programming languages in existence, the need for novel programming techniques and innovative language constructs drives the creation of new programming languages, both general-purpose (GPL) and domain-specific (DSL). Application domains are varied and include concurrency, querying, and hardware design. Despite sharing many common language features, these languages are typically implemented using monolithic compilers or interpreters. This is inefficient, as it prevents language developers from reusing existing features from other languages. Moreover, extending languages is difficult, since implementations are deeply tangled, leading to highly requested features

such as pattern matching in Java taking years to be fully implemented into the language. Similarly, specialization in Rust has remained an unstable feature for years. It is still not available in the stable release of the language due to the complexity of its implementation and potentially type system unsoundness issues [12, 53, 41], further illustrating how monolithic compiler architectures hinder the incremental adoption of new language constructs.

The SPL approach suits the development of programming languages well, as it addresses this limitation by building compilers and interpreters as compositions of modular language features: the compiler or interpreter emerges as the result of the composition process among features selected from a feature model. SPLs whose products are programming languages are dubbed *language product lines* (LPL) [56, 38].

Problem Statement. Two main approaches exist for constructing SPLs [38]. In the top-down approach, a feature model is designed first, then features are mapped to software artifacts, and finally software variants are generated through a configuration process. Manually designing is time-consuming, then this approach is best suited to creating SPLs from scratch, whereas changing the feature model at a later point is ill-advised, as changes to the feature model may affect both the feature implementations and the final products. In contrast, the bottom-up approach reduces the effort required for variability analysis and feature modeling by extracting feature models from existing software artifacts. Compared to the top-down approach, it trades

*Corresponding author.

Email addresses: federico.bruzzone@unimi.it (Federico Bruzzone )
cazzola@di.unimi.it (Walter Cazzola )
luca.favalli@unimi.it (Luca Favalli )

control for easier scalability and maintainability, as it intuitively supports the incremental addition of new components and features while automatically updating both the feature model and its mappings without human intervention.

A major challenge in feature model extraction is the lack of general protocols applicable across different technological spaces. Existing tools for feature model extraction are tightly coupled to specific platforms. Let us focus on LPLs as an example. AiDE [29] is designed to work solely with Neverlang, on the Eclipse platform and powered by FeatureIDE [51]. Similarly, Spoofox [58] provides its own wizard for creating Eclipse projects with skeletal language definitions and grammar extensions. The state of the art in LPL engineering is therefore in direct contrast with the current language design trend, which is shifting towards letting developers choose their own IDE and related tools. This is evident when looking at contributions such as the *language server protocol* (LSP)¹ which is considered the *de facto* standard in IDE development.

A similar issue affects SPLs in general: the tooling for AHEAD [7], FeatureHouse [3] and even AspectJ [34] are tied to (or at least intended for) a specific IDE.

Contribution. The contribution of this work is a generalized protocol enabling feature model extraction and supporting configuration processes. To achieve this, we made virtually no assumptions about how software features are represented, how composition is performed, or which IDE is used. This led us to focus on the fundamental structure of SPLs, independent of constraints imposed by specific language workbenches.

To validate the approach, we created a prototypical implementation that extracts a LPL from Neverlang language artifacts, comprised of a server written in Go and Prolog, a client backend written in Java and a client frontend written in Javascript using Cytoscape.² Given a Neverlang codebase, the user can interact with the generated feature model and derive a configuration through the graphical user interface. Although the implementation itself is not intended as a full-fledged contribution, but rather as proof of concept, we assess the potential of the protocol in terms of lines of code saved in case the server implementation, the frontend, or both were to be reused in a different technological space.

Structure. The remainder of this paper is structured as follows: §2 presents any background information needed to understand our contribution; §3 introduces LogLang a Neverlang-based language product line that will serve as both a running example and as an evaluation case study; §4 describes the protocol in detail; §5 presents a prototypical implementation of the protocol and its usage, while providing a rough estimate of the associated implementation effort. Finally, in §6 and §7 we present any related works and draw our conclusions on this work.

2. Background

This section introduces the foundational concepts underlying our work, namely software product lines, feature modeling,

language product lines, and language workbenches. While our contribution is not tied to a specific technological space, we also briefly present Neverlang and AiDE, as they will be relevant to discuss parts of the protocol and of the proof-of-concept implementation.

2.1. Software product lines and feature modeling

Variability-rich software system development builds on principles from SPL engineering and feature-oriented programming. An SPL is a family of related software products whose commonalities and variabilities are expressed in terms of *features* [4]. SPL engineering combines *domain engineering*, which defines reusable software artifacts, with *application engineering*, which derives concrete products by selecting and composing features. A central activity in SPL engineering is *feature modeling*, introduced by the FODA method [32]. A *feature model* captures the variability of a system by describing features and their dependencies. A product is derived by selecting a valid subset of features, called a *configuration*. Features belonging to a configuration are *active*, while all others are *inactive*. Configuration validity is determined by dependencies expressed either implicitly—through mandatory, optional, alternative, and grouped features, as well as parent—child relations—or explicitly via cross-tree constraints formulated as boolean expressions. Both implicit and explicit dependencies may introduce anomalies such as dead features, false-optional features, and *atomic sets* of features that always co-occur. Detecting and preventing such anomalies through static analysis is an active research area, encompassing both structural [9] and behavioral [8] approaches. When designing a generalized protocol for LPL extraction, a generalized protocol must be able to capture each of these elements.

2.2. Language product lines

Applying SPL principles to language development leads to LPLs [39], in which interpreters and compilers are derived from combinations of language features rather than implemented as monolithic systems. LPL engineering facilitates language reuse and variability management, allowing languages to be tailored for specific purposes such as security (*e.g.*, Java Card [22]), education [11, 21], or the incremental extension of existing languages with new constructs, such as type-checked SQL queries [28]. Variants of state-machine languages, for instance, have been successfully modeled as a single language family (*e.g.*, [52, 24, 56]). LPLs can be constructed following either a *top-down* or a *bottom-up* approach [38]. The top-down approach starts from a feature model derived through domain analysis [46, 4], after which features are mapped to language artifacts and configured into variants. The bottom-up approach, in contrast, derives the feature model automatically from existing language artifacts. This latter approach aligns with extractive and reactive SPL adoption techniques [37] and is the focus of this work.

2.3. Neverlang and AiDE

Neverlang [18, 54] is a language workbench that supports the modular development of programming languages. Language features are encapsulated in independent components called

¹<https://microsoft.github.io/language-server-protocol/>

²<https://cytoscape.org/>

slices, which can be compiled, tested, and reused across different language implementations. Slices are built from *modules* defining reference syntax and *roles* implementing semantic actions according to syntax-directed translation principles [2]. Dependencies between modules arise from shared attributes accessed or produced by semantic actions, enabling an explicit representation of feature interactions. Ensuring the soundness of such dependencies under separate compilation—*i.e.*, guaranteeing that no attribute is accessed before being defined—has been recently addressed through *nlgcheck* [14], a static analysis tool based on data-flow analysis that detects potential runtime errors at compile time while preserving the modularity benefits of separate compilation. Modules are composed into slices, and slices are further composed to produce a complete language implementation. Grammar composition, renaming mechanisms, and role sequencing enable flexible language assembly while preserving modularity.

AiDE [56, 55] extends Neverlang with variability management capabilities for LPL engineering. It extracts feature and dependency information from language modules and synthesizes a bottom-up feature model [29]. Its graphical interface allows developers to explore the generated model, configure variants, resolve dependencies, and test viable language instances. Neverlang and AiDE provide a suitable case study: the former supports LPL engineering while the latter, though tied to a specific technology, uses a feature model extraction algorithm [56, 55, 39] that can be generalized for the purpose of this work.

2.4. PROgrammation en LOGique

Colmerauer *et al.* [23] designed Prolog in the 1970s around the idea that programs can be expressed as logical descriptions of knowledge. It relies on a subset of first-order predicate logic and builds on seminal work in automated theorem proving [31, 47]. Although conceived for natural language processing, its conciseness and expressive power led to adoption in domains such as formal logic, knowledge representation and reasoning, and database programming [43].

In Prolog, *facts* are statements describing properties of objects or relationships between objects. For example, to encode that Ulysses, Penelope, Telemachus, Achilles, and others are characters appearing in Homer’s *Iliad* and *Odyssey*, we can declare a set of facts as shown below.

```

1 character(priam, iliad).
2 character(achilles, iliad).
3 character(hector, iliad).
4 character(ulysses, iliad).
5 character(helen, iliad).
7 character(ulysses, odyssey).
8 character(penelope, odyssey).
9 character(telemachus, odyssey).
10 character(helen, odyssey).

```

A collection of facts, and later of rules, constitutes a *database* encoding the knowledge of a given domain in logical form. By adding further facts, we can express additional properties like gender or relationships like parenthood.

```

1 male(priam).
2 male(achilles).
3 male(hector).
4 male(ulysses).
5 male(telemachus).
6 female(helen).
7 female(hecuba).
8 female(penelope).

```

```

1 father(priam, hector).
2 father(ulysses, telemachus).
3 mother(hecuba, hector).
4 mother(penelope, telemachus).

