Empirical Software Engineering (2022) 27:82
https://doi.org/10.1007/s10664-021-10074-6

Towards a recipe for language decomposition: quality assessment of language product lines

Walter Cazzola[1] · Luca Favalli[1]

Accepted: 3 November 2021

Abstract

Programming languages are complex systems that are usually implemented as monolithic interpreters and compilers. In recent years, researchers and practitioners gained interest in product line engineering to improve the reusability of language assets and the management of variability-rich systems, introducing the notions of language workbenches and language product lines (LPLs). Nonetheless, language development remains a complex activity and design or implementation flaws can easily waste the efforts of decomposing a language specification into language features. Poorly designed language decompositions result in high inter-dependent components, reducing the variability space of the LPL system and its maintainability. One should detect and fix the design flaws posthaste to prevent these risks while minimizing the development overhead. Therefore, various aspects of the quality of a language decomposition should be quantitatively measurable through adequate metrics. The evaluation, analysis and feedback of these measures should be a primary part of the engineering process of a LPL. In this paper, we present an exploratory study trying to capture these aspects by introducing a design methodology for LPLs; we define the properties of a good language decomposition and adapt a set of metrics from the literature to the framework of language workbenches. Moreover, we leverage the AiDE 2 LPL engineering environment to perform an empirical evaluation of 26 Neverlang-based LPLs based on this design methodology. Our contributions form the foundations of a design methodology for Neverlang-based LPLs. This methodology is comprised of four different elements: i) an engineering process that defines the order in which decisions are made, ii) an integrated development environment for LPL designers and iii) some best practices in the design of well-structured language decomposition when using Neverlang, supported by iv) a variety of LPL metrics that can be used to detect errors in design decisions.

Communicated by: Philippe Collet, Sarah Nadi, and Christoph Seidl

This article belongs to the Topical Collection: *Software Product Lines and Variability-rich Systems (SPLC)*

✉ Walter Cazzola
cazzola@di.unimi.it

Luca Favalli
favalli@di.unimi.it

[1] Computer Science Department, Università degli Studi di Milano, Milan, Italy

Published online: 07 April 2022

Keywords Language product lines · Feature modularity · Software variability · Domain specific languages

1 Introduction

Domain-specific languages (DSL) are programming languages capable of expressing aspects and abstractions of a specific application domain. Incorporating DSLs into the design of complex systems enables the direct interaction of domain experts with the development of defined system aspects (Combemale et al. 2014). Despite being designed for a precise purpose and defined audiences, many DSLs and programming languages in general share with each other several commonalities either from an abstract syntax, a concrete syntax or a semantics standpoint (Méndez-Acuña et al. 2016; Zschaler et al. 2009; de Lara and Guerra 2012). For instance, syntactically different constructs may share the same semantics (*e.g.*, an *if* statement in Java versus Python) or, conversely, share the same syntax but yield different semantics (*e.g.*, a sequential versus a parallel *for* loop). Such commonalities can improve reuse and minimize the efforts of from-scratch implementations when wisely leveraged by language developers (Méndez-Acuña et al. 2016). The advantage of fine-grained modularity in language workbenches (Erdweg et al. 2013; Fowler 2005)—*i.e.*, tools that provide high-level mechanisms for the implementation of languages and their usage in a unified environment—is twofold and enables the construction of both syntax and semantic variability-rich systems. In this context, the research community focused on defining a systematic reuse process by embracing feature-oriented programming and software product lines introducing the notion of LPLs (Kühn et al. 2015). A language feature encapsulates the subset of a language specification representing a functionality offered by a DSL: as stated in Vacchi and Cazzola (2015), each subset represents either language constructs—e.g., for loop—or language concepts (without concrete syntax)—e.g., scope and type system. In LPL engineering, language features are implemented independently and in separate modules as a result of a language decomposition combined into a compiler or interpreter through a configuration mechanism and finally used by application domain experts in a tight feedback loop (Favalli et al. 2020). To summarize, each product of an LPL is a compiler or interpreter for a DSL and the collection of all the DSLs produced by an LPL is called *language family*. To serve the purpose of several users, LPLs become complex systems with possibly hundreds of features and an exponential number of valid configurations. Nonetheless, LPLs should provide both flexibility and ease of use by ensuring the validity of products without limiting the variability space of the language family.

Successful product line engineering requires the definition of highly cohesive features with low coupling as any other kind of software engineering (Parnas 1971; Troy and Zweben 1981; Macro and Buxton 1987; Fenton 1991; Lanza and Marinescu 2006). Researchers are focusing on the development of integrated development environments (IDEs) which provide tools supporting LPLs system designer, the systematic derivation of sound language definitions and implementations (Visser et al. 2014) and the automatic generation of IDE services and debugging (Butting et al. 2018; Kühn et al. 2019; Favalli et al. 2020). Modern language workbenches do not directly address a formal specification for the quality in the design of language features, especially with respect to their modularity flaws. Despite not causing any errors from a user perspective, modularity flaws may result in highly inter-dependent modules and crosscutting features which are known to reduce the flexibility and maintainability of program families (Colyer et al. 2004). In language engineering separate constructs should be either independent or implemented as a unique feature. Yet, we argue that such specification could be defined with minimal effort thanks to the amount of meaningful information that a full-fledged language workbench with LPL support accesses at compile time. State-of-the-art language workbenches could help improve the quality of the design of language decompositions and of DSLs overall, by framing such information in well defined metrics and providing them to the language designer.

For each aspect we show how it can be tackled with the Neverlang language workbench and the Neverlang-based LPL engineering (LPLE) framework AiDE 2. This work is a follow up to our original contribution (Favalli et al. 2020) that introduced the LPLE process and an environment supporting it. This contribution further elaborates on Favalli et al. (2020) by defining the qualities of a good language decomposition developed following that process. The output of our research is the definition of the properties that a well designed language decomposition in Neverlang modules should have and a set of metrics for the measurement of those properties. Last but not least important, we also show how this evaluation can be easily integrated in a LPL engineering process to guide the design of language decompositions.

This work is validated by answering the following research questions:

RQ$_1$. What are the properties of a language decomposition in Neverlang?

RQ$_2$. How can errors in design decisions be detected in Neverlang LPLs?

To answer these research questions we perform an empirical evaluation on 26 Neverlang LPLs without applying any changes to either Neverlang or AiDE 2 but rather accessing compile-time information already available in any Neverlang LPL. What remains of this paper is structured as follows. Section 2 presents the foundations, terminology and technology. Section 3 introduces the design methodology. Section 4 describes the experiment and its result. Finally in Sections 5 and 6 we give an overview of the related work and draw our conclusions.

2 Background

In this section, we present the background and concepts of feature modeling, language workbenches and language product lines through the used tools and methods: Neverlang and AiDE 2.

2.1 Software product lines and feature modeling with FeatureIDE

Variability-rich software systems development applies concepts from product line engineering. This practice is usually referred to with the terms of feature-oriented programming and software product line (SPL) engineering. An SPL is a family of software products whose commonalities and differences can be described in terms of their features. SPL engineering combines concepts from domain engineering for the design and implementation of software artifacts with concepts from application engineering to create products from selected features (Apel et al. 2013). The SPL engineering activity often involves the feature modeling activity *i.e.*, the creation and maintenance of a feature model (FM) whose concept was first introduced as part of the FODA method (Kang et al. 1990) and expresses the variability of the system in terms of its features and of the dependencies among these features. In SPLs using the FM formalism, the deployment of a software product is closely tied to the usage of a FM for the definition of valid feature subsets called *configurations*. Given a valid configuration, any feature that belongs to the defined subset of features is said to be *active* for that configuration; all other features of the FM are *inactive*. The validity of a configuration is based on the dependencies between features that the FM declares. The FM structure can implicitly imply feature dependencies by defining mandatory features, optional features, or groups, and alternative features, as well as, the simple parent-child relationship—a feature can be active only if all parent features are also active. In addition, dependencies can be defined explicitly by the cross-tree constraint mechanism—*i.e.*, Boolean expression defining which terms are features from the FM; each term is set to true if the corresponding feature is active in the current configuration and false otherwise. If the truth value of any cross-tree constraint is false for a configuration then that configuration is invalid. Both implicit and explicit feature dependencies may result in dead features (that can never be active), false-optional features (that are marked as optional but are mandatory), and *atomic sets*—*i.e.*, sets of features such that all features are active in the same configuration, or none is. The quality of SPLs can be improved by performing static analysis of FMs for the detection of such anomalies. This is an active research area and includes structural (Benavides et al. 2010) and behavioral (ter Beek et al. 2019) approaches.

FeatureIDE (Thüm et al. 2014; Meinicke et al. 2016; Meinicke et al. 2017) is a SPL development environment that copes with all aspects of the development of SPLs. It supports the FM construction, the management of software artifacts, the configuration and product derivation.

SPL engineering support in FeatureIDE encompasses: 1) the *Feature Model Editor* for the creation, visualization and tracing of FMs, concrete and abstract features, feature dependencies and cross-tree constraints; 2) the *Configuration Editor* for the creation, modification, and validation of feature configurations (Pereira et al. 2016); and 3) various *Composers* for the derivation of product variants from a given valid feature configuration.

As an example, Fig. 1 showcases a FM for the language family of LogLang built with FeatureIDE.

FeatureIDE maintains consistency between all the different views during all phases of the development process. This includes all the editors as well as the *Feature Model Outline* which provides basic information and metrics about the FM and the variability space of the SPL. The Configuration Editor guides the development of valid configurations by checking their validity to feature dependencies and cross-tree constraints. Finally, a composer generates the final product variant for a given configuration by combining the active features from that configuration. We created AiDE 2 in our previous work (Favalli et al. 2020) by integrating FeatureIDE with AiDE to bring LPL support to FeatureIDE.

2.2 Language product lines

The development of families of programming languages and DSLs has gained popularity among researchers and practitioners, e.g., Ng et al. (2011), Kühn et al. (2014), Crane and Dingel (2005), and Zschaler et al. (2009). Similar to other software, DSL interpreters and compilers can be designed around the concept of product line, in relation to the features they include in their specification. SPL engineering applied to the development of programming languages is called LPL (Kühn et al. 2015) engineering and it is becoming a popular topic (Méndez-Acuña et al. 2016; Völter 2011; Wende et al. 2009; Grönniger and Rumpe 2010) in the research on DSLs because it eases the language development process (White et al. 2009). DSLs can be modeled as a family of languages rather than stand-alone monolithic implementations when developed in a LPL fashion. For instance, several works (Tratt 2008; Crane and Dingel 2005; Vacchi et al. 2013; Méndez-Acuña et al. 2017) showed that the variants of state machine languages can be modeled as a single family of programming

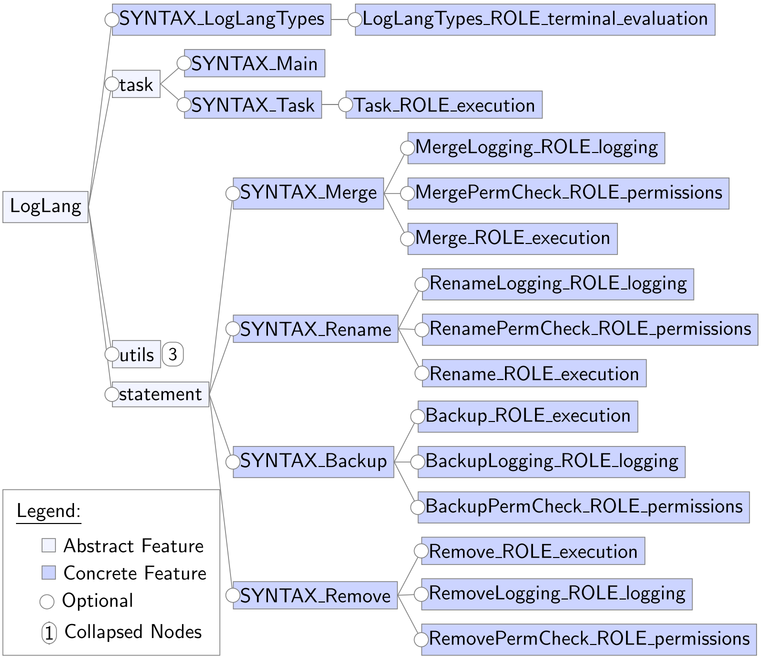


Fig. 1 Example feature model for the family of LogLang DSLs (Cazzola and Poletti 2010). Each sub-tree models the variability of a different language construct. For instance the syntax of the `rename` statement can be composed with any of three different semantics implementations, each represented by a leaf in the FM

languages. On the one side, specialized versions of full-fledged programming languages can be employed in case of security purposes (*e.g.,* Java Card Chen 2000) or teaching (Bettini and Crescenzi 2015; Cazzola and Olivares 2016; Hermans 2020). Language extensions, on the other hand, can be exploited to embed new language features into an existing programming language, such as type-checked SQL queries (Erdweg et al. 2011).

LPLs can be created by using either a *top-down* or a *bottom-up* approach (Kühn and Cazzola 2016). In the former approach, the feature model is created first as a result of the domain analysis which defines and documents the desired variability of the product family (Pohl et al. 2005; Apel et al. 2013), then the features are mapped to the language artifacts and finally language variants are developed though a configuration process. In the latter, the language developer creates individual language artifacts, then the FM is automatically generated from those artifacts. Given these definitions, the *top-down* approach is similar to the *proactive* SPLE adoption technique whereas the *bottom-up* approach employs aspects of both the *extractive* and *reactive* techniques (Krueger 2001). Henceforth, we will focus on the *bottom-up* approach for LPL engineering.