```

Given a database, a *query* is a request to retrieve or verify information from the database. Prolog answers Yes if the query can be proven from the available facts and rules, and No otherwise. For instance, the question “Is Ulysses male?” corresponds to the following query.

```

?- male(ulysses).
Yes

```

Whereas the same question posed for Helen yields a negative answer.

```

?- male(helen).
No

```

Variables—syntactically beginning with an uppercase letter or an underscore—can unify facts within the database. During query evaluation, Prolog searches for facts or rules that unify with the query and substitutes variables accordingly [43]. This mechanism enables queries such as “Who is Telemachus’ father?”. Multiple answers may exist depending on the query.

```

?- father(X, telemachus).
X = ulysses

```

```

?- character(X, odyssey).
X = ulysses ;
X = telemachus ;
X = helen ;
X = penelope ;
No

```

Goals in a conjunctive query may share variables, thereby constraining multiple predicates to refer to the same value. For example, the question “Who is a father in the *Odyssey*?” can be expressed as a conjunction of the goals `character(X, odyssey)` and `father(X, Z)`, where the variable X is shared:

```

?- character(X, odyssey), father(X, Z).
X = ulysses, Z = telemachus

```

Rules allow the derivation of new properties or relations from existing ones. For instance, the predicate `son(X, Y)` can be defined as holding when X is male and Y is either the father or the mother of X. Rules can be queried just as facts.

```

1 son(X, Y) :- father(Y, X), male(X).
2 son(X, Y) :- mother(Y, X), male(X).

```

```

1 module Backup {
2   reference syntax {
3     Backup ← "backup" String String;
4     Cmd ← Backup;
5     categories : Keyword = { "backup" };
6   }
7   role(execution) {
8     0 .{
9       String src = $1.string, dest = $2.string;
10      $$FileOp.backup(src, dest);
11    }.
12  }
13 }
14 slice BackupSlice {
15   concrete syntax from Backup
16   module Backup with role execution
17   module BackupPermCheck with role permissions
18 }
19 language LogLang {
20   slices BackupSlice RemoveSlice RenameSlice
21     MergeSlice Task Main LogLangTypes
22   endemic slices FileOpEndemic PermEndemic
23   roles syntax < terminals < permissions : run
24 }

```

Listing 1: Excerpt of the LogLang source code.

```

?- son(telemachus, Y).
Y = ulysses;
Y = penelope;
No

```

Although of course other options are viable, Prolog is well suited for the requirements of this work. It can be used to represent features and their dependencies in a software product line. Please refer to §5 for more details.

3. Running Example

LogLang. While our contribution does not make any assumptions with regard to the technological space being used to develop a SPL, we will present the feature model extraction protocol and our evaluation case study in terms of its application with LPLs written in Neverlang. We will specifically focus on the LogLang LPL, a family of languages for the definition of log rotating tasks, similar to the logrotate UNIX utility.

The Backup module shown in Listing 1 declares a syntax for the backup task (lines 2-6), comprised of two productions. On line 5, the grammar excerpt can be tagged by the developer to declare specific implementation concerns. We will discuss how these tags are relevant in §4. Each role, introduced by the keyword `role` (line 7), defines part of the language semantics. The specifics of how the semantics are implemented are not relevant to this work, but for easier understanding, it suffices to say that each role can be mapped to a production during the composition phase, then the evaluation is driven by passing attributes along the parse tree, according to the *syntax-directed translation* technique [2].

The composition of syntax and semantic assets into a language is twofold: 1. between modules, which yields slices, and 2. between slices, which yields a language implementation. The BackupSlice slice (lines 14-18) composes syntactic assets (in this case the reference syntax from the Backup module, line 15) and semantics assets (from two separate roles of two different modules, lines 16-17). Finally, the language descriptor (lines 19-24) composes several language features into a complete language interpreter or a compiler (lines 20-21), including other information such as parse tree visits (line 23) and relevant data structures (line 22).

Fig. 2 shows an exemplary feature model for the LogLang LPL. It contains all the language features that can be composed into a complete language specification, as well as their semantics. The features in the model comprise syntactic and semantic assets. Therefore, an SPL extraction mechanism must identify all syntactic and semantic assets from Neverlang modules, then, upon performing the composition of a configuration into a language, the composer should be able to generate all the relevant slices (lines 14-18) and the languages unit (lines 19-24). Of course, this may vary depending on the technological space, but in §4 we will show how this can be achieved without making any assumptions in this regard.

4. Feature Model Extraction Protocol

This section details a general protocol designed to extract a feature model from a set of software assets. The goal is to make minimal assumptions about the technological space in use, so that the same protocol can be reused.

4.1. Overview

Fig. 1 summarizes the most important steps of the feature model extraction protocol, as well as the additional steps that lead to the configuration, its validation and product derivation. The process is akin to the human circulatory system, as the information is first sent across and manipulated by several components, from the core—the client backend—all the way back to the peripheries—the client frontend—and then all the way back. The way forward—highlighted in red in Fig. 1 and comprised of steps ❶ to ❺—involves the extraction of features from the software assets, the creation of a feature model and the configuration process, whereas the way back—highlighted in blue and comprised of steps ❻ to ❸—involves the validation of a configuration and the product derivation.

In the following, let us introduce the actors involved in the process and then detail each step of their interaction.

4.2. Actors

The protocol involves the following actors. All actors are highlighted by boxes with sharp corners and in a different color. Other entities are represented by grey boxes with rounded corners. The final product is represented by a black box. Please refer to §4.3 for more details on the behavior of each actor.

The **User** is the actor in charge of developing the language assets and of interacting with the feature model through a frontend application to derive a product configuration. Depending

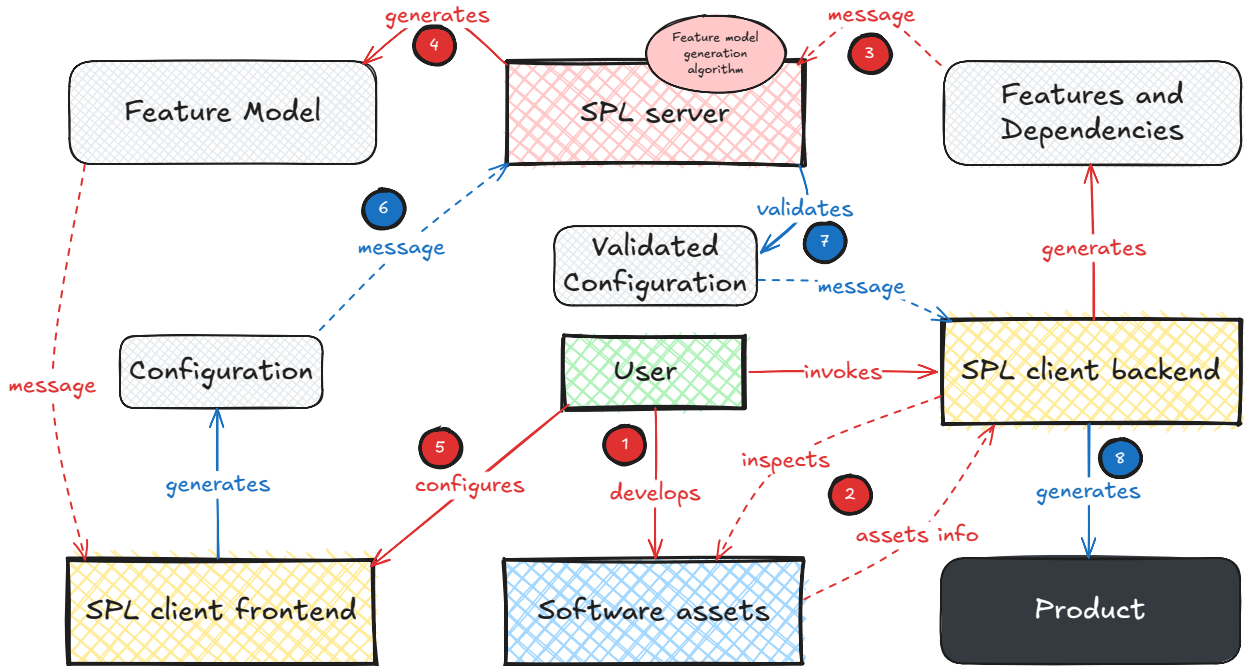


Figure 1: High-level overview of the protocol, including its actors and its steps.

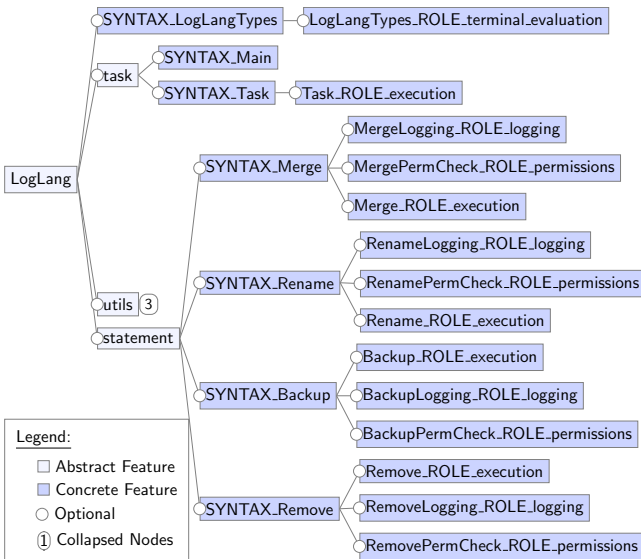


Figure 2: Feature model of the LogLang LPL.

on the development process and on the project scale, the user may be a single entity or a team of multiple entities, such as language developer, language deployer and language user [19].

The **Software assets** are the concrete software artifacts of the product line. With regards to our running example, it comprises the linguistic assets, including both syntax and semantics, that may be part of a language product—*i.e.* the reference syntax and roles from Neverlang modules. Instead, slices and languages are our expected output (the result of the composition) and therefore are not considered part of the software assets.

The **SPL client** is the software that allows the user to interact with the services offered by the server. It behaves as an

adapter that translates any concepts specific to a technological space (Neverlang in this case) into generic SPL concepts that can be understood by the server and vice-versa. The client is comprised of two pieces of software:

- the backend is a piece of software that may be integrated within the IDE as a plugin or used in a standalone fashion; its two tasks are supervising the project structure to collect any software assets to be fed to the server and translating a validated configuration into a product;
- the frontend is an application the user can interact with to inspect the feature model, eventually deriving a product configuration; it translates the feature model metadata from the server into a (optionally) graphical and interactive representation and it sends the active features selected by the user back to the server.

Going back to the example, the client backend should collect all relevant pieces of information from Neverlang modules, ignoring slices and languages during the way forward, then generate the correct slice and language units depending on the configuration it received from the server. Since different technologies—even when just considering language workbenches only [27]—handle composition in a different manner, the SPL client backend is the most likely portion of the architecture that needs to be re-implemented when changing technological space.

The **SPL server** receives the information regarding software assets from the client backend, generates the feature model and sends it to the client frontend. In fact, the feature model generation algorithm is possibly the most important component of the SPL server: in this work we adopt a tag-based bottom-up feature model construction algorithm that was already tested in literature (see §4.4), but this component is intended to be fully customizable, as long as it is able to consume the same pieces

of information. Please notice that in this stage the pieces of information received by the server hold no notion regarding the fact that the original software assets were written in Neverlang, instead it will rely on generic concepts such as artifacts, features, tags and atoms (more on that later). Upon the user selecting a configuration, the client frontend and the server perform a back and forth communication apt at validating the configuration itself. Eventually, the user will commit a valid configuration that the server sends back to the client backend. Finally, as previously mentioned, the client backend translates the configuration into a product. In our example the product is a language interpreter or a compiler compatible with the Neverlang runtime environment.

4.3. Protocol

The actors interact according to the following protocol; refer to the numbered steps in Fig. 1 for a visual overview.

The way forward. The user develops the software assets ❶ that constitute the product line. The protocol may either be integrated into the development process from the outset or adopted retrospectively, allowing it to be applied to pre-existing software assets. In our example, the user would develop Neverlang modules akin to those shown in Listing 1 lines (1-13).

The user then interacts with the SPL client backend. Depending on the implementation, this backend may be a standalone command-line application or a component integrated directly into the development environment. Upon invocation, the SPL client backend inspects the software assets contained in the project ❷ and extracts the relevant information. Although the precise nature of this information will be detailed later in §4.5 and §4.6, it is sufficient for now to note that it describes the software artifacts and the capabilities they provide and/or require. For instance, the Backup module shown earlier would provide two artifacts: one for the syntax and one for the semantics.

The extracted information is subsequently transformed by the SPL client backend into a message whose representation is independent of the technological space of the original language assets ❸. This representation can therefore be interpreted by the SPL server in terms of features and their dependencies.

The message is transmitted—either locally, via sockets or pipes, or over the network—to the SPL server. The server uses this message to generate a feature model ❹ that includes all features, their relationships and dependencies, as well as the mapping between features and the corresponding software artifacts. A possible output of this phase is a piece of metadata that represents a feature model akin to that shown in Fig. 2. Notably, the feature model is unaware of the fact that the original artifacts were written in Neverlang, thus the SPL server remains agnostic to both the development process of the artifacts and the specific technological space.

The feature model is subsequently serialized into a message and transmitted to the SPL client frontend, where it is rendered as an interactive graphical representation. Through this interface, the user can configure ❺ a product by selecting and deselecting features in the feature model.

The way back. During the configuration process, each user interaction with the SPL client frontend triggers a message that

is transmitted to the SPL server ❻. This architectural decision deliberately decouples the user interface from the configuration validation logic. In particular, the SPL client does not implement any mechanism for checking configuration validity or enforcing constraints. Instead, all validation logic is centralized on the server side, where consistency rules, feature constraints, and domain-specific requirements are evaluated in a uniform and authoritative manner. This centralization not only reduces redundancy but also ensures that validation behavior remains consistent across different client implementations. Consequently, the frontend can be replaced, extended, or customized to support alternative user experiences—such as graphical interfaces, command-line tools, or web-based configurators—without requiring any modification or reimplementing of the underlying validation logic.

In our example, we developed a simple yet complete graphical user interface that offers capabilities similar to other commonly used SPL engineering environments such as FeatureIDE.

The interaction between user inputs and the SPL server progressively refines the configuration state and ultimately leads to the construction of a complete product configuration ❽. In this process, the client and the server establish a contract in which the client trusts the server to perform all relevant checks regarding the validity of the configuration. This configuration is therefore validated by the server and guaranteed to satisfy all structural and semantic constraints defined by the product line. Once validated, the configuration is considered ready as a concrete software product. At this stage, it is sent back to the SPL client backend, which leverages technological space-specific knowledge to translate the validated configuration into a final product ❾. In our case, the client backend would receive the list of all the active features from the server and use it to generate all the required slices (akin to Listing 1 lines 14-18) and a language unit (akin to Listing 1 lines 19-24).

Depending on the technological space and the specific needs of its users, the client backend can either execute the product directly or generate it as a piece of source code, that can be later compiled, distributed or even modified manually using traditional development workflows. In our example, we opted to simply generate the slices and language source code to be later fed to the Neverlang compiler.

4.4. Feature Model Structure

In this section, we discuss the structure adopted for the technological space-agnostic feature model and its constituent features. The proposed design largely adheres to established practices in the literature, while incorporating certain specific choices that warrant further discussion. Please notice that, due to the modularity of the proposed architecture, it is relatively easy to replace the feature model hereby presented and its generation algorithm with alternatives without affecting the rest of the architecture, namely the SPL client and the software assets, and, most importantly, the content of the messages sent among the actors.

Hierarchical Structure. The feature model is organized according to a hierarchical structure—*i.e.*, a tree—that serves as

its structural backbone. Within this tree, some features represent alternative realizations of the same application concern, whereas others correspond to subcomponents of a broader functionality. This hierarchical organization provides a clear decomposition of concerns and supports a systematic representation of variability.

On top of this backbone tree, we superimpose a graph whose edges capture the dependencies among features. In other words, the tree encodes structural relationships, while the graph represents crosscutting constraints and inter-feature dependencies. Through this hybrid approach, we preserve a well-structured and comprehensible feature model while simultaneously enabling the explicit representation of all relevant dependencies.

In more complex scenarios, certain dependencies may be influenced by the values of feature attributes. These attributes are determined dynamically at runtime during the configuration process, according to user inputs. Such inputs may modify only the dependency edges in the graph, leaving the structural edges of the underlying tree unchanged. This separation between structural relationships and dynamic dependencies aims to keep the model stable and interpretable, even in the presence of runtime variability.

Tagging Software Components. Features that relate to similar concerns are grouped by means of the notion of *tag*, in a manner consistent with literature [39]. A tag is a label that provides a concise, high-level description of the concerns in which a feature is involved, while remaining sufficiently abstract to avoid being tied to a specific technological context. A feature may show multiple tags, suggesting that it is involved with several concerns.

Although the SPL server processes tags in a standardized and uniform way dictated by the feature model generation algorithm, the responsibility for assigning tags to software components lies with the SPL client backend. Depending on the chosen implementation strategy, tags may either be provided manually by the user or automatically inferred and attached by the backend.³ In our running example, we opted for the latter, as Neverlang provides a tagging mechanism, as shown in Listing 1 line 5. This design ensures flexibility while preserving a common semantic interpretation of tags at the server level.

Each feature may be associated with zero or more tags. For instance, consider a feature responsible for managing a motion sensor in a residential environment. Such a feature could be tagged with *sensor*, *motion*, and *security*. The *sensor* tag identifies its relation to sensing functionalities; *motion* emphasizes its role in detecting movement; and *security* highlights its potential use in intrusion monitoring. This tagging mechanism facilitates the grouping and management of related features, as features sharing one or more tags are likely to address overlapping or closely related concerns.

Feature Tree Derivation. Tags are used to automatically derive the initial backbone of the feature tree from the set of software components by means of Algorithm 1, which was part of a

³Although feasible, writing protocol messages by-hand is not advised. If the technological space does not natively support any tagging mechanisms, one may use code comments with a special syntax that can be parsed by the SPL client backend.