Figure 1 showcases the FM generated in bottom-up fashion for the family of LogLang variants. LogLang (Cazzola and Poletti 2010) is a simple DSL that describes tasks for a log rotating tool similar to the Unix logrotate utility with a modular Neverlang implementation. An interpreter—*i.e.*, a product of the LogLang language family—can be generated by providing a valid configuration to the composer. We denote a language variant to be *viable* if its language recognizes/evaluates the selected language constructs with the expected semantics.

```
module Backup {
  reference syntax {
    provides { Backup: backup, statement; Cmd: statement; }
    requires { String; }
    Backup ← "backup" String String;
    Cmd    ← Backup;
    categories  : Keyword = { "backup" };
    in-buckets  : $1 ← { Files }, $2 ← { Files };
    out-buckets : $1 → { Files }, $2 → { Files };
  }
  role(execution) {
    0 .{
      String src  = $1.string, dest = $2.string;
      $$FileOp.backup(src, dest);
    }.
  }
}
slice BackupSlice {
  concrete syntax from Backup
  module Backup with role execution
  module BackupPermCheck with role permissions
}

language LogLang {
  slices BackupSlice RemoveSlice RenameSlice
         MergeSlice Task Main LogLangTypes
  endemic slices FileOpEndemic PermEndemic
  roles syntax < terminal-evaluation < permissions : execution
}
```

Listing 1 Syntax and semantics for the backup task

2.3 Neverlang and AiDE in a nutshell

Neverlang (Cazzola 2012; Cazzola and Vacchi 2013; Vacchi and Cazzola 2015) is a language workbench for the modular development of programming languages. Language components, called slices, embody the concept of language features and are developed as separate units that can be independently compiled, tested, and distributed, enabling developers to share and reuse the same units across different language implementations. The basic development unit is introduced by the keyword **`module`**. A module may contain a **`reference syntax`** definition with one or more productions and/or roles. Each role, introduced by the keyword **`role`**, defines a compilation phase by declaring semantic actions that should be executed when some syntax is recognized, as prescribed by the *syntax-directed translation* technique (Aho et al. 1986). Semantic actions are also responsible for the definition and evaluation of the attributes characteristics of the attribute grammar: attributes to which semantic actions perform an assignment are inherited or synthesized depending on how they are defined. Henceforth we will refer to both inherited and synthesized attributes as *provided* attributes for brevity, whereas *required* attributes are those whose values can be accessed during the evaluation of a semantic action. Both required and provided attributes—*i.e.*, any attributes that are *referenced* in a semantic action—are source of dependencies between modules, since their value will be generated or accessed respectively by different modules. Syntactic definitions and semantic roles are tied together using slices.

Listing 1 illustrates the implementation of the `Backup` feature of the LogLang LPL. The `Backup` module declares a reference syntax for the backup task (lines 2-10). The reference syntax of a module also piggybacks (Kühn et al. 2019) information for basic IDE services, such as syntax highlighting (line 7) and code-completion (lines 8-9), to automatically provide IDE support for languages in which they are included. Semantic actions are attached to nonterminals of the productions (lines 12-15) by referring to their position in the grammar: numbering starts with 0 and grows from the top left to the bottom right.[1] Thus, the `Backup` nonterminal on line 5 is referred to as `$0` and the two `String` nonterminals on the right-hand side of the production as `$1` and `$2`, respectively. Attributes are accessed from nonterminals using the same criterion by dot notation as in line 13. In contrast, the `BackupSlice` (lines 18-22) declares that it will promote the reference syntax from the `Backup` module to concrete syntax for our language (line 19) and combine it with the semantics actions from two separate roles of two different modules (lines 20-21). Finally, the **`language`** descriptor (lines 24-29) indicates which slices should be composed to generate the language interpreter and the IDE (lines 25-26). Therefore, composition in Neverlang is twofold: i) between modules, which yields slices, and ii) between slices, which yields a language implementation. Composition is also supported through bundles that behave just as languages but they can be embedded in other languages. The grammars are merged to generate the complete language parser. Any gaps in the grammar can be filled by using the **`rename`** mechanism: any nonterminal can be renamed to match a nonterminal provided by another production in the grammar. Renames will be discussed further in Section 4 with some usage examples. Semantic actions are performed with respect to the parse tree of the input program; roles are executed in sequence and traversal options specified in the **`roles`** clause (line 28) of the **`language`** descriptor, e.g., `permission` is executed after `parsing` and `terminal-evaluation`. Besides, the **`language`** clause can declare **`endemic slices`** whose instances are shared across multiple compilation phases (line 27).[2]

Neverlang supports LPL engineering thanks to AiDE (Vacchi et al. 2013; Vacchi et al. 2014). AiDE is a variability management tool tailored for the development of LPLs. It extracts information provided by Neverlang modules (lines 3-4 of Listing 1) to determine the language features and their dependencies and synthesizes the corresponding FM for a given language family (Vacchi and Cazzola 2015) in a bottom-up fashion using the algorithm introduced in (Vacchi et al. 2013) and refined in Favalli et al. (2020). Through its graphical user interface, the user can explore the FM, choose language features, create a language variant, and test it. Moreover, AiDE tracks all unresolved dependencies—*i.e.*, all open nonterminals in the current configuration—and guides the renaming mechanism to bind them to other nonterminals already in the current configuration. The flexibility of LPLs is desirable for all actors of the development process. From a stakeholder perspective, the ability to create several language variants from the same language assets through a FM allows addressing the needs of a large pool of different users with minimum additional investment. From the designer standpoint, providing a language variant in which only a subset of the language features is made available to the developers helps aligning the development team to the same vocabulary of abstractions and in turn improving the maintainability and efficiency of software artifacts. From the final user perspective, a well designed language variant is a DSL

[1] Neverlang also provides a labeling mechanism for productions, so that nonterminals are referred via an offset from such a label, e.g., $BKP[1] is the first nonterminal from the right-hand side of the BKP production.

[2] Please see Vacchi and Cazzola (2015) for further details.

perfectly suited to the domain and stripped of all the low-level implementation concerns. For instance, language constructs, programming practices, chosen paradigm and compiler can all impact energy consumption (Pinto et al. 2014; Trefethen and Thiyagaligam 2013; Lima et al. 2016). This is relevant in contexts under strict energy consumption constraints, such as embedded systems (Tiwari et al. 1994; Tavares et al. 2008). A restricted language variant can ensure only the most efficient constructs are used by removing the less efficient ones from the language. Language restrictions can also prevent dangerous behaviour to ensure safety (Hatton 2007) and to improve overall software quality (Basalaj 2008).

AiDE also relies on a quick feedback loop for the behavior of the language variant under construction by allowing for the dynamic update of a language variant during the configuration phase. AiDE updates a language descriptor with the newest set of active features whenever a valid configuration is deployed.

AiDE is currently integrated with FeatureIDE and the Gradle build tool[3] to ease the generation of Java artifacts and the deployment of an archive containing the runtime for the language variant. FeatureIDE also enables to perform the variability space analysis of any FM. This version of AiDE is the latest available version and it is called AiDE 2 (Favalli et al. 2020).

3 Towards a Design Methodology for Language Product Lines

We hereby describe the concepts that enable the development and evaluation of well designed Neverlang-based language families. Our contribution in this regard is manifold and includes i) the bottom-up LPL engineering process, ii) the properties of a well designed language decomposition and iii) the metrics to measure the quality of LPLs. Moreover, iv) we lay the foundations of a design methodology that encompasses all other contributions, as well as a dedicated IDE, under a unified vision by following Parnas' steps. According to Parnas (1971), a design methodology for any software system should account for five design aspects of that system:

1) the order in which decisions are made;
1) what constitutes good structure for a system;
1) methods of detecting errors in design decisions;
1) specification techniques;
1) tools for system designers.

Notice that the *system* (points 2 and 5) is a LPL in our case and that each point is instantiated as one of the contributions we presented. Point 1 is the engineering process. Point 5 is the LPL engineering environment. Both points were firstly presented in Favalli et al. (2020) and are hereby substantially extended to suit our vision of a LPL design methodology. Points 2 and 3 will be addressed in this work by further elaborating on the concepts proposed in Favalli et al. (2020). More precisely, we address point 2 by introducing a set of desired properties for LPLs and their language components and point 3 by introducing metrics for the evaluation of LPLs and investigating any ideal and threshold values for those metrics. We do not address the topic of specification techniques (point 4) for Neverlang LPLs, which is not directly tied to this work and should be tackled on with a different approach in the future.

[3] https://gradle.org/

All points of the design methodology are deeply interconnected. In fact, the methods for detecting errors in design decision are based on the properties of a well structured system and should be able to determine whether the system achieves these properties or not. Any detected errors affect the order in which decisions are made: the engineering process should be able to adapt so that the errors are corrected before propagating to other elements of the system. Finally, all other points must be supported by the proper tools to be applicable in any real use case.

In this section, we tackle each one of these points: first we show the bottom-up LPL engineering process (Section 3.1), followed by the LPL engineering environment (Section 3.2), the properties of a well designed language decomposition (Section 3.3) and finally the metrics for the detection of design errors in LPLs (Section 3.4).

3.1 The LPL Engineering Process—Point 1

The bottom-up LPL engineering process was first introduced in our original work (Favalli et al. 2020). In this work, we elaborate on that contribution to establish the order in which decisions are made—*i.e.*, point 1 of Parnas' vision. The *Business Process Model and Notation* (BPMN) model[4] in Fig. 2 illustrates the proposed LPL engineering process by showcasing the activity of the three different roles. Each role is involved in either the design, development, deployment or usage of languages from a language family. The process tries to grant continuous flexibility and responsiveness in the development and extension of LPLs while maximizing the separation of concerns among the three roles of *language developer*, *language deployer* and *language user*. In the BPMN, each role is represented as a separate *swim lane* whose overlap may be minimal or none to highlight the differences in their skills. The *language developer* has notions of language development and feature-oriented programming. The *language deployer* is a domain expert with expertise on the specific concepts of the domain at hand without any knowledge in language development. Finally, the *language user* is the final user that can use any language variants and deploy programs without necessarily knowing any low level language implementation details. The three roles use different artifacts, thus improving the applicability of the process in a distributed environment by minimizing conflicts when using a versioning control system. For instance, the *language user* can deploy programs using a language variant while the *language developer* is refactoring a language feature used in the same variant without causing any conflicts since the two roles do not share any artifacts.

The process starts following the initial request of an unspecified stakeholder—which may or may not coincide with the *language user*—asking for an interpreter or a compiler for a given language. The *language developer* (first layer) performs the initial phase of analysis of that language, either against a language specification or an existing monolithic implementation. The result is a language decomposition into language features, *i.e.*, language concepts or constructs with minimal dependency with other features. Language features should be developed and tested separately from each other, possibly by several programmers guided by the same designer with minimal interaction to grant a good language decomposition. Applying the bottom-up approach, the language decomposition also results in the generation of a description of the variability space of the system, *i.e.*, a FM. The FM is the only artifact subject of both the *language developer* and the *language deployer* activities and should be shared—e.g., by a commit to a central repository.

[4] BPMN is a standard graphical notation for the specification of business processes based on flowcharts and activity diagrams.

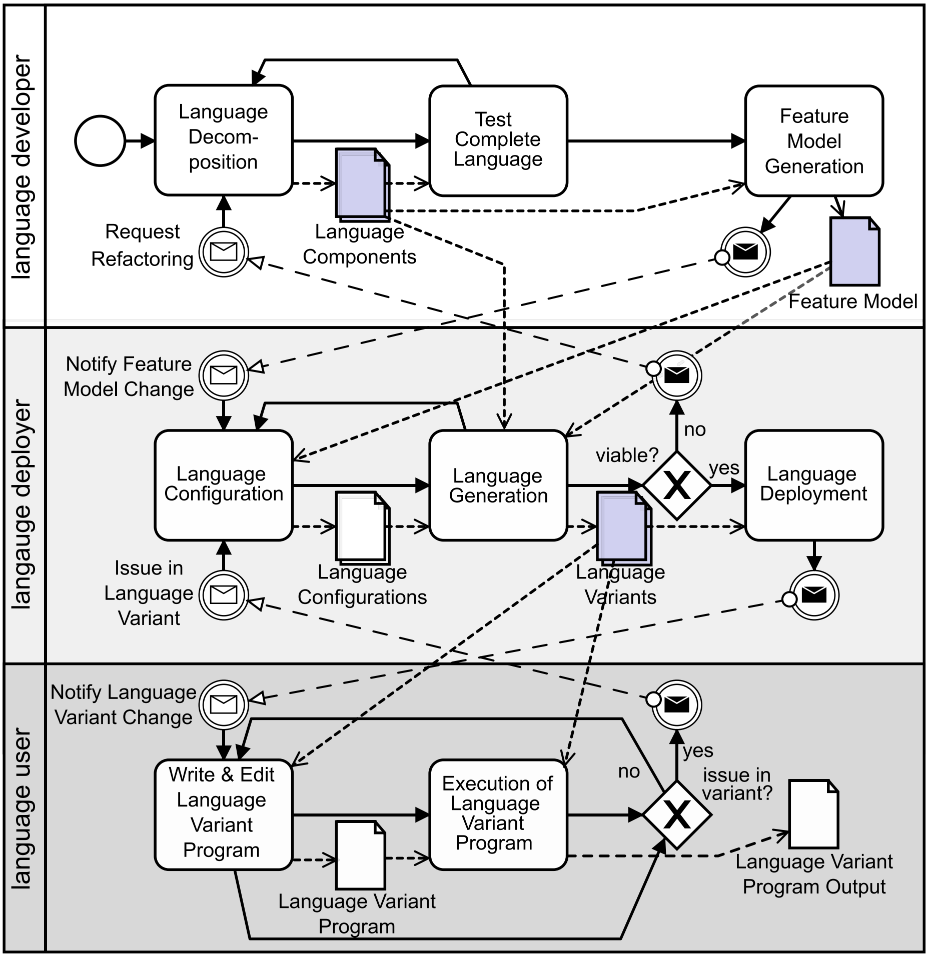


Fig. 2 BPMN model describing the language product line engineering process (Favalli et al. 2020)

Now, the *language deployer* (second layer) can inspect the latest FM to choose and pick (Kühn et al. 2015) language features from the FM, thus creating or revising a *language configuration*. In addition, the configuration for a language variant should also refer to the compilation phases of the target compiler. Finally, variability in programming languages has to cope with additional problems such as the *ripple effect* (Yau et al. 1978), *i.e.*, adding or removing a language feature may provoke the addition or removal of several language features in cascade if no strategy to resolve unfulfilled dependencies on a syntactic level is provided. This usually results in the creation of atomic sets of features (see Section 2). The *language deployer* reports back to the *language developer* requesting to update a set of language features when a language configuration is not viable; either because required language features are missing, are too coarse grained or do not compose. Otherwise, the *language deployer* deploys the language variant from the respective configuration together with its IDE services (Kühn et al. 2019), e.g., an editor with syntax highlighting, code completion and a debugger.

Once a language variant is deployed and committed, *language users* (third layer) can choose any of the available variants in the language family to write and run code in such a language variant. They can report back to the *deployer* if any issue is found either on the syntactic or the semantic level. The *language deployer* then repeats all steps of the configuration process (possibly including their requests for updates to the *developer*) to accommodate the *user* request. The result of this process can be either the update of the corresponding variant or the creation of a new one. From the *user* perspective each language variant is an isolated programming language providing some declared capabilities, including an IDE. The *language deployer* has knowledge of all the variants and of the configurations used to generate them. The *language developer* knows all the language features that constitute the LPL and their implementation.

This LPLE process is designed to allow iterative evolution of language families: when supported by effective modular design in the early stages of development, the three layers can proceed independently and concurrently. In the BPMN model in Fig. 2 only the language components, the FM and the language variants are highlighted, because they are the only artifacts shared between more than one role. Moreover, it should be noticed that by using a language workbench that supports separate compilations such as Neverlang the overlapping in the usage of language components is limited since neither the source nor the binaries of the language components are required to compile a `language` unit. Instead, the *deployer* only needs access to the latest FM to generate and compile a valid language, regardless of how the individual language features are implemented. Nonetheless, all language components binaries must be available when the language compiler is finally used. The interactions between roles are limited to: i) requests from *language user* to *language deployer* and from *language deployer* to *language developer* when the BPMN model is traversed from bottom to top; ii) update notifications from *language developer* to *language deployer* and from *language deployer* to *language user* when the BPMN model is traversed from top to bottom. The proposed LPLE process outlines the activity of all the roles involved in the development of LPLs and *the order in which decisions are made* and thus satisfies point 1 of Parnas' vision.

3.2 The LPL engineering environment—Point 5

All phases of the engineering process must be supported by a dedicated *tool for system designers* to satisfy point 5 of Parnas' vision. As we introduced in Favalli et al. (2020), our approach is the combination of the state-of-the-art in SPL (FeatureIDE) and LPL (AiDE) engineering dubbed AiDE 2. Using the same environment for the design, development, deployment and usage of languages reduces the implementation efforts for dedicated IDEs for each language variant while ensuring a quick feedback loop and rapid deployment. Neverlang embraces this process by providing a *full loop* in which the tools used to deploy the Neverlang ecosystem are the same used to deploy the ecosystem for any Neverlang-based product: Neverlang is bootstrapped and its IDE is generated using **`categories`**, **`in-buckets`** and **`out-buckets`** as in Listing 1.

The *language developer* uses a Neverlang distribution with a compiler and an Eclipse-based IDE developed using Neverlang. The *developer* implements language components in the form of Neverlang modules. The FeatureIDE environment is synchronized with the AiDE bottom-up algorithm that performs the reverse engineering of a FM for the current language family given an AiDE environment, *i.e.*, a set of modules.

AiDE 2 provides an extension of the default FeatureIDE *Configuration Editor*: the *Neverlang Configuration Editor*. With this tool, the *deployer* can create, update and deploy language configurations, including the selection of both features and roles as well as a manual solution to the *ripple effect* through the renaming mechanism. Language variants are generated from configuration files by the AiDE 2 composer and deployed using Gradle. A Neverlang distribution piggybacks the IDE for the generated compiler. The *user* must register a variant on AiDE 2 by indexing the location of the language binaries in a configuration file. The environment is then capable of automatically detecting the new language variant and its IDE, that the *user* can adopt in the development process of programs written in the corresponding language. Compilers are Java classes[5] with a main method and can thus be directly executed within the environment, in both execution and debug mode.

In summary, Neverlang and AiDE can support all phases of the *language developer* activity, whereas AiDE, FeatureIDE and Gradle all the phases of the *language deployer* activity. Finally, Neverlang supports the usage of language variants.

3.3 Properties of a well designed language decomposition—Point 2

We now introduce the properties that a well designed language decomposition should have to tackle point 2 of Parnas' design methodology. The properties we introduce and the respective metrics (Section 3.4) are mainly based on the works of Briand Briand et al. (1998) and Coleman et al. (1994), which we adapt to the framework of Neverlang-based LPLs.

LPLE encompasses both domain engineering and application engineering aspects. The quality assessment of LPLs must therefore cope with both these aspects. We address the quality in the design of the variability space of a LPL and then of its language components in this order. Notice that language component is synonymous of module in Neverlang; henceforth the two terms will be used interchangeably.

Properties of a well designed variability space. The main concern of SPL and LPL engineering is improving the reusability of software artifacts. A key factor in determining the value from reuse opportunities in LPLs is scoping. If the scope is too large, the investment may be wasted on assets that will never be reused. If the scope is chosen too narrow, components may be designed in a way that does not support reuse across enough relevant products (Schmid 2002). Moreover, if the number of products in a product family grows too large, for the user it is impractical to find the correct product and to specify a valid configuration by keeping track of all the features during the configuration process (Pereira et al. 2016). Since the number of configurations is exponential in the number of features, LPLs should apply techniques to reduce the number of configurations to be monitored (Kim et al. 2010) or split complex LPLs into smaller LPLs towards a multi-LPL approach (Rosenmüller et al. 2011). Dealing with smaller LPLs also stems the problem of increased connectivity associated to programmers using information they should not possess about other modules (Parnas 1971). Therefore a well designed LPL should find the correct scope by establishing a trade-off in the number of products it provides. To summarize, we are interested in the following *qualitative properties* of the variability space:

- *self-descriptiveness*—property of a system or component containing enough information to explain its objectives and properties (ISO/IEC/IEEE International Standard 2017).
- *encapsulation*— concept that access to the names, meanings, and values of the responsibilities of a class is entirely separated from access to their realization (ISO/IEC/IEEE International Standard 2017).

[5] The Neverlang language workbench supports Java, Scala, Kotlin and JRuby semantic actions, but `language` units are always translated to Java source code.

Moreover, we are interested in the following *quantitative properties* of the design of the variability space:

- *adaptability*—degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments (ISO/IEC/IEEE International Standard 2017).
- *complexity*—degree to which a system's design or code is difficult to understand because of numerous components or relationships among components (ISO/IEC/IEEE International Standard 2017).
- *modifiability*—degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality (ISO/IEC/IEEE International Standard 2017).

In particular, a LPL is *adaptable* when its products can be used in several different contexts and is *modifiable* when its structure and implementation can be changed without affecting the quality of other features and existing products. A well designed variability space meets the proper trade-off among these properties by decreasing complexity and increasing adaptability while taming the decrease in modifiability. Notice that the list of properties we introduce represent the foundations of a design methodology for Neverlang LPLs and might be extended and completed in future works.

Properties of well designed language components A well designed modularization brings several benefits to the entire software system (Parnas 1972): the development can proceed in parallel with minimal communication, it is possible to change one module without affecting the others and the system can be studied one module at a time. As a result of a well designed modularization, the whole system should be better designed because it is better understood by the developers (Parnas 1972). This fits our view of a design methodology: in bottom-up LPLE, the FM is the result of running the generation algorithm on a set of language components. In our case AiDE 2 takes a set of Neverlang modules and generates the FM accordingly as described in Section 2. For this reason, the design of the variability space is tightly tied to the design of its modules. It is known that most development efforts and funds go towards the testing and maintenance of software products (McCabe 1976; Pressman 2005; de Vasconcelos et al. 2017; Fernández-Sáez et al. 2018) and that automated software maintainability analysis can be used to guide software-related decision making (Coleman et al. 1994). Thus, the identification of modules that are hard to test and maintain is a fundamental requirement in the development of any software system. In turn, this requirement led to the definition of design properties such as cohesion and coupling that are said to affect reusability, maintainability and fault-proneness (Briand et al. 1998). The correlation between those properties was supported by empirical evidence (Briand et al. 1994; Card et al. 1985; Card et al. 1986). Cohesion and coupling can be used to evaluate several aspects of modules design. Coupling is strongly related to the probability of fault detection (Briand et al. 1998). On the other hand lack of cohesion—despite not being directly associated to faults empirically—can hinder the design of the system. Parnas (1972) stated that a modularization is effective when there is no confusion in the intended interface with other system modules. To this goal each module should be a small manageable unit that can be easily understood and well programmed. In feature-oriented programming, low cohesion is an indicator of several concerns being merged into the same feature. While not directly causing faults in software products, lack of cohesion in features represents an opportunity for further decomposition. High coupling is an indicator of the same concern being split into several features. A refactoring process should compose coupled features into a single one. Increasing cohesion and reducing coupling in turn improves time of development, flexibility and comprehensibility (Parnas 1972). To summarize, we are interested in the following *quantitative properties* of the language components:

- *cohesion*—degree to which the tasks performed by a single software module are related to one another (ISO/IEC/IEEE International Standard 2017).
- *coupling*—degree of interdependence between software modules (ISO/IEC/IEEE International Standard 2017).
- *complexity*—degree to which a system or component has a design or implementation that is difficult to understand and verify (ISO/IEC/IEEE International Standard 2017).
- *maintainability*—degree with which a software system or component can be modified to change or add capabilities, correct faults or defects, improve performance or other attributes, or adapt to a changed environment (ISO/IEC/IEEE International Standard 2017).

In particular, a well designed language decomposition should provide modules with high cohesion and maintainability but with low coupling and complexity. Notice that the list of properties we introduce represent the foundations of a design methodology for Neverlang LPLs and might be extended and completed in future works.

3.4 Metrics for the detection of design errors in LPLs—Point 3

In this section, we propose a framework of quantifiable metrics that can be used to measure the properties presented in Section 3.3. These metrics are evaluated through static and/or dynamic quality assurance techniques that asses the level of quality of a language decomposition. Eventually this enables the language designers to determine if the requirements for a well structured LPL are met. In this regard, the metrics constitute a method for the detection of errors in design decisions and thus they fulfill point 3 of Parnas' vision of design methodologies. Both the LPL engineering process presented in Section 3.1 and the LPL engineering environment presented in Section 3.2 are designed to work in conjunction with the evaluation of these metrics. More precisely, in Section 3.1 we stated that the *language deployer* can report to the *language developer* to request a refactoring on a set of language features when they are too coarse grained or do not compose. Now, this step is streamlined by the introduction of our metrics: the LPL engineering environment provides tools for the evaluation of these metrics, thus enabling the detection of design errors without requiring manual inspection from the *language deployer*. When a LPL reaches the deployment phase, it should meet the quality standards imposed by the stakeholders. This refactoring process is repeated iteratively until the results match the requirements.

The framework we propose is based on metrics taken from the literature of object-oriented systems. Our contribution is the definition of the same metrics in the context of the Neverlang language workbench. This is done by mapping concepts of object-orientation to the corresponding concepts of Neverlang. In the following paragraph we will introduce the metrics for each of the quantitative properties we proposed in Section 3.3. Our focus is on the metrics for the evaluation of language components since, to the best of our knowledge there is no prior contribution in this regard. This proposal will be supported by an empirical study (Section 4) to determine the relation among these metrics and any ideal values.

Metrics for the evaluation of variability spaces. The FM is one of the most important artifacts in SPL engineering since errors in the FM can propagate to subsequent SPL phases (Bezerra et al. 2015). The FM is a valuable tool during the design phase and the configuration process thanks to the definition of optional and mandatory features, as well as of alternative sets. Therefore, all the metrics used to evaluate the variability space perform an analysis of the FM. Qualitative properties are not measured by metrics. Encapsulation is the result of the engineering process: the activity of *language developers* is separated from that of *language deployers*. The two roles do not share any artifacts thus the feature implementation and their representation are separated. Self-descriptiveness requires that a FM contains enough information to explain its domain while not containing information pertaining several domains. This property can be qualitatively assessed based on abstract features, that do not have any impact at the implementation level but enable reasoning on language variants (Thüm et al. 2011). AiDE 2 maps features with no concerns in common to separate FM sub-trees through abstract features. High arity near the root of the FM is undesirable because it is a sign of an LPL dealing with several different domains. High arity near the leaves indicates the presence of several variants for the same feature, all dealing with the same domain. Instead, there is a large variety of metrics can be perform the quality assessment of the FM in literature (El-Sharkawy et al. 2019). A thorough evaluation of all the metrics proposed in literature is out of the scope of our contribution. Instead, in this work we focus on a subset of the existing metrics to show how literature fits our vision of a LPL design methodology. For each property of the variability space from Section 3.3, the red boxes highlight the characteristics a metrics should have to be considered viable. The yellow boxes show how each metric fits the above property by respecting all characteristics. For instance, **Metric 2.3** is a valid metric for **Property 2** because it matches the *nonnegativity* and *nondecreasing monotonicity* requirements while having the same *null value*, *worst value* and *ideal value*. Notice that some metrics (such as number of configurations) may satisfy the definition of more than one property. If the ideal value does not match between the two properties, ensuring that the variability space is well designed will require finding an acceptable trade-off. For each metric, we also add a rationale explaining why the metric was chosen to measure the corresponding property.