Algorithm 1 buildFeatureTree(p:Node)

```

1: begin
2: tagInstances(t) := length([ c for c in p.children() if t in
   c.tags() ])
3: T = [ t for t in c.tags() for c in p.children() if tagIn-
   stances(t)>1 ]
4: while length(T) > 0 do
5:   t' = t with max tagInstances in T
6:   n = newNode(name = t')
7:   for each c in p.children() do
8:     if t in c.tags() then
9:       c.removeTag(t')
10:      p.removeChild(c)
11:      n.addChild(c)
12:     end if
13:   end for
14:   p.addChild(n)
15:   T = [ t for t in c.tags() for c in p.children() if tagIn-
   stances(t)>1 ]
16: end while
17: for each c in p.children do
18:   buildFeatureTree(c).
19: end for
20: end

```

prior work [39]. Recall that this is only one possible algorithm and other options may prove as effective. The construction process begins with a dummy tree in which all features are direct children of a single, tagless root node. This initial structure serves as a neutral starting point for hierarchical refinement.

At each iteration, the most frequently occurring tag is identified and promoted to an abstract feature. All sibling nodes containing that tag are then relocated as children of the newly created abstract node, and the extracted tag is removed from those features. When no tag appears more than once among a set of siblings, the algorithm proceeds recursively on each child node.

Despite its conceptual simplicity, the algorithm exhibits several noteworthy properties. First, it is guaranteed to produce a tree structure. Since it is initially invoked on a tree and each transformation step either relocates, removes, or retains child nodes without introducing multiple parent assignments, no node can become a child of more than one parent. Consequently, the resulting structure is free from overlapping features. Second, because the algorithm generates an elementary feature model, all nodes are optional with respect to their parent. This reflects the absence of mandatory constraints at the structural level and delegates additional constraints to the dependency graph. Finally, the algorithm can accommodate crosscutting constraints by distributing them across different branches of the derived feature tree. In this way, structural organization and crosscutting relationships remain conceptually separated yet consistently represented within the overall model.

The final configuration of the *feature model tree* comprises three types of nodes:

- *root node*, which represents the principal element of the

model. It must be activated in every valid configuration and represents the program as a whole.

- *central nodes*, corresponding to abstract features introduced from tags originally associated with concrete features. These nodes serve a purely structural purpose and organize the feature space.
- *leaf nodes*, which correspond to concrete features. Only these nodes may participate in dependency edges. Users are primarily expected to interact with these nodes during the configuration process.

4.5. Protocol Components

We can now examine the information exchanged through the protocol for the purpose of extracting a feature model from linguistic assets in detail.

Artifacts. We dub “artifact” any piece of code that can be part of one or more features in the product line. Depending on the technological space of choice, an artifact can either be a class, a function, or, in the context of language workbenches, a grammar production, a semantic asset or a combination of the two. Either way, the protocol is not concerned with how entity are implemented, as such choice is left to the implementation of the SPL client backend, but rather to the association between artifacts and features. In this context, the following statements are true:

- a feature is comprised of one or more artifacts;
- an artifact is part of one or more features.

This leaves almost complete flexibility in terms of artifacts and their reusability: depending on the implementation of the SPL client backend, users can either opt for a one-to-one mapping between artifacts and features, let an artifact be part of only one feature or leave the complete flexibility of reusing an artifact across several features. For instance the `for` and `while` loops may share part of their semantics—*i.e.*, some artifacts—while being different features.

Feature Dependencies. Features inherently establish relationships with other features in order to accomplish specific functional objectives [59]. A feature dependency arises when one or more program elements—such as methods or attributes—belonging to a given feature rely on elements that reside outside the structural boundaries of that feature. A representative example is the definition of an attribute within one feature that is subsequently accessed or utilized by another feature [17]. Such references introduce a level of coupling that must be explicitly represented in the feature model to ensure that the configuration process leads to a valid product.

In practice, we associate dependencies with software artifacts rather than directly with features. Then, features derive their dependencies from the artifacts they are comprised of—*i.e.*, the dependency set of a feature is the union of the dependencies of its artifacts. For instance, the Backup syntax presented in Listing 1 is, of course, incomplete: it lacks the definition on how to expand the `String` nonterminal, among other things; therefore it will depend on other syntactic assets that “know” how to perform this expansion. Conversely, it “knows” how to expand the `Cmd` nonterminal (on the left-hand side of the second

production) and therefore it will be able to satisfy a dependency of any syntactic assets having `Cmd` on the right-hand side of a production.

From a configurational perspective, dependency relations impose constraints on feature activation. If a feature depends on another feature, the selection (activation) of the former logically entails the selection of the latter. Additionally, certain features may be mutually incompatible. For instance, when two features require exclusive access to a non-shareable resource, they cannot be simultaneously activated.

Atoms. In order to represent the complete range of possible dependencies without making any assumptions on the technological space, we use the concept of an *atom*. An atom is defined as a token that can either be required or provided by an artifact—and therefore by a feature. An atom is modeled as a dictionary-like structure, encoding arbitrary information.

Atoms constitute the minimal unit of dependency information within the protocol. An artifact may declare that it provides a given atom, requires a given atom, or both, depending on its functional role within a configuration.

Going back to the previous example, a very reasonable expression of the dependencies for the Backup syntax from Listing 1 in JSON format is the following.

```
{
  "name": "Backup syntax",
  "requires": {
    "all": [ { "nonTerminal": "String" } ]
  },
  "provides": [
    { "nonTerminal": "Backup" },
    { "nonTerminal": "Cmd" }
  ]
}
```

Requirements Types. You may have noticed the “all” key in the prior example. In fact, to represent the full range of possible relationships between features, we distinguish four distinct types of requirements:

- **all**—atoms that must be present for the feature to function correctly;
- **not**—atoms that are incompatible with the feature;
- **any**—groups of atoms in which at least one must be present;
- **one**—groups of atoms in which exactly one must be present.

To capture more intricate dependencies, such as *any*-type requirements between two groups of atoms where at least one group must be present, abstract features can be used. These abstract features can require the individual atoms while simultaneously providing their combination, thereby expressing higher-level dependency structures.

For instance, a syntactic definition requires that “all” non-terminals on the right-hand side are provided by at least one feature, whereas semantic assets require that exactly “one” implementation of a data structure is available to work properly (such as, the `$$FileOp` data structure in Listing 1, line 10).

Variables and Globals. To further enhance variability and

compatibility across different contexts, some dependencies may be expressed in terms of the value of specific attributes that can be set at configuration time. Feature models supporting feature attributes are often referred to as “extended feature models”.

To support this capability, we introduce the concepts of local and global variables, which will henceforth be referred to as *variables* and *globals*, respectively, for brevity.

Variables are bound to a particular artifact implementing one or more features. For example, consider a feature `display` that can operate in either a light or dark interface mode, implemented via an artifact `display concrete`. A variable `$mode` can be used to specify the interface requirements of this artifact, as illustrated below.

```
{
  "name": "display concrete",
  "requires": {
    "all": [ {"interface": "$mode"}, {"power": "on"} ]
  },
  "variables": [ { "name": "$mode", "default": "dark" } ],
  "provides": [{"display": "screen"},
    ↪ {"feedback": "video"}],
  "tags": ["screen"]
}
```

Globals adhere to the same principle, with the distinction that they are shared across the entire SPL. Any modification to a global variable affects all features within the system. For instance, Neverlang provides a language-wide nonterminal renaming mechanism (please refer to [19] for more details) that allows for any nonterminal to be renamed at will. This functionality was introduced to fix the incompatibility among language assets that were originally compatible except for the nonterminal used.⁴ In this context, all Neverlang nonterminals can always be considered as globals whose value can be set across the entire configuration, therefore, if globals are highlighted with the @ symbol, the dependencies of the Backup syntax can be rewritten as follows.

```
{
  "name": "Backup syntax",
  "requires": {
    "all": [ { "nonTerminal": "@String" } ]
  },
  "provides": [
    { "nonTerminal": "@Backup"},
    { "nonTerminal": "@Cmd" }
  ]
}
```

If a required or provided atom is defined in terms of a variable or a global, then changing its value may affect the dependencies between features, because a specific dependency may be satisfied only if specific attributes have a specific value at configuration time.

⁴Consider how an asset for additive expressions with the syntax `Sum <- Term "+" Expr` would be incompatible with another asset with the `Expr <- AddExpr` syntax; the two assets can be rendered compatible by renaming `Sum` to `AddExpr`.

4.6. Messages

We can finally introduce the set of messages that are sent across the protocol’s components to convey all necessary information with the necessary degree of generality. As shown in the examples provided so far, we opted for a JSON implementation of messages in this work, although other options are of course viable. Although examples are omitted for brevity, please consider that all messages share this structure. We also assume that the connection is reliable, thus omitting any acknowledgment messages.

features. Used to load the SPL.

Content: Collection of artifacts and features within the workspace, each with its list of composing artifacts. Each artifact declares its own dependencies, tags and attributes. Features can require only artifacts among this collection.