Property 1: Adaptability

- *Nonnegativity.* The adaptability of an LPL is nonnegative because an LPL cannot be adapted to a negative number of different contexts.
- *Nondecreasing monotonicity.* The adaptability of the union of two disjoint LPLs is greater than or equal to the adaptability of each individual LPL. The union of two LPLs is at least as adaptable as the union of the respective contexts.
- *Null value*: 0. The adaptability of an empty LPL is 0 because an empty LPL can be adapted to 0 contexts.
- *Worst value*: 0. An LPL with 0 adaptability cannot be used in any context.
- *Best value*: ∞. The adaptability of an LPL can be increased arbitrarily. The higher the adaptability, the more contexts it can be used in.

Metric 1.1: Number of configurations

- *Rationale.* Each configuration corresponds to a different product, *i.e.*, a different context the LPL can be adapted to.

- *Nonnegativity.* An FM cannot contain a negative number of configurations.
- *Nondecreasing monotonicity.* Adding features and constraints of an existing FM to a disjoint FM does not decrease the number of valid configurations.
- *Null value*: 0. An empty FM has 0 valid configurations.
- *Worst value*: 0. An LPL can have 0 valid configuration if the language family does not contain any language.
- *Ideal value*: ∞. There is no upper bound (except for hardware and software limitations) to the number of configurations of a FM.

Property 2: Complexity

- *Nonnegativity.* The complexity of a LPL is nonnegative because it is impossible to understand a LPL with negative effort.
- *Nondecreasing monotonicity.* The complexity of the union of two LPLs is greater than or equal to the complexity of each individual LPL. The effort required to understand the union of two LPLs is at least as much as the effort of understanding the most complex of the two LPLs.
- *Null value*: 0. The complexity of an empty LPL is 0 because an empty LPL can be understood with no effort.
- *Worst value*: ∞. The complexity of an LPL can increase arbitrarily. The higher the complexity, the more difficult it is to understand.
- *Ideal value*: 0. The lower the complexity, the easier the LPL is to understand. An LPL with 0 complexity requires no effort to be understood.

Metric 2.1: Number of configurations

- *Rationale.* Each configuration corresponds to a different product. Each product is a different element that must be understood.

- *Nonnegativity.* An FM cannot contain a negative number of configurations.
- *Nondecreasing monotonicity.* Adding features and constraints of an existing FM to a disjoint FM does not decrease the number of valid configurations.
- *Null value*: 0. An empty FM has 0 valid configurations.
- *Worst value*: ∞. Understanding an infinite amount of configurations requires infinite effort.
- *Ideal value*: 0. Understanding 0 configurations requires no effort.

Metric 2.2: Number of features

- *Rationale.* Each feature is a different component that must be understood.

- *Nonnegativity.* An FM cannot contain a negative number of features.
- *Nondecreasing monotonicity.* The union of two FMs has at least the same number of features as the most complex of the two FMs.
- *Null value*: 0. An empty FM has 0 features.
- *Worst value*: ∞. Understanding an infinite amount of features requires infinite effort.
- *Ideal value*: 0. Understanding 0 features requires no effort.

Metric 2.3: Number of constraints

- *Rationale.* Each constraint is a different relationship among components that must be understood.

- *Nonnegativity.* An FM cannot contain a negative number of constraints.
- *Nondecreasing monotonicity.* The union of two FMs has at least the same number of constraints as the most complex of the two FMs.
- *Null value*: 0. An empty FM has 0 constraints.
- *Worst value*: ∞. Understanding an infinite amount of constraints requires infinite effort.
- *Ideal value*: 0. Understanding 0 constraints requires no effort.

Metric 2.4: Number of atomic sets

- *Rationale.* Each atomic set represents a relationship among components that must be understood. According to Mann and Rock [71] an atomic set represents a group of features that can be treated as a single unit during the analysis. Atomic sets can therefore be refactored into a unique feature to reduce complexity..

- *Nonnegativity.* An FM cannot contain a negative number of atomic sets.
- *Nondecreasing monotonicity.* The union of two FMs has at least the same number of atomic sets as the most complex of the two FMs.
- *Null value*: 0. An empty FM has 0 atomic sets.
- *Worst value*: ∞. Understanding an infinite amount of atomic sets requires infinite effort.
- *Ideal value*: 0. Understanding 0 atomic sets requires no effort.

Modifiability can be hard to assess and thus we measure its opposite property (lack of modifiability). If lack of modifiability is low then modifiability is high and vice versa.

Property 3: Lack of modifiability

- *Nonnegativity.* An LPL cannot be modified with negative efficiency.
- *Non monotonicity.* Modifying an LPL can both improve or reduce the efficiency of further modifications.
- *No null value.* The efficiency of modifying an empty LPL cannot be assessed.
- *Worst value*: 1 or 100%. An LPL with lack of modifiability equal to 1 cannot be modified efficiently because any modification introduces defects or degrades existing product quality.
- *Ideal value*: 0. Modifying an LPL with 0 lack of modifiability requires no effort.

Metric 3.1: Relative number of features appearing in constraints

- *Rationale.* The relative number of features appearing in constraints highlights the degree of relation between features and constraints in a FM. When the relative number of features appearing in constraints is high, modifying or deleting a feature can affect the existing constraints and vice versa. Dependency preservation algorithms and slicing techniques must take this into account when features are deleted [60].

- *Nonnegativity.* A constraint cannot contain a negative number of features.
- *Non monotonicity.* Modifying features, constraints and joining FMs can both increase and decrease the relative number of features appearing in constraints.
- *No null value.* An empty FM has 0 features over 0 constraints. Therefore the number of features appearing in constraints cannot be computed.
- *Worst value*: 1. If the relative number of features appearing in constraints is 1 each modification to a feature affects at least another feature.
- *Ideal value*: 0. If the number of features appearing in constraints is 0 then it means that there are no constraints and modifying a feature do not affect any other feature.

Metrics for the evaluation of language components. Cohesion and coupling were originally defined for the evaluation of object-oriented classes. Yet, they were successfully applied to other fields, such as procedural software (Henry and Selig 1990). In this work we attempt a similar approach by mapping concepts from object orientation to the corresponding Neverlang concepts to assess the cohesion and coupling of language modules. This mapping requires three concepts to be redefined: i) classes are mapped to Neverlang modules, ii) methods are mapped to semantic actions and iii) class fields are mapped to grammar attributes. Table 1 contains the formal definition and description of the used metrics based on these definitions along with the original object-oriented counterpart that inspired them. Following the original framework, we measure the opposite of cohesion—*i.e.*, lack of cohesion in modules. Lack of cohesion occurs when a module contains several semantic actions that do not refer any attributes in common. Coupling is caused by semantic actions from different modules referencing the same attributes. Notice that cohesion and coupling are conflicting factors in the design of language modules, which is in line with previous definitions of these metrics (Perepletchikov et al. 2007). Now we contextualize the metrics from Table 1 with regards to the definitions of cohesion and coupling. The approach is the same we used to justify the metrics used on the FM.

Property 1: Lack of cohesion

- *Nonnegativity.* The lack of cohesion of a module is nonnegative because semantic actions cannot be negatively related to one another.
- *Nonincreasing monotonicity.* The lack of cohesion of a module does not increase if relationships are added between semantic actions.
- *Null value*: 0. The lack of cohesion of a module which no semantic actions is 0.
- *Worst value*: ∞. The lack of cohesion of a module can increase arbitrarily. The higher the lack of cohesion, the less semantic actions in that module belong to the same module. A module with ∞ lack of cohesion only contains semantic actions that have no relationships with one another.
- *Ideal value*: 0. The lower the lack of cohesion, the more semantic actions in a module belong to the same module. If lack of cohesion is 0 then all semantic actions are related.

Table 1 Cohesion and coupling metrics for language product lines

Metric	Description	OO equivalent
LCOA1 (lack of cohesion in actions)	The number of pairs of actions in the module that do no reference any attributes in common on the same nonterminal symbols.	LCOM1 (Chidamber and Kemerer 1991) (lack of cohesion in methods)
LCOA2	The number of pairs of actions in the module that do no reference any attributes in common on the same nonterminal symbols minus the number of pairs of actions that do. If this difference is negative, LCOA2 is set to zero.	LCOM2 (Chidamber and Kemerer 1994)
LCOA3	Let G be an undirected graph where the vertices are the actions of a module and there is an edge between two vertices if the corresponding actions reference at least one attribute in common on the same nonterminal symbols. LCOA3 is then defined as the number of connected components of G.	LCOM3 (Hitz and Montazeri 1995)
Co (connectivity)	Let $\lvert V\rvert$ be the number of vertices in the graph G from LCOA3, and $\lvert E\rvert$ the number of edges. Then Co $= 2\dfrac{\lvert E\rvert - (\lvert V\rvert - 1)}{(\lvert V\rvert - 1)\,(\lvert V\rvert - 2)}$.	Co or C (Briand et al. 1998; Hitz and Montazeri 1995) (connectivity)
LCOA5	Let $S = \{s_1, \ldots, s_n\}$ be the set of actions of a module which reference the attributes $A = \{a_1, \ldots, a_m\}$. Let $M_j = \left\{s \in S \mid s \text{ references } a_j\right\}$ and $\mu_j = \lvert M_j\rvert$ then LCOA5 $= \dfrac{\frac{1}{m}\left(\sum\limits_{j=1}^{m} \mu_j\right) - n}{1 - n}$	LCOM5 (Briand et al. 1998)
Coh (cohesion)	Given n, m and μ_j as in LCOA5, Coh $= \dfrac{\sum\limits_{j=1}^{m} \mu_j}{nm}$ is a normalized representation of individual references to attributes in actions.	Coh (Briand et al. 1998) (cohesion)
CBM (coupling between modules)	A module is coupled with another if actions in eithser module reference attributes which are also referenced by the other module on the same nonterminal symbols. CBM for a module is then defined as the number of other modules to which it is coupled.	CBO (Chidamber and Kemerer 1994) (coupling between objects)

Metric 1.1: LCOA1

• *Rationale.* If a semantic action does not use any attributes that are also used by another semantic action then the execution of either does not affect the other. The two actions are unrelated because they operate on different parts of the grammar or implement a different semantic concern/role.

• *Nonnegativity.* A module contains a non negative number of pairs of actions.
• *Nonincreasing monotonicity.* Adding a reference to an attribute in a semantic actions does not increase the number of pairs of semantic actions that do not refer any attribute in common.
• *Null value*: 0. A module with no semantic actions contains 0 pairs of semantic actions.
• *Worst value*: ∞. A module with LCOA1 $= \infty$ contains an infinite number of pairs of actions, an infinite number of which refer no attributes in common.
• *Ideal value*: 0. In a module with LCOA1 $= 0$, taken any pair of actions from that module, the two semantic actions refer at least one attribute in common. A module with one or less semantic action also has LCOA1 $= 0$.

Metric 1.2: LCOA2

• *Rationale.* LCOA2 is similar to LCOA1. However, while pairs of semantic actions that do not refer any attribute in common reduce cohesion, pairs that do increase cohesion.

• *Nonnegativity.* LCOA2 is nonnegative by definition (see Table 1).
• *Nonincreasing monotonicity.* Adding a reference to an attribute in a semantic actions does not increase the number of pairs of semantic actions that do not refer any attribute in common and can increase the number of pairs of semantic actions that refer attributes in common.
• *Null value*: 0. A module with no semantic actions contains 0 pairs of semantic actions.
• *Worst value*: ∞. A module with LCOA2 $= \infty$ contains an infinite number of pairs of actions, an infinite number of which refer no attributes in common. Moreover, the number of pairs of semantic actions that refer attributes in common is lower than the number of pairs that do not.
• *Ideal value*: 0. In a module with LCOA2 $= 0$, for each pair of actions that do not refer any attribute in common, there is at least another pair that do. A module with one or less semantic action also has LCOA2 $= 0$.

Metric 1.3: LCOA3

• *Rationale.* Each connected component of graph G (see Table 1) is a group of actions that are related—*i.e.* they refer the same attributes. Two different connected components in G represent semantic actions that are unrelated to one another: each connected component should be refactored in a separate module.

• *Nonnegativity.* G contains a non negative number of connected components.
• *Nonincreasing monotonicity.* Adding a reference to an attribute in a semantic actions does not decrease the edges of G and therefore cannot decrease the number of connected components.
• *Null value*: 0. Graph G for a module with no semantic actions contains 0 vertices and therefore 0 connected components.
• *Worst value*: ∞. A module with LCOA3 $= \infty$ is described by a graph G with an infinite number of connected components.
• *Ideal value*: 0. A module with no semantic actions has LCOA3 $= 0$ and. In this instance the ideal value is a corner case because a module with LCOA3 $= 0$ has no semantic capability. The ideal value for a module that contains at least one semantic action is LCOA3 $= 1$, because all the semantic actions belong to the same connected component.

Metric 1.3b: Co

• *Rationale.* Co adds information to LCOA3 and can be measured only when LCOA3 $=$ 1. Lack of cohesion is minimum when G only has one connected component. However, the topology of G can affect cohesion: a clique is more cohesive than a chain. Co measures this relationship within a connected component. Co is maximum in a clique and minimum in a chain. Therefore, the higher Co, the better.