Response: None.

Usage: The client backend sends the currently available language assets to the server to trigger the generation of the feature model according to Algorithm 1.

featureModel. Used to display the feature model in the client frontend.

Content: The feature model generated by the server, comprised of the features and their relations.

Response: None.

Usage: Upon computing the feature model, the SPL server sends it to the SPL client backend for it to be displayed to the user.

updateAttribute. Used to change the value of an attribute.

Content: Feature name, attribute name, value. Together, these pieces of information allow unique identification of an attribute within the feature model.

Response: Any change to the feature dependencies within the feature model.

Usage: Notifies the SPL server about the fact that the user changed the value of a feature attribute. Such a change may trigger updates to feature dependencies. The SPL server computes any updates and sends it back to the SPL client backend.

activate. Used to toggle the activation status of a feature in the current configuration.

Content: Name of the feature to activate or deactivate.

Response: List of currently active features.

Usage: Notifies the SPL server about the fact that the user changed the activation status of a feature. Such a notification causes the server to updated the internally stored configuration, updating the set of active features. This does not only include the selected feature, but also any dependent features. The resulting set is sent back to the SPL client frontend to updated the view presented to the user.

validate. Used to check if a configuration is valid and to obtain hints on how to fix it.

Content: None.

Response: A map whose keys are the names of any invalid features and whose values are the respective unfulfilled requirements. These requirements are expressed through a dictionary with four possible keys: ALL, NOT, ANY, ONE, each mapped to the

atoms required but not provided or vice-versa. The response also contains providers for each atom—*i.e.*, the list of features that could resolve the dependency by being activated or deactivated.

Usage: request the server to check if the requirements for each active features are fulfilled. If the response is positive, the configuration is valid and can be committed, otherwise the server responds with hints on how to reach a valid configuration.

commit. Used to export the current configuration from the SPL client frontend all the way back to the language assets.

Content: List of active features and attribute values.

Response: None.

Usage: Notifies the SPL server that the user wants to translate the current valid configuration into a product. The same message is then forwarded from the SPL server to the SPL client backend, triggering the product generation. How this is performed depends on the chosen language workbench and the SPL client backend implementation. Regardless, the expected behavior is for the SPL client backend to generate a new language and add it to the workspace among other language assets.

5. Case Study

We briefly present a prototypical implementation of an architecture orchestrated according to the protocol. An in-depth discussion of the implementation details is beyond the scope of this paper, although this case study serves the purpose of demonstrating the applicability of our contribution. Each paragraph focuses on a component that can be independently developed and replaced to adapt to different requirements and scenarios. The software associated to the topics discussed in this section is available at [\[LF\] > \[TODO\]](#).

SPL client backend. The SPL client backend is comprised of two Neverlang-specific adapters. The first adapter generates the *featureModel* message in JSON format by extracting all relevant pieces of information from the .class files obtained through the compilation of Neverlang source files. The client backend is implemented as a Java library that creates an instance of available language assets and then uses Neverlang’s reflective capabilities [20] to create the appropriate artifacts and features for each module. Dependencies and the respective atoms used by the protocol are determined on a syntactical level by considering required and provided nonterminals: each module’s productions are analyzed to identify nonterminals that are not provided by the module itself and that must be provided elsewhere. Each nonterminal is associated with an attribute that reflects its name, allowing the user to rename it and modify the corresponding dependencies possibly filling any gap existing within the grammar [29].

Regarding the semantics, the client creates a feature for each semantic role—*i.e.*, visit of the parse tree—each with two available attributes: priority and visiting order.

The second adapter takes the validated configuration from the SPL server and generates a Neverlang language, akin to Listing 1 lines 19-24. This adapter does not require any reflective capabilities, as it simply translates a JSON input into Neverlang syntax, then writes the result to file.

```

1 exists(Atom) :-
2     exists([], Atom).
4 exists(CantProvide, Atom) :-
5     provides(Provider, Atom),
6     \+ member(Provider, CantProvide),
7     valid(Provider, CantProvide).

```

Listing 2: Atom existence property.

```

1 valid(ToCheck, CantProvide) :-
2     provides(ToCheck, _),
3     requisitesAll(ToCheck, CantProvide),
4     requisitesNot(ToCheck, CantProvide),
5     requisitesAny(ToCheck, CantProvide),
6     requisitesOne(ToCheck, CantProvide).
8 valid(ToCheck) :-
9     valid(ToCheck, []).

```

Listing 3: Feature validity property.

SPL server (Go). The server core implementation leverages the "net/http" standard Go library package. Most relevant information, including feature and artifact names, tags, attribute names and values, hashes, and even atoms can all be represented as strings. To avoid any ambiguous data structures and leverage the strong typing guarantees, we introduced various alias types for the string type.

Local and global attributes are represented as map data structures.

The natural choice for implementing atoms is using dictionaries, as the "encoding/json" package deserializes information into a map by default.

Throughout the implementation, the server stores hashes rather than strings whenever possible to ensure that their format is compatible with the atom naming conventions needed by the Prolog validator.

Notably, the Go core also implements the feature model generation algorithm. This portion of the code is meant to be replaced if the proposed algorithm is unsuitable for the needs of the specific technological space.

SPL server (Prolog). While Go offers a vast standard library to implement client-server communication and serialization capabilities, we deemed a logic programming language as Prolog more suitable to implement the configuration validation capabilities offered by the server.

As shown in §2, Prolog programs are built from *atoms*—the base components of logical statements used in declarations and relationships. Note that this use of “atom” is specific to Prolog terminology and distinct from the protocol-level atoms introduced in §4. The validation process we chose to implement is based on two fundamental principles: atoms may exist (Listing 2), and features may be valid (Listing 3). Intuitively, an atom exists when it is provided by a valid feature, whereas a feature is considered valid when all the following conditions are met:

- it provides at least one atom,
- all of its required atoms are provided by some other fea-

```

1  requisitesAll(ToCheck) :-
2      requisitesAll(ToCheck, []).
4  requisitesAll(ToCheck, CantProvide) :-
5      \+ (requiresAll(ToCheck, Thing),
6          \+ exists([ToCheck | CantProvide], Thing)).
8  requisitesNot(ToCheck) :-
9      requisitesNot(ToCheck, []).
11 requisitesNot(ToCheck, CantProvide) :-
12     \+ (requiresNot(ToCheck, Thing),
13         \+ exists([ToCheck | CantProvide], Thing)).
15 requisitesAny(ToCheck) :-
16     requisitesAny(ToCheck, []).
18 requisitesAny(ToCheck, CantProvide) :-
19     \+ (requiresAny(ToCheck, -, GroupID),
20         \+ (requiresAny(ToCheck, Thing, GroupID),
21             exists([ToCheck | CantProvide], Thing))).
23 requisitesOne(ToCheck) :-
24     requisitesOne(ToCheck, []).
26 requisitesOne(ToCheck, CantProvide) :-
27     \+ (requiresOne(ToCheck, -, GroupID),
28         requiresOne(ToCheck, Thing, GroupID),
29         exists([ToCheck | CantProvide], Thing),
30         requiresOne(ToCheck, Another, GroupID),
31         exists([ToCheck | CantProvide], Another),
32         Thing \= Another
33     ).

```

Listing 4: Requirements validity checks implementation.

ture,

- none of its undesired atoms are provided by any feature.

Verifying the existence of an atom is straightforward, whereas determining a feature’s validity is more complex, as it requires assessing all dependencies types while preventing circular dependencies. Prolog’s expressive capabilities enable a relatively simple implementation of these dependency checks. There are five required checks:

- `provides(feature, atom)`,
- `requisitesAll(feature, atom)`,
- `requisitesNot(feature, atom)`
- `requisitesAny(feature, atom, groupID)`, and
- `requisitesOne(feature, atom, groupID)`.

The logic implemented in Prolog goes beyond checking configuration validity. As previously mentioned, the validation process also provides suggestions regarding which features to activate or deactivate to adjust it. To achieve this, the system first evaluates the validity of all active features and records those that are found to be invalid with their missing requirements. Then, for each missing requirement of each invalid feature it finds any possible providers and reports them to the user through the response to the `validate` message.

In summary, the Prolog dependency validation logic represents the core of the SPL server as it serves both the purposes of validating a configuration and of informing the user on how to fix an invalid configuration. Despite that, thanks to the capabilities offered by Prolog in terms of querying the knowledge base, the

implementation itself is rather succinct, as it amounts to a total of only 32 lines of code, as shown in Listing 4.

SPL client frontend. To demonstrate the interaction with the protocol from a user’s perspective, we developed a simple graphical user interface displaying changes in the feature model graph. The interactive graph is implemented through a versatile Javascript library called Cytoscape [49]. To provide an effective means of interacting with feature attributes, we integrated another library called Tippy⁵ used to create interactive pop-ups on the nodes and the edges of the graph.

Please consider that we made the deliberate choice of not using any well-known variability management software such as FeatureIDE to avoid introducing implementation bias, although in real-world use cases it may be beneficial to implement the SPL client frontend as a plugin for existing software rather than from scratch.