Metric 1.4: LCOA5
• *Rationale.* LCOA5 measures the number of distinct attributes referred by semantic actions in a module. LCOA5 differs from the other metrics because it is normalized; the worst value is LCOA5 $= 1$, that represents a module in which each action refers a different attribute. Instead, LCOA5 is 0 if all semantic actions refer all attributes which means the module is very cohesive..

• *Nonnegativity.* LCOA5 < 0 if $\sum_{j=1}^{m} \mu_j > nm$ which is impossible because $\forall j, \mu_j \leq n$. In particular, $\mu_j = n$ if attribute a_j is referred by all semantic actions in S.
• *Nonincreasing monotonicity.* Adding a reference to an attribute in a semantic actions increases $\sum_{j=1}^{m} \mu_j$ and decreases LCOA5 as a result.
• *Null value*: 0. We do not measure LCOA5 for modules with no semantic actions. However, we can say that in a module with no semantic actions (and no referred attributes) all semantic actions refer all attributes.
• *Worst value*: 1. In a module with LCOA5 $= 1$, each attribute is referred by only one semantic action.
• *Ideal value*: 0. If LCOA5 $= 1$, $\sum_{j=1}^{m} \mu_j = nm$, thus all attributes are referred by all semantic actions.

Metric 1.4b: Coh
• *Rationale.* Coh is just a variation on LCOA5. Coh measures the fraction of attributes that are referred by all semantic actions. Therefore the two metrics are similar and we will not discuss Coh in detail. However Coh measures cohesion instead of lack of cohesion. The higher Coh, the better.

Property 2: Coupling
• *Nonnegativity.* The coupling of a module is nonnegative because modules cannot have a negative degree of interdependence to one another.
• *Nondecreasing monotonicity.* The coupling between modules does not decrease if relationships are added between modules.
• *Null value*: 0. The lack of coupling of a module with no semantic actions is 0.
• *Worst value*: ∞. The coupling of a module can increase arbitrarily. The higher the coupling, the more semantic actions in that module depend from semantic actions in another module. A module with ∞ coupling depends from an infinite number of other modules.
• *Ideal value*: 0. The lower the coupling, the less semantic actions in a module depend from semantic actions in other modules. If coupling is 0 then all semantic actions are independent from other modules.

Metric 2.1: CBM
• *Rationale.* If a semantic action uses any attributes that are also used by another semantic action in a different module then the execution of either affects the other. The two actions (and the corresponding modules) are coupled because they operate on the same part of the attribute grammar and implement an overlapping semantic concern/role.

• *Nonnegativity.* A module cannot refer the same attributes as a negative number of other modules.
• *Nondecreasing monotonicity.* Adding a reference to an attribute in a semantic actions does not decrease the number of semantic actions in different modules that refer attributes in common.
• *Null value*: 0. A module with no semantic actions contains 0 semantic actions and can be paired with no other modules.
• *Worst value*: ∞. A module with CBM $= \infty$ contains references to attributes that are also referred in an infinite number of other semantic actions in different modules.
• *Ideal value*: 0. A module with CBM $= 0$ has no degree of interdependence with other modules their semantic action do not refer any attribute in common.

With regards to complexity and maintainability, we do not introduce any new metric. For this reason we do not provide a rationale for all the metrics as we did above and instead define the concept of operator and operand in Neverlang and apply metrics from literature. We apply several metrics to measure the complexity of Neverlang modules: lines of code (LoC), McCabe's cyclomatic complexity (CC) (McCabe 1976), Halstead's complexity metrics (Halstead 1977)—volume (V), difficulty (D), effort (E), development time (T) and delivered bugs (B). Moreover we measure the maintainability of modules through the Coleman's maintainability index (MI) (Coleman 1992) and the normalized derivative used in Visual Studio (VS) (Chen et al. 2017). We define the cyclomatic complexity of a Neverlang module as the sum of the cyclomatic complexities of the semantic actions in that module—thus a module with no actions will have a cyclomatic complexity of 0. Halstead's complexity measures are calculated upon the vocabulary—*i.e.*, the number of operands and operators. Neverlang operators are **`module`**, **`imports`**, **`reference syntax`**, **`:`**, ←,**`categories`**, **`in-buckets`**, **`out-buckets`**, **`role`**, `[]`. Identifiers as module names, labels, offsets, terminal and nonterminal symbols, attribute and role names are instead operands. We define the set of operators in a module—both total (N_1) and distinct (η_1)—as the union of Neverlang operators and Java operators in each semantic action. The same applies to operands—total (N_2) and distinct (η_2). Given this distinction, Halstead's complexity metrics and the maintainability index are then used with their original meaning and computed with the conventional formulas. Please refer to the reported works for a full overview of the complexity and maintainability metrics.

4 Evaluation

We setup an experiment assessing the quality in the design of LPLs with respect to the metrics presented in Section 3.4. The experiment tries to answer RQ_1 by applying the proposed metrics against a wide range of LPLs and RQ_2 by comparing the effects of different design strategies on the experimental results. On the basis of the collected data, we will also try to define some best practices that should be applied when designing a language decomposition with the goal of improving the maintainability of LPLs and their reuse.

4.1 Experimental setup

Hardware setup. All experiments were run on an 64 bits Arch Linux machine with an Intel Core i7-1065G7 3.9GHz processor and a 16 GB RAM. The hardware setup affects the measurement of the valid configurations lower bound estimation.

Software setup. Metrics were extracted from both Neverlang source code implementations and compiled binaries using the Neverlang 2.1.2 runtime in combination with AiDE 2.0.1 and FeatureIDE 3.6.1. Henceforth, whenever we mention AiDE 2 we are referring to the toolchain comprising Neverlang, AiDE and FeatureIDE, *i.e.*, the integrated LPL development environment introduced in Favalli et al. (2020) for brevity. AiDE 2 exploits the latest version of the Neverlang language variability support (Vacchi et al. 2013) through an extension of the original AiDE FM generation and management algorithms (Cazzola and Olivares 2016).

Data setup. The subject for this empirical evaluation is a collection of Neverlang LPLs, including Neverlang itself. Each LPL is composed of a collection of Neverlang source files implementing the language features and a FM in `XML` format compliant with FeatureIDE. The considered LPLs can be logically classified into three groups: i) *legacy* LPLs created before the introduction of the design methodology, ii) *sub-languages* LPLs created applying the design methodology, and iii) *refactored* LPLs—*i.e.*, LPLs redesigned to maximize the reuse of language assets from other LPLs. This classification provides a first broad subdivision of the LPLs based on the differences in their development process: comparing the results of *legacy* LPLs with those of *sub-languages* we can evaluate how the design methodology affects the results. All the considered LPLs are shown in Table 2 along with some of the metrics and general project information. For each LPL, Table 2 also reports the work in which it was originally introduced, if any. The codebase contains a wide variety of different LPL projects. Below, a brief description of each of the considered LPLs.

- Neverlang is a legacy LPL. implements the translator from Neverlang source to Java. Neverlang is an LPL applying the bootstrapping technique (Cazzola and Vacchi 2013)
- LogLang is a legacy LPL. It implements a family of languages for scripting the tasks of log maintenance in compliance with the `logrotate` linux tool
- Javascript is a legacy LPL. This LPL is a family of Javascript-based language interpreters compliant to the ECMAScript 3 specification

Table 2 Feature model information for Neverlang LPLs considered in this experiment

Category	Project	From	LoC	(actions)	Modules	Features	(semantic)	Constraints	Configs
legacy	Neverlang	Vacchi and Cazzola (2015)	1650	(349)	40	81	(41)	18	231935 †
	LogLang	Cazzola and Poletti (2010)	284	(38)	15	28	(18)	19	104532
	Javascript	Cazzola and Olivares (2016)	4199	(1399)	108	262	(121)	162	172169 †
	State Machines	Vacchi et al. (2013)	948	(247)	24	64	(37)	36	271356 †
	Tyl$_{\text{legacy}}$	–	4981	(2335)	78	190	(123)	186	185459 †
	Java	Kühn and Cazzola (2016)	5488	(1233)	113	307	(169)	180	155522 †
	Java Role Extension	Kühn and Cazzola (2016)	202	(36)	5	19	(10)	0	3156
	Object Teams	Kühn and Cazzola (2016)	1036	(222)	16	55	(46)	10	288678 †
	PowerJava	Kühn and Cazzola (2016)	300	(77)	7	26	(14)	3	49193
	Rava	Kühn and Cazzola (2016)	309	(76)	5	20	(10)	1	2866
	Java Relations	Kühn and Cazzola (2016)	1026	(284)	17	61	(34)	11	284600 †
	Rumer	Kühn and Cazzola (2016)	1630	(466)	30	101	(60)	20	222048 †
sub-languages	Types	–	686	(107)	44	44	(26)	0	274388 †
	Expressions	–	2471	(911)	84	113	(54)	1	206616 †
	Variables	–	493	(135)	18	44	(25)	12	283325 †
	Errors	–	0	(0)	0	2	(1)	0	2
	Compilation Unit	–	24	(4)	2	3	(1)	0	3
	Arrays	–	391	(113)	12	32	(19)	3	328545 †
	Statements	–	149	(20)	6	16	(8)	2	770
	Control Flow	–	385	(50)	12	29	(16)	0	333033 †
	Functions	–	321	(74)	7	27	(16)	7	83457
refactored	Desk	Vacchi and Cazzola (2015)	63	(8)	3	8	(4)	2	15
	Lambda	–	107	(20)	4	10	(6)	0	150
	Tyl$_{\text{refactored}}$	–	241	(64)	4	15	(9)	1	522
	JS + Slicing	–	5	(0)	1	2	(0)	0	2
	Java + SM	–	5	(0)	1	2	(0)	0	2
overall	–	– 27398	(8268)	656	1561	(868)	674	–	

Values marked with a † represent a lower bound in the number of valid configurations. Note that "(actions)" represents the absolute frequency of LoC in semantic actions out of the total LoC and "(semantic)" represents the absolute frequency of semantic features out of the total number of features

- State Machines is a legacy LPL. The State Machines LPL defines a DSL for the description of state machines that are then translated into Java code
- $\text{Tyl}_{\text{legacy}}$ is a legacy LPL. Products of the $\text{Tyl}_{\text{legacy}}$ LPL are DSLs for enterprise resource planning (ERP) that translate the source code to different, more refined semantics by feeding it to different language variants—with the same syntax and different semantics—in succession: a Java main class first calls the *import* language variant—which accepts a list of Tyl source files and builds the symbol table for all the declared Tyl modules—and then the *translation* language variant that uses the information collected by the *import* language variant to type check the program and finally transpile to Java. $\text{Tyl}_{\text{legacy}}$ includes a QueryDSL that could be refactored out and distributed as a standalone LPL extension. Notice that a *refactored* version of Tyl is also present
- Java is a legacy LPL. The Neverlang implementation of Java is a Java-to-Java source code translator
- Java Role Extension, Object Teams, PowerJava, Rava, Java Relations, and Rumer are *legacy* LPLs. Each of these LPLs implements a different language extension based on Java that embrace the role-based programming paradigm (Kühn et al. 2014) to distinguish between classes and role types.
- Types is a sub-languages LPL. Types contains the definition of all Java primitive types (including numeric separators) with heavy emphasis on modularization
- Expressions is a sub-languages LPL. This LPL defines all the most used operators in infix, prefix and postfix version with customizable operator priority
- Variables is a sub-languages LPL. This *sub-language* contains a portable definition of identifiers, of a symbol table, as well as the concept of block and scope
- Errors is a sub-languages LPL. Errors in this LPL leverage Neverlang endemic slices to build an error report at compilation phase. This LPL is a corner case of our evaluation because it is totally composed of endemic slices. This will be our running example to show how the metrics we introduced are currently not suited to evaluate endemic slices
- Compilation Unit is a sub-languages LPL. This LPL provides an entry point for the parser of any language and the semantics for the generation of a Java class using the syntax-directed translation technique (Aho et al. 1986) regardless of the underlying syntax used for the compilation unit
- Arrays is a sub-languages LPL. The Arrays LPL implements arrays with a Python-like syntax, as well as the slicing operator
- Statements is a sub-languages LPL. The Statements *sub-language* contains the glue code to hook other *sub-languages* to statements and blocks of imperative programming languages
- Control Flow is a sub-languages LPL. While, do-while, for loops, switches and if statements are part of the Control Flow *sub-language*
- Functions is a sub-languages LPL. This LPL defines a function table as well as the syntax and the semantics for the declaration and usage of functions
- Desk is a refactored LPL. This implementation of the Desk DSL performs heavy reuse of the Types, Expressions and Variables sub-languages
- Lambda is a refactored LPL. Lambda applies a multi-phase strategy (similar to $\text{Tyl}_{\text{legacy}}$) to resolve any lambda expression by running a second interpreter that performs the evaluation of the expressions. This interpreter is run only on a sub-tree of the abstract syntax tree (AST)
- $\text{Tyl}_{\text{refactored}}$ is a refactored LPL. This version of Tyl reimplements some of the variants of $\text{Tyl}_{\text{legacy}}$ while maximizing reuse of assets from *sub-languages* LPLs

```
public class Bar {
  void foo(int arg) {
    StateMachine sm = state machine Door {
      state Opened
      state Closed
      transition from Opened to Closed: Close
      transition from Closed to Opened: Open
    };
  }
}
```

Listing 2 Accepted Java+SM syntax

- JS+Slicing is a refactored LPL. JS+Slicing is defined as a LPL which depends on Javascript and Arrays and combines them to allow the use of the array slicing operator in Javascript
- Java+SM is a *refactored* LPL. Java+SM combines Java and State Machines so that state machine definitions are accepted as valid Java expressions (as in Listing 2).

The codebase above contains a wide variety of different projects. Notice that *legacy* LPLs are obtained by the decomposition of a well-defined language. Instead, *sub-languages* LPLs are not decompositions of any programming language per se, but rather describe the variability of a family of sub-languages. As introduced in (Cazzola et al. 2018), every sub-language contains a subset of language features from a so-called host language to support a well-defined programming aspect of that language. Therefore, a *sub-languages* LPL describes the variability of a family of sub-languages: the features of a *sub-languages* LPL are those supporting the same programming aspect across several host languages. Each product of a *sub-languages* LPL is a sub-language and cannot be used alone; instead they rely on the presence of other language features provided by other sub-languages (Cazzola et al. 2018). Most LPLs are implemented either as families of interpreters or as families of translators with Java back-end; most of the *sub-languages* LPLs implement both interpreters and translators. The considered LPLs substantially differ in scope, including minimal projects with just a small set of available language components as well as language families with hundreds of language features and possibly millions of variants. The dataset and the used scripts for running the experiment are available at Zenodo.[6]

Process. Neverlang was used to access source code information (lines of code, number of modules, semantic actions and roles), FeatureIDE was used for any information regarding the LPLs variability space (number of features, number of constraints, and number of configurations), whereas AiDE 2 was used to compute cohesion, coupling, complexity and maintainability metrics. All test results were stored in CSV format for further elaboration. The data collection was automated by using a custom AiDE 2 wrapper without bringing any changes to the Neverlang framework. As already discussed, *legacy* and *sub-languages* projects apply completely different design strategies in their language decompositions. Most notably *sub-languages* were developed using AiDE 2 and applying the design methodology introduced in Section 3.4 whereas *legacy* projects were developed before the introduction of AiDE 2 and of the design methodology. *Legacy* projects can therefore be used as a control group to compare the evaluation results between projects that apply the design methodology

[6] https://doi.org/10.5281/zenodo.5236547

against those that do not and to detect whether the methodology brings any improvement in the quality of LPLs. The results of this comparison will be discussed in Section 4.5.

4.2 Results

General experiment statistics. We evaluated 26 LPLs: 12 *legacy*, 9 *sub-languages*, and 5 *refactored*. Among the several millions of valid configurations, we explicitly defined 53 languages: for each language, we performed the configuration process to deploy a language variant; each of the 53 languages was tested to ensure its syntactic and semantic validity. Table 2 summarizes the basic information of each LPL and the overall results. The codebase amounts to a total of 656 modules and 27398 lines of Neverlang code—8268 of which represent code in semantic actions and the remaining 19130 represent syntactic definitions and other Neverlang constructs (mainly declaration, **`imports`** and **`roles`**). The Neverlang modules implement 2447 semantic actions—3.73 actions per module on average. Each LPL project is described by a FM generated by AiDE 2 for a total 1561 language features and 674 constraints; 868 of the total language features are semantic features—2.82 semantic actions per language feature on average.

As seen in Section 2, Neverlang semantic actions are implemented in Java by default, hence they can instantiate and use external Java classes; similarly endemic slices declare instances of Java objects which will be globally accessible by any semantic action. These classes are not considered in the experiment since they are used as black-boxes and their cohesion, coupling, complexity and maintainability are comparable to those of external libraries in object-oriented systems. These concepts are meaningful only on Neverlang modules where all the syntax and semantics of a language are implemented. Slices, bundles, and languages are the main constructs for feature composition and their evaluation only affect the amount of glue code needed in language definitions.

Feature model metrics. FeatureIDE limits to one hour the computation of any metrics on the FM. Formally, the upper bound in valid configurations for a FM with n features, depth 2, and no cross-tree constraints is 2^n; due to the properties of FMs introduced in Section 2, the upper bound lowers when the depth of the FM or the number of constraints increases. This means that the number of valid configurations increases exponentially with the number of features (Batory et al. 2000) and that computing the exact number of valid configurations is just not feasible for large projects. Table 2 highlights whether the reported number of configurations is an exact value or a lower bound. The results show that FeatureIDE can compute the exact number of valid configurations on FMs with 28 features at most. Using better hardware and more efficient methods for counting the number of valid configurations may reveal a closer approximation. However, we are not interested in an exact result in this work. Finding the language variant that meets the user's requirements in a LPL with several hundreds of thousands of valid configuration is hard regardless of how close the approximation is Pereira et al. (2016). It should also be considered that a high number of configurations affects the viability of solutions to NP-hard problems such as slicing (Krieter et al. 2016). Therefore, spending several hours and computational effort during the daily development process of LPLs to determine an exact number of configurations (or a better approximation) may be not worth it depending on the application. In our use case the number of features suffices as an indicator of the growing size of the LPL: a high number of features hints at the possibility of splitting the LPL in a multi-dimensional variability modeling approach (Rosenmüller et al. 2011). Still, there might be other use cases in which an exact number of configurations is required. If that is the case, different product line verification approaches based on #SAT (Sundermann et al. 2021) and Binary Decision Diagrams (BDDs) (Cordy et al. 2013) should be considered. On a side note, renames—*i.e.*, the Neverlang way of stopping the ripple effect—are not expressed in the FM and cannot be considered when computing valid configurations. An invalid configuration from the FM perspective could still generate a valid language variant if the correct renames are defined. As a result, the variability space of the language family is further widened by renames. The effects of renames on the size of the variability space will be part of a future work.

We evaluated the presence of atomic sets in each LPL: being either all active or all inactive in a given configuration, they behave as a single feature and thus represent excellent refactoring points. Language components in atomic sets should be either merged or refactored to eliminate the dependency. Atomic sets were present in only three legacy projects thus we list them explicitly instead of showing them in Table 2 for brevity reasons:

- Javascript has 2 atomic sets of 2 features each (both associated to assignment expressions);
- Java has 1 atomic set of 13 features (including all the mathematical expressions);
- Tyl$_{legacy}$ has 1 atomic set of 3 features (including variable declarations and assignments) and 1 atomic set of 7 features (including all the mathematical expressions).

AiDE FMs contain an average of 0.21 constraints per feature. However, only 30 constraints come from *sub-languages* and *refactored* LPLs whereas *legacy* LPLs contain 0.36 constraints per feature. The Pearson correlation coefficient (Pearson 1895) (PCC) between number of features and number of constraints is 0.94 for legacy LPLs and 0.09 for *sub-languages* LPLs. This relation is reported in Fig. 3. The result highlights a linear increase in the number of constraints with respect to the size in *legacy* projects. We can conclude that LPLs developed without applying our design methodology result in FMs with more constraints and a high relative number of features appearing in constraints: the association between these metrics and quality aspects such as low modifiability and high complexity is discussed in this work (see Section 3.4) and evaluated in literature (El-Sharkawy et al. 2019). For the same reasons, the results presented in this paragraph are not used to answer

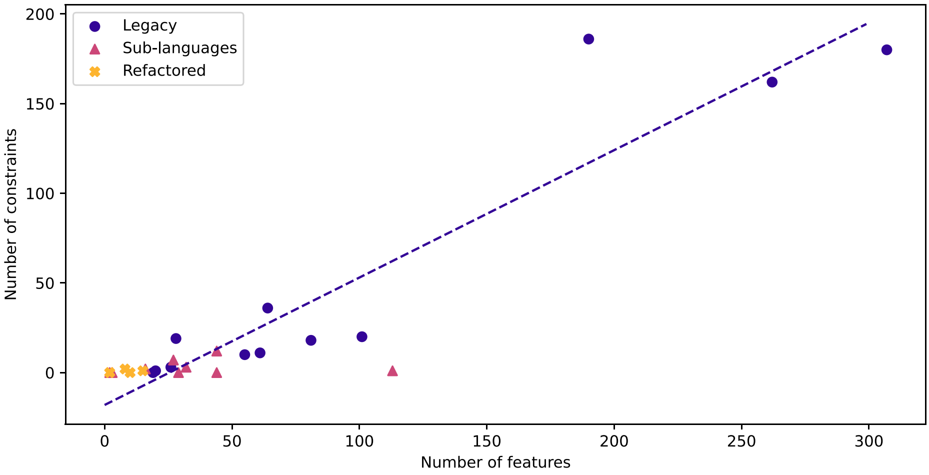


Fig. 3 Number of constraints in Neverlang LPLs with respect to their number of features

RQ$_1$ and RQ$_2$. Instead, the remainder of this section will focus on the properties of language components introduced in Section 3.3 and their evaluation.

Cohesion and coupling. We assessed each of the LPLs with respect to each of the metrics introduced in Table 1. First we can observe that some of these metrics are not applicable to all modules. Co applies only to modules with an LCOA3 value of 1—*i.e.*, completely interconnected actions dependency graph—and with at least three semantic actions. Co was applied to only 68 (10.57%) modules since Neverlang modules tend to be very small. LCOA5 was applied to modules that refer at least one grammar attribute and defining two semantic actions (438 modules or 66.77% of the total). Coh was applied to modules referencing at least one grammar attribute and defining one semantic action (560 modules or 85.37% of the total). Similarly, results are not available for the Errors LPL because it does not define any Neverlang module but just endemic slices for the definition and collection of compilation errors. We mentioned above that our metrics do not apply to the evaluation of endemic slices; the results report this fact for completeness. Table 3 summarizes the results for each LPL whereas Table 4 compares *legacy* and *sub-languages* LPLs. *Refactored*

Table 3 Cohesion and coupling on Neverlang modules for each LPL sorted by increasing LCOA1

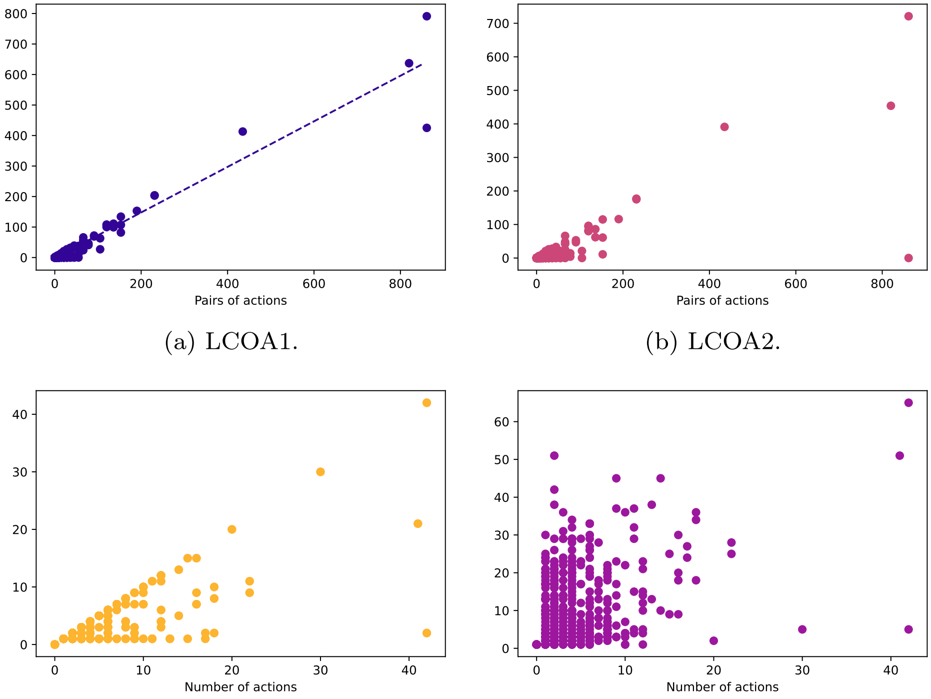
Project	LCOA1	LCOA2	LCOA3	Co	LCOA5	Coh	CBM
JS + Slicing	0.00	0.00	0.00	–	–	–	1.00
Java + SM	0.00	0.00	0.00	–	–	–	1.00
Compilation Unit	0.00	0.00	0.50	–	–	–	1.00
LogLang	0.20	0.20	1.07	–	1.00	0.95	11.53
Types	0.27	0.27	1.00	1.00	0.54	0.87	2.73
Desk	0.33	0.33	1.00	–	1.00	0.75	1.67
Expressions	0.35	0.11	2.63	1.00	0.28	0.90	3.12
Variables	0.39	0.39	1.11	–	0.77	0.80	2.33
Statements	0.50	0.50	1.17	–	0.50	0.87	1.00
Lambda	0.50	0.25	1.00	–	0.47	0.73	2.00
Javascript	1.40	0.70	2.41	–	0.23	0.89	13.94
Arrays	1.42	1.17	1.67	–	0.82	0.61	2.17
Control Flow	1.67	1.50	2.08	–	0.57	0.76	1.83
Functions	2.29	2.14	2.29	–	0.93	0.51	2.71
State Machines	3.29	0.92	2.25	1.00	0.65	0.69	4.92
PowerJava	4.57	2.86	3.57	–	0.76	0.51	4.71
Rava	4.80	3.40	5.00	–	0.58	0.53	3.80
Neverlang	5.05	2.62	3.02	–	0.68	0.66	7.35
Tyl$_{\text{refactored}}$	9.00	6.00	2.50	–	0.65	0.66	2.50
Tyl$_{\text{legacy}}$	9.24	5.42	1.45	0.76	0.28	0.80	8.97
Java Role Extension	11.20	7.80	5.20	–	0.71	0.49	1.00
Rumer	13.57	9.00	6.00	–	0.67	0.48	6.47
Java	23.41	13.88	2.50	0.92	0.54	0.64	17.90
Java Relations	25.65	17.06	7.12	–	0.64	0.50	6.65
Object Teams	90.19	81.81	9.12	–	0.61	0.55	4.38
Errors	–	–	–	–	–	–	–

Table 4 Comparison between *sub-languages* and *legacy* LPLs

Metric	Overall			Legacy			Sub-languages		
	Mean	Median	σ	Mean	Median	σ	Mean	Median	σ
Features	60.04	28.50	77.21	101.17	62.50	93.99	34.44	29.00	31.32
Constraints	35.92	2.50	54.98	53.83	18.50	71.34	2.78	1.00	3.91
LCOA1	9.67	0.00	49.39	13.53	0.50	58.67	0.56	0.00	1.45
LCOA2	6.47	0.00	39.40	9.04	0.00	46.90	0.42	0.00	1.12
LCOA3	2.65	2.00	3.29	2.99	2.00	3.62	1.91	1.00	2.16
Co	0.85	1.00	0.29	0.84	1.00	0.29	1.00	1.00	0.00
LCOA5	0.48	0.60	0.37	0.48	0.60	0.35	0.48	0.50	0.45
Coh	0.74	0.70	0.27	0.71	0.62	0.27	0.83	1.00	0.24
CBM	8.80	5.00	9.46	11.46	9.00	10.14	2.70	2.00	2.13

LPLs do not yield significant results because—being specifically designed with the goal of maximizing reuse—they mostly focus on glue code and contain a minimal set of features. Table 4 shows how *sub-languages* consistently perform better on all metrics and highlights how applying a design methodology can improve the quality of language modules.