Our prototype can show all relevant kinds of information, including feature nodes (both concrete and abstract), structural edges (parent-child) and dependency edges, dead features and so on. Each interaction with the graph, be it feature activation or deactivation, attribute value edit, or configuration commit, causes the corresponding message to be sent from the client to the server. In case of an `updateAttribute` message, the response may cause the feature model to update; in this case, the graphical representation presented by the client to the user is updated accordingly.

Configuration process. The protocol implementation was tested against a LogLang and a Javascript implementation written in Neverlang, the latter comprised of 50 programming concepts implemented in 73 Neverlang slices. Through the SPL client frontend, we were able to configure all aspects of a Javascript dialect, starting from the configuration process, all the way back to the language generation and usage, including intermediate steps such as nonterminal renaming, syntax and semantics configuration. We replicated experiments we performed in prior works, such as the gradual extension of Javascript for didactic purposes [19]. However, the frontend is still in a proof-of-concept state and its usability with large LPLs such as Javascript is limited. Thus, we will hereby showcase configuration examples regarding LogLang: the client frontend and the configuration process are inspired by products such as FeatureIDE, whose description is beyond the goal of this paper, but please let us showcase the intended workflow and summarize the capabilities of the protocol with an example. First, the user invokes the client backend that inspects the LogLang software assets and generates a collection of artifacts, features and their dependencies in JSON format, according to the structure presented in §4. Then, this JSON is parsed by the SPL server in one of three possible modes:

- read from pipe between client and server;
- loaded from file through the command line;
- loaded from file through GUI.

This triggers the feature model generation algorithm and the result is forwarded to the client frontend that graphically presents

⁵<https://atomiks.github.io/tippyjs/>

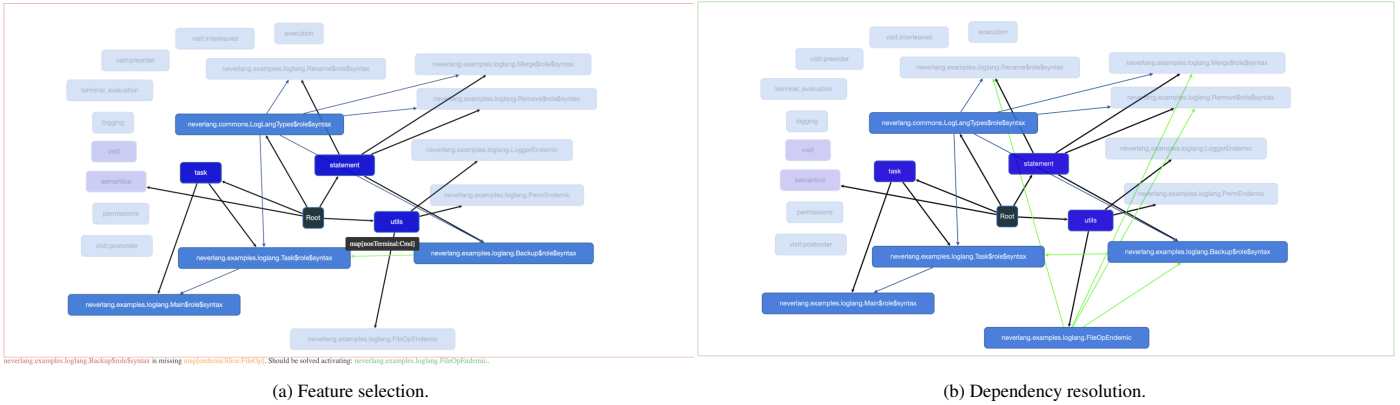


Figure 3: Exemplary configuration process step within the SPL client frontend.

it to the user.

Consider the case presented in Fig. 3, showing a screenshot of the client frontend in which the LogLang feature model was loaded and in which the user is manually configuring a product. Fig. 3a contains a partial (invalid) configuration that cannot be committed; as suggested in the box—highlighted in red due to the configuration being invalid—the dependencies can be solved by activating the FileOpEndemic feature.⁶ Fig. 3b shows the same configuration process after activating the suggested feature; as a result, the box is highlighted in green: no more features need to be activated or deactivated and thus the configuration is valid and can be committed.

Other configuration aspects were tested in a similar manner; most notably, configuring the semantics, eventually led to a selection among one of three possible features: `visit:preorder`, `visit:postorder`, and `visit:interleaved` (currently not selected and therefore greyed out on the left side of Fig. 3a). The user can decide the parse tree traversal strategy for the language semantics by activating the corresponding feature. As dictated by the SPL client backend, each feature concerned with a semantic role has a “priority” local variable that accepts an integer value. Upon committing a valid configuration, the value of this variable will be used by the client backend to determine the role execution order (each role representing a parse tree visit).

The configuration process may eventually lead to a situation in which the partial configuration cannot be fixed by taking any additional actions. In this case, the box will be red as in Fig. 3a, but there will be no available suggestions on the bottom, since the validation step performed by the Prolog server was unable to find any features whose provided atoms could render the configuration valid.

Please note that this front-end application is agnostic to both the language workbench used and to the server-side validation process: it simply displays the information received from the server and sends back any inputs received from the user. This means that the same experiment could be replicated using a different frontend, provided it can send/receive the correct messages from/to the server.

Implementation effort. Due to the qualitative nature of our

contribution, a full-fledged empirical evaluation is beyond the scope of this paper, however let us complete this section by briefly estimating the development effort needed to implement each component in the architecture.

The implementation is in a proof-of-concept state that could be rendered more complex with more sophisticated functionalities and because implementing different components using different languages may present some trade-offs. For instance, swapping the Prolog core with a different language may offer better performance at the cost of a more verbose implementation. The generality of such a discussion is therefore questionable, but we hope readers will be able to perceive the potential benefits of using such an architecture to extract product lines in different technological spaces.

Our implementation is comprised of 2551 lines of code (LoC) in total, of which:

- 1339 LoC of SPL server (Prolog + Go);
- 535 LoC of SPL client frontend (Javascript);
- 677 LoC of SPL client backend (Java).

We can further divide the SPL server into the following:

- 53 LoC of Prolog core;
- 410 LoC of feature model generation algorithm;
- 876 LoC of other server functionalities (message generation, parsing etc.).

Each of these five components—Prolog validation core, feature model generation algorithm, Go server infrastructure, Java client backend, and JavaScript client frontend—can be reused independently. In summary, this gives us a rough estimate of the potential reduction in implementation effort in the following scenarios:

1. 84% less effort in replacing the feature model generation algorithm without changing the technological space nor the rest of the server nor the client;
2. 98% less effort in swapping the validation logic without changing the technological space, nor the rest of the server nor the client;
3. 48% less effort in replacing the entire server without changing the rest;
4. 73% less effort in using the architecture with a different technological space (for instance using a different language workbench instead of Neverlang, thus replacing

⁶Package names and role names omitted for brevity.

the Java backend) while reusing the rest of the architecture;

5. 79% less effort in replacing the client frontend (for instance, using a different GUI or an existing tool such as FeatureIDE) while reusing the rest of the architecture.

Among these cases, we consider case number 4 to be the most likely, since the components representation and the composition mechanisms can differ wildly depending on the technological space, whereas the other cases are a matter of preference with regards to the extraction and configuration logic and of user experience. Of course, the actual implementation effort may vary depending on the actual technology being used and increases if more than one component needs to be replaced at the same time, however the modular architecture allows developers to tune the effort to their needs.

6. Related Work

This section positions our contribution with respect to feature model extraction, language workbenches, protocol-based approaches, feature model analysis, and compositional language development.

6.1. Feature Model Extraction and SPL Adoption

Our work follows the bottom-up, extractive approach to SPL engineering originally identified by Krueger [37]. Several techniques have been proposed for reverse-engineering feature models from existing assets. She *et al.* [50] reverse-engineer feature models from natural-language descriptions of configuration options (*e.g.*, Kconfig), whereas our protocol operates on generic artifacts and their provide/require relationships without assuming a particular variability encoding. Acher *et al.* [1] focus on extracting, merging, and slicing feature models from heterogeneous artifact descriptions, treating feature models as first-class entities—complementary to our emphasis on a workbench-agnostic extraction protocol. Haslinger *et al.* [30] use formal concept analysis to extract feature models from source code; unlike their implementation-level focus, our protocol abstracts over artifact representations through atoms, making it applicable across technological spaces. Martinez *et al.* [40] developed BUT4Reuse, a bottom-up framework for extracting reusable elements from product variants. While BUT4Reuse identifies commonalities and variabilities across existing products, our protocol extracts a feature model from modular artifacts organized around explicit dependency declarations. Assunção *et al.* [5] survey re-engineering approaches for SPL adoption; our contribution extends this landscape with a protocol-level abstraction that decouples the extraction process from both the language workbench and the development environment.

6.2. Language Workbenches and Modular Languages

Language workbenches provide integrated environments for defining and composing programming languages, and several support modular development to varying degrees. Our contribution is orthogonal to the choice of workbench and aims at

bridging different technological spaces through a common protocol.