Figure 4a, b, and c show the increase in lack of cohesion when the number of semantic actions (or pairs of semantic actions) in a module increases. In particular, we applied the



(a) LCOA1.

(b) LCOA2.

(c) LCOA3.

(d) CBM.

Fig. 4 Cohesion (**a-c**) and coupling (**d**) with respect to the number of actions per module

Table 5 Summary of the result of complexity metrics on Neverlang LPLs sorted by decreasing MI

Project	CC	LoC	V	D	E	T	B	MI	VS
JS+Slicing	0.00	5.00	5.00	4.00	172.08	9.56	0.01	125.37	73.31
Java+SM	0.00	5.00	5.00	4.00	172.08	9.56	0.01	125.37	73.31
Compilation Unit	1.00	12.00	12.00	7.00	863.91	47.99	0.03	104.45	61.08
Types	0.98	15.59	15.59	4.66	1090.56	60.59	0.03	98.59	57.66
LogLang	1.80	18.93	18.93	6.53	1925.15	106.95	0.05	93.64	54.76
Desk	1.00	21.00	21.00	8.33	2782.56	154.59	0.06	90.92	53.17
Statements	1.67	24.83	24.83	8.17	3334.29	185.24	0.07	87.96	51.44
Lambda	1.75	26.75	26.75	9.75	7093.58	394.09	0.11	84.10	49.18
Variables	2.44	27.39	27.39	8.61	8589.59	477.20	0.11	83.61	48.89
Control Flow	2.50	32.08	32.08	6.92	2954.60	164.14	0.06	83.01	48.54
Expressions	3.27	29.42	29.42	14.27	24029.29	1334.96	0.21	79.94	46.75
Neverlang	3.85	41.35	41.35	9.72	12058.23	669.90	0.15	74.35	43.48
Arrays	3.00	32.58	32.58	8.92	8136.50	452.03	0.12	79.77	46.65
State Machines	3.96	39.50	39.50	11.33	10387.41	577.08	0.13	76.64	44.82
Javascript	4.69	38.88	38.88	14.41	21270.55	1181.70	0.20	75.05	43.89
Java Role Extension	5.20	40.40	40.40	17.60	20477.80	1137.66	0.24	73.75	43.13
Functions	3.29	45.86	45.86	11.14	11165.46	620.30	0.15	72.57	42.44
PowerJava	4.57	42.86	42.86	14.00	21421.32	1190.07	0.23	72.09	42.16
Java	7.36	48.57	48.57	15.06	35645.70	1980.32	0.26	69.22	40.48
Rumer	7.80	54.33	54.33	21.57	49047.38	2724.85	0.37	66.17	38.70
$\text{Tyl}_{\text{refactored}}$	4.00	60.25	60.25	10.25	21061.02	1170.06	0.22	65.48	38.29
Rava	5.40	61.80	61.80	20.80	48978.01	2721.00	0.41	63.50	37.13
Java Relations	7.65	60.35	60.35	21.94	57099.55	3172.20	0.43	63.50	37.13
$\text{Tyl}_{\text{legacy}}$	8.45	63.86	63.86	17.73	50014.63	2778.59	0.36	62.74	36.69
Object Teams	9.38	64.75	64.75	21.00	54806.41	3044.80	0.42	61.72	36.09
Errors	–	–	–	–	–	–	–	–	–

PCC between LCOA1 and the number of pairs of actions[7] in a module and observed a strong linear correlation of 0.97 highlighting the fact that the lack of cohesion scales quadratically in the number of semantic actions. Conversely, there is no apparent relation between the number of actions in a module and CBM (Fig. 4d). We can conclude that reducing the size of a module in terms of its semantic actions can increase cohesion; increasing the size of a module does not reduce coupling instead.

Code complexity and maintainability. Table 5 contains the results of the evaluation of complexity and maintainability metrics on each LPL. McCabe's CC validity is often discussed due to its theoretical weakness (Ebert et al. 2016); the initial proposed limit of 7 ± 2 CC has been relaxed over time and the belief is that CC is no more useful than a LoC metric. In fact, Table 5 shows that average CC tends to be higher for projects in which average

[7] The number of pairs of actions in a module with n actions is $\binom{n}{2} = \frac{n(n-1)}{2}$.

LoC is also high. Nonetheless CC is widely used for fault prediction in industrial production and can be applied to the evaluation of other metrics such as MI. All the considered LPL projects scored an average CC below 10, with the highest being Object Teams at 9.38 and the lowest Types at 0.98. The average CC is 3.80. Both JS+Slicing and Java+SM have an average CC of 0.00 because they do not implement any semantic action and instead just perform syntax checking on source code.

Halstead's complexity measure source code properties by comparing them to physical matter properties such as volume; volume is also used for the computation of MI and VS. For each Halstead metric the lower the result, the better. More abstract measures such as volume, difficulty and effort are translated into concrete estimations: required time to program and number of delivered bugs.

MI collects the data from CC, LoC and Halstead metrics to estimate the maintainability of a software system. According to Coleman (1992) a MI value above 85 (or the corresponding VS=49.71) indicates that the software is highly maintainable, a value between 85 and 65 (or the corresponding VS=38.01) suggests moderate maintainability, and a value below 65 indicates that the system is difficult to maintain. Table 5 reports the MI results: LPLs in green are highly maintainable, LPLs in yellow are moderately maintainable and LPLs in red are difficult to maintain. Once again, *sub-languages* and *refactored* LPLs apply the design methodology and show average to high maintainability. All the LPLs that are difficult to maintain are part of the *legacy* project, on which the design methodology was not applied.

4.3 Principal component analysis

To answer RQ_1 we performed a principal component analysis (PCA) on our results. Thanks to the PCA we can extract the dimensions that have the most relevance on the results to obtain the properties of Neverlang language decompositions. Each dimension is represented by one of the metrics we evaluated on language components. Figure 5 depicts the results of the PCA. To perform the PCA we discard any dimensions containing null values, leaving 13 dimensions. We normalize the results and list the principal components. Then, we discard the least relevant components according to the Kaiser rule (Kaiser 1960): only the principal components with an eigenvalue above 1 are kept. The rationale is that any principal component with an eigenvalue below 1 is less relevant than the original dimensions. For each principal component we determine the original dimensions that have the most impact by analyzing the covariance matrix. This analysis reveals three principal components that describe 88.9% of the variance in the dataset.

- *PC1* (67.4% of the variance) is mainly determined by V, B, LoC, CC, E, T in order of relevance. Other dimensions have lower impact. All the most relevant dimensions described by *PC1* are metrics used to evaluate *complexity*.
- *PC2* (13.2% of the variance) is mainly determined by LCOA2, LCOA1 and LCOA3 in order of relevance. Other dimensions have much lower impact. All the most relevant dimensions described by *PC2* are metrics used to evaluate *cohesion*.
- *PC3* (8.3% of the variance) is mainly determined by VS and MI in order of relevance. Other dimensions have much lower impact. All the most relevant dimensions described by *PC2* are metrics used to evaluate *maintainability*.

Notice that only three of the four properties of language decompositions that we introduced in Section 3.3 are matched by a principal component. In fact, CBM is not among the most relevant dimensions in any of the principal components. Instead, the variance of CBM is

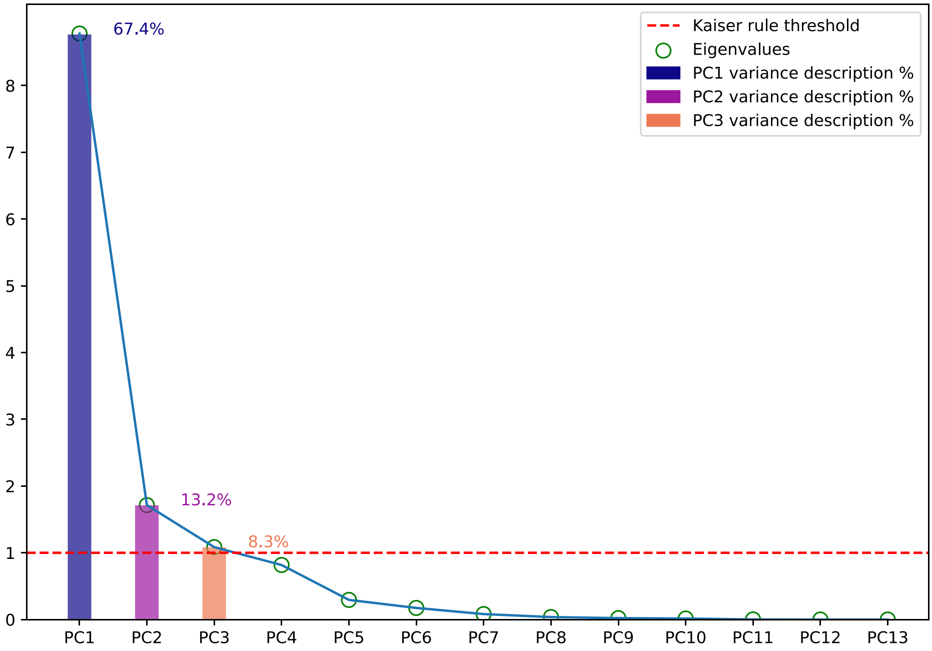


Fig. 5 Results of the PCA performed on the subject systems and the considered metrics. Each element on the x-axis is one of the principal components. On the y-axis, their respective eigenvalue

described by other dimensions. We can conclude that we are interested in three properties of a language decomposition in Neverlang: *complexity*, *cohesion* and *maintainability*; *coupling* has a limited impact on the variance of the results and is described by the other properties. This result is relevant because CBM is the only metric among the considered ones that can only be evaluated on a set of language components. In other words, we cannot evaluate CBM of a single language component but only the CBM of a language component within a LPL. The evaluation of standalone language components is enabled by the fact that the coupling property does not have a big impact on the variance of the results.

4.4 Thresholds

To answer RQ_2 we must determine a replicable method for the detection of design errors in Neverlang LPLs. We use the metrics for the evaluation of language components: a low score in any of these metrics will suggest a refactoring opportunity or the need for a review of the design choices. For complexity and maintainability metrics we stick to the quality thresholds defined in literature that we reported in Section 4.2. However, cohesion and coupling metrics were first defined in Section 3.4 and therefore there is no prior work that investigates any ideal value. For this reason, we perform a quartile analysis on our dataset to determine the thresholds between well designed components and components with average design and between components with average design and poorly designed components. The results are reported in Fig. 6. We consider any modules with a score in the interquartile range (IQR) to have average design. Values below the first quartile (Q1) indicate good design and values above the third quartile (Q3) indicate bad design. The only exception is Coh, for