Spoofox [58, 33] relies on declarative meta-languages for syntax, transformations, and type checking, but its tooling is tightly coupled to its own meta-languages and to the Eclipse platform. MPS [57] adopts a projectional editing approach that supports language composition via extension and embedding, yet its tight IDE integration makes it difficult to decouple the SPL engineering process from the platform. Xtext [10] generates parsers and editors from grammar specifications within Eclipse, offering grammar inheritance and mixin composition but limited SPL support. MontiCore [36] promotes grammar-based modular language development through inheritance, embedding, and aggregation, yet its feature modeling capabilities remain tied to its own infrastructure. Rascal [35] offers powerful facilities for source code analysis and language prototyping but lacks dedicated SPL engineering support. Melange [26] enables assembly of reusable language modules through merge, slice, and inherit operators, though its composition mechanisms are specific to the EMF technological space. SCOLAR [44, 45] supports language aggregation of black-box components in a low-code platform that follows principles similar to those adopted by product line engineering technologies. LionWeb⁷ is an ecosystem to develop language-oriented systems on the web that handles clients access to functionalities through *models* (graph data structures) and *repositories*. The LionWeb models are designed to interoperate and to reuse existing other pre-existing compatible components.

In all cases, SPL engineering capabilities—when present—are embedded within the workbench and cannot be reused across platforms. Our protocol addresses this by externalizing feature model extraction and configuration logic into an independent server communicating with workbench-specific backends through a standardized message format.

6.3. Protocol-Based Approaches in SW Engineering

Our protocol draws inspiration from the LSP, which decouples language-specific intelligence from editor concerns via a client-server architecture [16]. The Debug Adapter Protocol (DAP)⁸ and Build Server Protocol (BSP)⁹ apply the same principle to debugging and build tooling, respectively.

Bruzzone *et al.* [13] applied this decoupling philosophy to language server generation for language families built with Neveerlang, proposing a modular generation process paired with an automated LSP plugin generator for multiple editors. They also introduce the *variant-oriented programming* paradigm and a cross-artifact coordination layer for managing interdependent software variants. While their approach targets editing support (*i.e.*, language servers and plugins), our protocol targets a complementary concern: feature model extraction, construction, and interactive configuration. Both could be combined for end-to-end, workbench-agnostic LPL tooling. To the best of our knowledge, no prior work other than ours has proposed a

⁷<https://lionweb.io/>

⁸<https://microsoft.github.io/debug-adapter-protocol/>

⁹<https://build-server-protocol.github.io/>

protocol-based decoupling specifically for SPL or LPL feature model extraction.

6.4. Feature Model Analysis and Configuration

Benavides *et al.* [9] survey automated analysis operations on feature models, including detection of dead features, false-optional features, and void models. Our protocol delegates all validation logic to the SPL server, allowing any such technique to be integrated within its implementation.

Batory [6] established a formal link between feature models, context-free grammars, and propositional formulas, enabling SAT-based [42] and BDD-based [25] analysis and configuration. Our use of Prolog as the reasoning engine offers an alternative that naturally supports constraint propagation and interactive exploration, while remaining general enough to accommodate other solvers. Czarnecki and Wasowski [25] additionally investigated feature model merging and specialization; although our protocol currently focuses on single-model extraction, the message-based architecture could support composition operations in future work.

Bruzzone *et al.* [15] address a complementary downstream challenge: prioritizing configurations when their combinatorial explosion makes exhaustive exploration infeasible. They rank features via centrality measures on graph structures extracted from the Rust compiler, with validity guaranteed by a SAT solver over a CNF encoding. While their work focuses on selecting relevant configurations for analysis and testing, our protocol targets the upstream extraction and interactive configuration of feature models; the two could be combined to guide configuration-aware optimization.

6.5. Feature-Oriented and Compositional Language Development

Feature-oriented software development (FOSD) [4] treats features as first-class modular units. Tools such as AHEAD [7] and FeatureHouse [3] support composition of feature modules across multiple artifact types; our protocol complements them by providing a standardized mechanism for extracting the variability structure governing such compositions.

Delta-oriented programming [48] offers an alternative paradigm in which a core product is incrementally modified through delta modules, as in DeltaJ. While delta-oriented programming focuses on how artifacts are composed, our protocol focuses on how their dependencies are represented, extracted, and validated in a workbench-agnostic manner. More recently, *variant-oriented programming* [13] has been proposed for managing interdependent software variants arising when language artifacts are reused across a language family. Through a cross-artifact coordination layer it ensures variant consistency—complementary to our protocol, which governs how variability is extracted and configured at the feature model level.

FeatureIDE [51] integrates feature modeling, configuration, and code generation within Eclipse; AiDE [29] builds on it for LPL engineering with Neverlang. Our protocol generalizes the role of FeatureIDE by decoupling feature model management from both IDE and language workbench, enabling heterogeneous environments to interoperate through a shared protocol.

7. Conclusion

We have presented a generalized, workbench-agnostic protocol for bottom-up feature model extraction, with a particular focus on Language Product Lines. The protocol abstracts from specific language workbenches and IDEs by representing variability through atoms and their provides/requires relations, and by clearly separating concerns between SPL server, backend, and client.

We realized this protocol in a prototype that extracts a feature model from Neverlang language artifacts and exposes it through a Cytoscape-based graphical frontend. The case study illustrates that the same SPL infrastructure can be reused across different technological spaces, while backends remain thin adapters over existing artifacts.

Our current implementation targets a single workbench and a limited set of artifacts; broader validation across additional workbenches and technological spaces is left for future work, together with the integration of richer analysis operations and configuration support. We envision our protocol as complementary to protocol-based language tooling such as LSP, and plan to explore their combination for end-to-end, workbench-agnostic LPL support.

References

- [1] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. 2014. Extraction and Evolution of Architectural Variability Models in Plugin-Based Systems. *Software and Systems Modeling* 13, 4 (July 2014), 1367–1394.
- [2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts.
- [3] Sven Apel and Christian Kästner. 2009. An Overview of Feature-Oriented Software Development. *Journal of Object Technology* 8, 5 (July 2009), 49–84.
- [4] Sven Apel, Alexander von Thein, Philipp Wendler, Armin Größlinger, and Firk Beyer. 2013. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, Betty H. Chang and Klaus Pohl (Eds.). IEEE, San Francisco, CA, USA, 482–491.
- [5] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Reengineering Legacy Applications into Software Product Lines: A Systematic Mapping. *Empirical Software Engineering* 22, 6 (Feb. 2017), 2972–3016.
- [6] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 4th International Conference on Software Product-Line (SPLC'05) (LNCS 3714)*, Henk Obbink and Klaus Pohl (Eds.). Springer, Rennes, France, 7–20.

- [7] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering* 30, 6 (June 2004), 355–371.
- [8] Maurice H. ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini. 2019. Static Analysis of Featured Transition Systems. In *Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC'19)*, Laurence Duchien and Thomas Thüm (Eds.). ACM, Paris, France, 39–51.
- [9] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (Sept. 2010), 615–636.
- [10] Lorenzo Bettini. 2016. Implementing Type Systems for the IDE with Xsemantics. *Journal of Logical and Algebraic Methods in Programming* 85, 5 (Aug. 2016), 655–680.
- [11] Lorenzo Bettini and Pierluigi Crescenzi. 2015. An Eclipse IDE for Teaching Java--. In *Proceedings of 10th International Conference on Software Technology (ICSOFT'15) (Communications in Computer and Information Science 586)*, Pascal Lorenz, Jorge Cardoso, Leszek A. Maciaszek, and Marten van Sinderen (Eds.). Springer, Colmar, France, 63–78.
- [12] Federico Bruzzone and Walter Cazzola. 2026. Meta-Monomorphizing Specializations. arXiv:2602.12973 [cs.PL] <https://arxiv.org/abs/2602.12973>
- [13] Federico Bruzzone, Walter Cazzola, and Luca Favalli. 2026. Code Less to Code More: Streamlining Language Server Protocol and Type System Development for Language Families. *Journal of Systems and Software* 231 (Jan. 2026). doi:10.1016/j.jss.2025.112554
- [14] Federico Bruzzone, Walter Cazzola, and Luca Favalli. 2026. From Separate Compilation to Sound Language Composition. arXiv:2602.03777 [cs.PL] <https://arxiv.org/abs/2602.03777>
- [15] Federico Bruzzone, Walter Cazzola, and Luca Favini. 2026. Prioritizing Configuration Relevance via Compiler-Based Refined Feature Ranking. arXiv:2601.16008 [cs.PL] <https://arxiv.org/abs/2601.16008>
- [16] Hendrick Bündler and Herbert Kuchen. 2019. Towards Multi-editor Support for Domain-Specific Languages Utilizing the Language Server Protocol. In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19)*, Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic (Eds.). Springer, Prague, Czech Republic, 225–245.
- [17] Bruno B. P. Cafeo, Elder Cirilo, Alessandro Garcia, Francisco Dantas, and Jaejoon Lee. 2016. Feature Dependencies as Change Propagators: An Exploratory Study of Software Product Lines. *Information and Software Technology* 69 (Jan. 2016), 37–49.
- [18] Walter Cazzola. 2012. Domain-Specific Languages in Few Steps: The Neverlang Approach. In *Proceedings of the 11th International Conference on Software Composition (SC'12) (Lecture Notes in Computer Science 7306)*, Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book (Eds.). Springer, Prague, Czech Republic, 162–177.
- [19] Walter Cazzola and Luca Favalli. 2022. Towards a Recipe for Language Decomposition: Quality Assessment of Language Product Lines. *Empirical Software Engineering* 27, 4 (April 2022). doi:10.1007/s10664-021-10074-6
- [20] Walter Cazzola and Luca Favalli. 2023. The Language Mutation Problem: Leveraging Language Product Lines for Mutation Testing of Interpreters. *Journal of Systems and Software* 195 (Jan. 2023). doi:10.1016/j.jss.2022.111533
- [21] Walter Cazzola and Diego Mathias Olivares. 2016. Gradually Learning Programming Supported by a Growable Programming Language. *IEEE Transactions on Emerging Topics in Computing* 4, 3 (Sept. 2016), 404–415. doi:10.1109/TETC.2015.2446192 Special Issue on Emerging Trends in Education.
- [22] Zhiqun Chen. 2000. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, Reading, MA, USA.
- [23] Alain Colmerauer and Philippe Roussel. 1996. The Birth of Prolog. In *History of Programming Languages*, Thomas J. Bergin and Richard G. Gibson (Eds.). ACM, Chapter VII, 331–367.
- [24] Michelle L. Crane and Juergen Dingel. 2005. UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05) (Lecture Notes in Computer Science 3713)*, Lionel Briand and Clay Williams (Eds.). Springer, Montego Bay, Jamaica, 97–112.
- [25] Krzysztof Czarnecki and Andrzej Wąsowski. 2007. Feature Diagrams and Logics: There and Back Again. In *Proceedings of the 11th International Software Product Line Conference (SPLC'07)*, Tomoji Kishi and Dirk Muthig (Eds.). IEEE, Kyoto, Japan, 23–34.
- [26] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: a Meta-Language for Modular and Reusable Development of DSLs. In *Proceedings of the 8th International Conference on Software Language Engineering (SLE'15)*, Davide Di Ruscio and Markus Völter (Eds.). ACM, Pittsburgh, PA, USA, 25–36.
- [27] Florian Drux, Nico Jansen, and Bernhard Rumpe. 2022. A Catalog of Design Patterns for Compositional Language Engineering. *Journal of Object Technology* 21, 4 (Oct.

- 2022), 4:1–13. doi:10.5381/jot.2022.21.4.a4 Special Issue dedicated to Antonio Vallecillo on his 60th Birthday.
- [28] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: Library-Based Syntactic Language extensibility. In *Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA'11)*. ACM, Portland, Oregon, USA, 391–406.
- [29] Luca Favalli, Thomas Kühn, and Walter Cazzola. 2020. Neverlang and FeatureIDE Just Married: Integrated Language Product Line Development Environment. In *Proceedings of the 24th International Software Product Line Conference (SPLC'20)*, Philippe Collet and Sarah Nadi (Eds.). ACM, Montréal, Canada, 285–295.
- [30] Elisabeth N. Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2013. On Extracting Feature Models from Sets of Valid Feature Combinations. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE'13) (LNCS 7793)*, Vittorio Cortellessa and Daniel Varrò (Eds.). Springer, Rome, Italy, 53–67.
- [31] Jacques Herbrand. 1968. *Logical Writings*. Springer. «Écrits Logiques» translated in English by Jean van Heijenoort.
- [32] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.
- [33] Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. 2010. Pure and Declarative Syntax Definition: Paradise Lost and Regained. In *Proceedings of ACM Conference on New Ideas in Programming and Reflections on Software (Onward! 2010)*. ACM, Reno-Tahoe, Nevada, USA.
- [34] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeff Palm, and Bill Griswold. 2001. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01) (LNCS 2072)*, Jørgen Lindskov Knudsen (Ed.). Springer-Verlag, Budapest, Hungary, 327–353.
- [35] Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. EASY Meta-Programming with Rascal. In *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering III (GTTSE'09) (Lecture Notes in Computer Science 6491)*, João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.). Springer, Braga, Portugal, 222–289.
- [36] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2010. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer* 12, 5 (Sept. 2010), 353–372.
- [37] Charles W. Krueger. 2001. Easing the Transition to Software Mass Customization. In *Proceedings of the 1st International Workshop on Software Product-Family Engineering (PFE'01) (Lecture Notes in Computer Science 2290)*, F. van der Linden (Ed.). Springer, Bilbao, Spain, 282–293.
- [38] Thomas Kühn and Walter Cazzola. 2016. Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines. In *Proceedings of the 20th International Software Product Line Conference (SPLC'16)*, Rick Rabiser and Bing Xie (Eds.). ACM, Beijing, China, 50–59.
- [39] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. 2015. Choosy and Picky: Configuration of Language Product Lines. In *Proceedings of the 19th International Software Product Line Conference (SPLC'15)*, Goetz Botterweck and Jules White (Eds.). ACM, Nashville, TN, USA, 71–80.
- [40] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves le Traon. 2015. Automating the Extraction of Model-Based Software Product Lines from Model Variants. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*, Myra Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE, Lincoln, NE, USA, 396–406.
- [41] Nicholas D. Matsakis. 2018. Maximally Minimal Specialization: Always Applicable impls. Blog Post. <https://smallcultfollowing.com/babysteps/blog/2018/02/09/maximally-minimal-specialization-always-applicable-impl>
- [42] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.—Software Product Lines Online Tools. In *Companion Proceedings of the 24th Conference on Object-Oriented Programming Systems Languages and Applications (Companion OOPSLA'09)*. ACM, Orlando, FL, USA, 761–762.
- [43] Pierre M. Nugues. 2014. *An Introduction to Language Processing with Perl and Prolog* (second ed.). Springer.
- [44] Jérôme Pfeiffer and Andreas Wortmann. 2021. Towards the Black-Box Aggregation of Language Components. In *Proceedings of the 3rd International Workshop on Modelling Language Engineering (MLE'21)*, Federico Ciccozzi, Thomas Degueule, Romina Eramo, and Sébastien Gérard (Eds.). IEEE, Fukuoka, Japan, 576–585.
- [45] Jérôme Pfeiffer and Andreas Wortmann. 2023. A Low-Code Platform for Systematic Component-Oriented Language Composition. In *Proceedings of the 16th International Conference on Software Language Engineering (SLE'23)*, Thomas Degueule and Elizabeth Scott (Eds.). ACM, Cascais, Portugal, 208–213.
- [46] Klaus Pohl, Klaus Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

- [47] J. Alan Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (Jan. 1965), 23–41.
- [48] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *Proceedings of the 14th International Software Product Line Conference (SPLC'10) (Lecture Notes on Computer Science 6287)*, Jan Bosch and Jaejoon Lee (Eds.). Springer, Jeju Island, South Korea, 77–91.
- [49] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S. Baliga, Jonathan T. Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. 2003. Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks. *Genome Research* 13, 11 (Nov. 2003), 2498–2504.
- [50] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, Harald Gall and Nenad Medvidović (Eds.). IEEE, Waikiki, Honolulu, Hawaii, 461–470.
- [51] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 79, 1 (Jan. 2014), 70–85.
- [52] Laurence Tratt. 2008. Domain Specific Language Implementation Via Compile-Time Meta-Programming. *ACM Transactions on Programming Languages and Systems* 30, 6 (Oct. 2008), 31:1–31:40.
- [53] Aaron Turon. 2017. Shipping Specialization: A Story of Soundness. Blog Post. <https://aturon.github.io/blog/2017/07/08/lifetime-dispatch/>.
- [54] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A Framework for Feature-Oriented Language Development. *Computer Languages, Systems & Structures* 43, 3 (Oct. 2015), 1–40. doi:10.1016/j.cl.2015.02.001
- [55] Edoardo Vacchi, Walter Cazzola, Benoît Combemale, and Mathieu Acher. 2014. Automating Variability Model Inference for Component-Based Language Implementations. In *Proceedings of the 18th International Software Product Line Conference (SPLC'14)*, Patrick Heymans and Julia Rubin (Eds.). ACM, Florence, Italy, 167–176.
- [56] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoît Combemale. 2013. Variability Support in Domain-Specific Language Development. In *Proceedings of 6th International Conference on Software Language Engineering (SLE'13) (Lecture Notes on Computer Science 8225)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer, Indianapolis, USA, 76–95.
- [57] Markus Völter, Daniel Ratiu, Bernhard Schätz, and Bernd Kolb. 2012. mbeddr: an Extensible C-Based Programming Language and IDE for Embedded Systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH'12)*. ACM, Tucson, AZ, USA, 121–140.
- [58] Guido H. Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. 2014. Language Design with the Spoofox Language Workbench. *IEEE Software* 31, 5 (Sept./Oct. 2014), 35–43.
- [59] Huilin Ye and Hanchang Liu. 2005. Approach to Modelling Feature Variability and Dependencies in Software Product Lines. *IEE Proceedings—Software* 152, 3 (June 2005), 101–109.