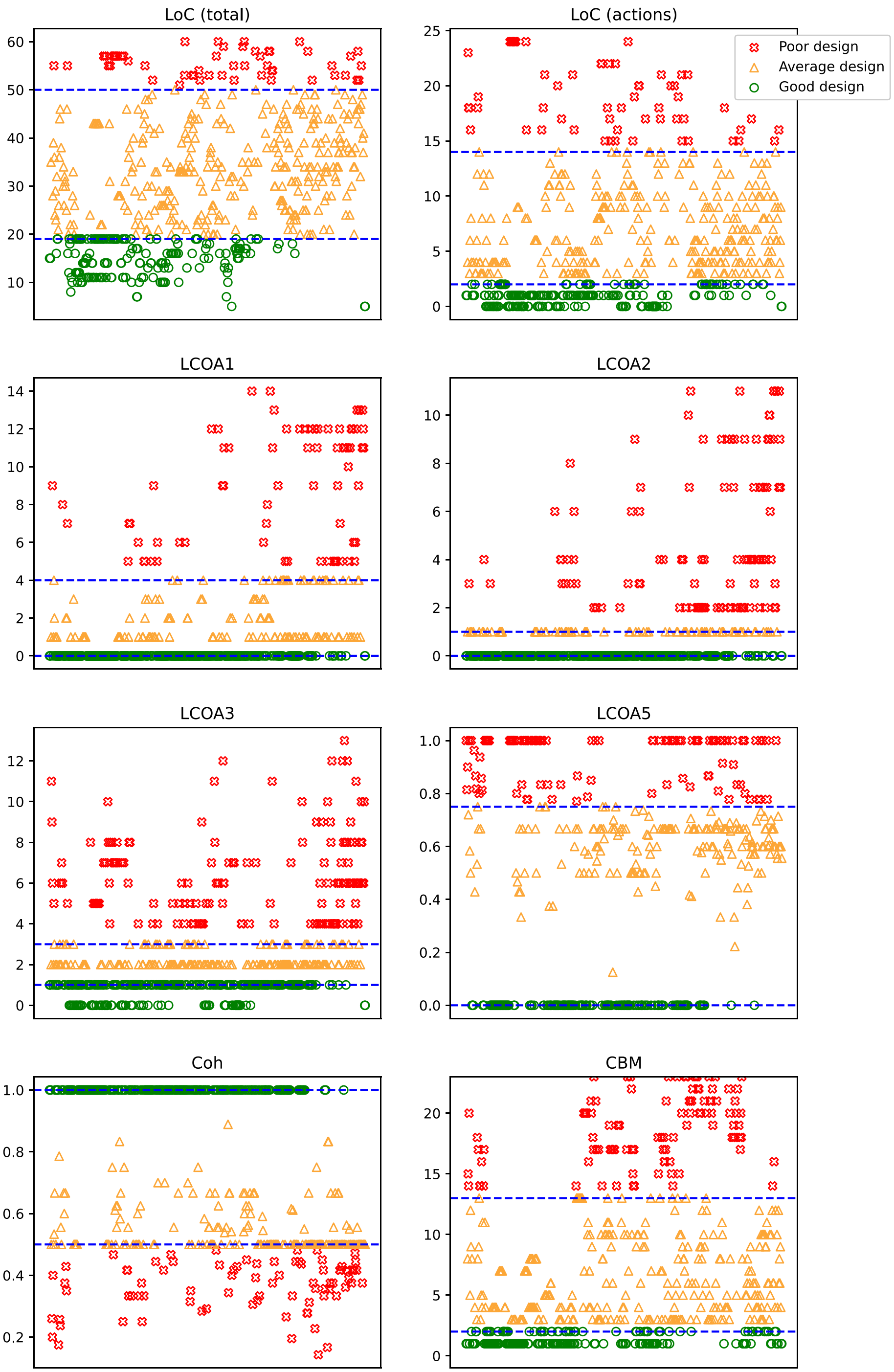


Fig. 6 Evaluation results of several metrics and their quartiles. Some results are omitted for better readability. Modules in the IQR is considered to have average design. Modules below Q1 are well designed and modules above Q3 are badly designed or vice versa depending on the metric

which values above Q3 indicate good design and values below Q1 indicate bad design. This analysis reveals the following thresholds.

- **LoC (total)**
  good design: LoC ≤ 19
  average design: $19 <$ LoC ≤ 50
  bad design: LoC > 50

- **LoC (actions)**
  good design: LoC ≤ 2
  average design: $2 <$ LoC ≤ 14
  bad design: LoC > 14

- **LCOA1**
  good design: LCOA1 ≤ 0
  average design: $0 <$ LCOA1 ≤ 4
  bad design: LCOA1 > 4

- **LCOA2**
  good design: LCOA2 ≤ 0
  average design: $0 <$ LCOA2 ≤ 1
  bad design: LCOA2 > 1

- **LCOA3**
  good design: LCOA3 ≤ 1
  average design: $1 <$ LCOA3 ≤ 3
  bad design: LCOA3 > 3

- **LCOA5**
  good design: LCOA5 ≤ 0
  average design: $0 <$ LCOA5 ≤ 0.75
  bad design: LCOA5 > 0.75

- **Coh**
  good design: Coh ≥ 1
  average design: $0.5 \leq$ Coh < 1
  bad design: Coh < 0.5

- **CBM**
  good design: CBM ≤ 2
  average design: $2 <$ CBM ≤ 13
  bad design: CBM > 13

Notice how it is relatively easy to keep lack of cohesion to a minimum in most modules. For this reason, any result below the optimal value is considered average design on lack of cohesion metrics.

4.5 Discussion

The experimental results outline the amount of data that can be inferred from a LPL with relative ease. Now we summarize these results with respect to our research questions to outline our contribution towards a design methodology for LPLs.

RQ$_1$. What are the properties of a language decomposition in Neverlang?

We answered this research question by performing a PCA on our dataset. The design of Neverlang language components is determined by three different properties, each represented by a principal component:

— *cohesion*—manner and degree to which the tasks performed by a single software module are related to one another.
— *complexity*—degree to which a system or component has a design or implementation that is difficult to understand and verify.
— *maintainability*—ease with which a software system or component can be modified to change or add capabilities, correct faults or defects, improve performance or other attributes, or adapt to a changed environment.

$\mathbf{RQ_2}$. How can errors in design decisions be detected in Neverlang LPLs?

Errors in the definition of the variability space can be detected by evaluating a series of metrics on the FM. These metrics are widely used and discussed in literature [37]. AiDE 2 FMs use the same formalism as the other FMs, thus the same metrics can be used in their evaluation. Please refer to the literature for an overview on how FM metrics can be used to detect errors in design decision. With regards to language components metrics, we performed a quartile analysis to measure the quality of 26 Neverlang-based LPLs. We determined the results a well designed language component should score. A design error is detected whenever any of the following results is obtained:

CC > 9
MI < 65
VS < 38.01
LoC (total) > 50
LoC (actions) > 14
LCOA1 > 4
LCOA2 > 1
LCOA3 > 3
LCOA5 > 0.75
Coh < 0.5
CBM > 13

Since each metric is associated to a different property of LPLs, the metric that highlighted the design error also determines which property should be improved.

Moreover, our evaluation showed that the Neverlang LPLs on which we applied our design methodology performed better on average on all metrics.

4.6 Lessons learned

Now we provide an overview of the lessons learned from this evaluation and how the design properties translate into design practices on Neverlang LPLs. Finally we show how our contribution can be adapted to other language workbenches.

Scope of a language family. We found that the best trade-off in the number of configurations that can be computed with AiDE 2 is met at 28 features: Table 2 shows that it was possible to provide an exact number of configurations for LogLang—which contains 28 features—and for all the other LPLs with less than 28 features but we only got a lower bound for Control Flow—which contains 29 features and for all the other LPLs with more than 29 features. Of course this result is not objective and might change based on several factors: machine performance, time limit provided by the tool, development requirements, resolution algorithm and number of constraints. However, as a general rule a manageable language decomposition should contain between 25 and 30 features to enable exhaustive approaches and a small degree of FM analysis in AiDE 2.

Taming the complexity of LPLs. Constraints are a useful tool for guiding the configuration process in LPLs but they should be wisely and sparingly used to avoid limiting the language family's richness. In fact our evaluation shows that Javascript, $\text{Tyl}_{\text{legacy}}$ and Java are both the LPLs with the highest number of constraints (162, 186 and 180 respectively) and the only LPLs containing any atomic sets, that we associated to lack of modifiability in Section 3.4. In general, the design process should keep cross-tree constraints to a minimum. Take LogLang as an example. It is an average size LPL with 28 features. At the same time, it shows a relatively high number of constraints: 19 constraints, with 78.50% of its features appearing at least once in a constraint which is the highest out of all the considered LPLs. The high relative number of features appearing in constraints is indeed a design flaw since it affects the configuration process: when selecting a feature from the FM there is a high chance of causing the *ripple effect*. This result is in fact empirically coupled with high CBM: Table 3 shows that LogLang has an average CBM of 11.53, which is the third highest value. This result hints that the design of the LogLang language decomposition could be improved to reduce coupling, constraints and features appearing in constraints. This can be achieved with a simple design practice. Take Listing 1 as an example: AiDE 2 defines a constraint between the `Backup` feature and any feature providing the `String` nonterminal because the `Backup` syntax directly depends on the `String` nonterminal. Given the FM from Fig. 1 this translates into the constraint `SYNTAX_Backup` $\Longrightarrow$ `SYNTAX_LogLangTypes`. Each LogLang configuration containing the `SYNTAX_Backup` feature must also contain the `SYNTAX_LogLangTypes` feature. Let us assume we extend the DSL by adding the expressions—*i.e.*, the DSL must perform backup operations between files whose path is not hardcoded in a string but is the result of an expression instead. The approach would be to apply a rename to the LogLang language: `String`→ `Expression`. However this rename changes all the `String` nonterminals in all productions of the language to Expression nonterminals which may not be desirable and may cause unexpected behavior or even syntax clashes. Instead, we propose the refactoring in Listing 3. It uses dummy nonterminals `BackupSource` and `BackupTarget` and delegates the hooking of dummies with concrete nonterminals to the **`language`** unit and, in particular, to the **`rename`** construct. This results in the SYNTAX_Backup $\Longrightarrow$ SYNTAX_LogLangTypes constraint not being defined by AiDE 2, reduces coupling accordingly and enables more variability in the LPL products. For instance, `BackupTarget` is the result of an expression in Listing 3 whereas `BackupSource` behaves the same in both Listings 1 and 3. A manual inspection of the LogLang FM shows that 7 out of the 19 constraints are of this kind thus applying this design practice whenever possible would reduce the number of constraints to 12. We applied this design practice to all LPLs created using our design methodology. As a result, the relative number of features in constraints is 32.98% on average in *legacy* LPLs and only 12.28% in *sub-languages* LPLs. The same applies to CBM: average CBM is 11.46 in *legacy* LPLs and 2.70 in *sub-languages*. It is arguable that the refactoring from Listing 3 is more verbose and reduces the readability of the system as a whole, however we introduced in Section 3.4 that a well designed language decomposition can be studied one module at a time. Dummy nonterminals can be declared without knowledge of other modules since `BackupSource` and `BackupTarget` are not intended to be used by any other module. The original implementation requires knowledge of at least two modules instead: the module requiring the `String` nonterminal (`Backup`) and the module providing it (`LogLangTypes`). The overhead of introducing dummy nonterminals is paid by the *language deployer* during the configuration process since a viable language configuration requires all the dummy nonterminals to be renamed to concrete nonterminals. This problem, however, can be mitigated by using the AiDE 2 language configuration editor that keeps track of all open nonterminals and helps the *language deployer* figuring out if any additional renames are needed (Favalli et al. 2020).

Increasing the number of semantic actions in a module negatively impacts their lack of cohesion: Fig. 4a shows that LCOA2 and LCOA1 in particular have a good fit for a quadratic curve with respect to the number of semantic actions in a module; the number of semantic actions is also an upper bound for LCOA3 by construction. Object Teams has the highest lack of cohesion—90.13, 81.81 and 9.12 for LCOA1, LCOA2 and LCOA3 respectively—associated to a high count of semantic actions per module—150 actions in 16 modules. Once again, applying the design methodology can improve the results: *sub-languages* LPLs contain 2.02 actions per module—41.56 actions in 20.56 modules—and present lower lack of cohesion—0.56, 0.42 and 1.91 for LCOA1, LCOA2 and LCOA3 on average respectively as shown in Table 4. On the other hand, Fig. 4d shows that increasing the number of semantic actions does not benefit CBM either, despite the intuition that big modules should increase the likelihood of dependent actions being in the same module. These results incentivize the development of small Neverlang modules with a few semantic actions.

Low complexity and high maintainability are not always associated to low lack of cohesion and coupling in modules. The most interesting case is LogLang: Table 3 shows high CBM while Table 5 highlights high MI. This is achieved by the usage of endemic slices, as shown in Listing 1: referencing endemic instances in Neverlang modules not only generates a variability point in which the behavior of a semantic action can be changed by swapping endemic slices in a configuration but also delegates the complexity of the algorithm to an external Java class, rendering the semantic action easier to maintain as a result. Implementing different roles and therefore different semantic actions in separate modules can also help decreasing average complexity of the system since the CC of a module is the sum of the CC of its semantic actions. A good language decomposition should then take advantage of endemic slices while minimizing the number of referred attributes in semantic actions.

We expect all of the above design practices to improve the variability of the language family and to ease the configuration process of language variants. Scaling to larger language families should not be done by adding more features to the same FM, but applying a multi-dimensional variability modeling approach (Rosenmüller et al. 2011) in which each dimension describes the variability of a family of sub-languages instead. For instance, we propose to improve the approach we propose in Favalli et al. (2020)—in which the configuration process focuses on one LPL and the products of the LPL are **`language`**

```
module BackupSyntax {
  reference syntax {
    provides { Backup: backup, statement; Cmd: statement; }
    requires { BackupSource; BackupTarget; }
    Backup ← "backup" BackupSource  BackupTarget;
    Cmd    ← Backup;
  }
}

language LogLang {
  slices BackupSlice RemoveSlice
         RenameSlice MergeSlice Task
         Main LogLangTypes bundle(Expressions)
  endemic slices FileOpEndemic PermEndemic
  roles syntax < terminal-evaluation < permissions : execution
  rename {
      BackupSource → String;      BackupTarget → Expression;
  }
}
```

Listing 3 Refactoring Listing 1 to reduce CBM and the constraints introduced by `AiDE`

units—by using the configuration editor to deploy **`bundle`** units from different LPLs and then combining them into an interpreter or a compiler.

AiDE 2 can provide much information about the quality of language decomposition without any change to the Neverlang compiler. All the data needed for this experiment can be statically evaluated by accessing the Neverlang source code. Lack of cohesion, complexity and maintainability metrics only need information about a single module hence the overhead is negligible. Conversely, measuring CBM requires source-level information from all the concrete features in a LPL and becomes more time-consuming as the number of features increases. We limited the overhead thanks to AiDE 2 which already stores a reference to each module into an environment object to build the FM. Moreover, the PCA performed in Section 4.3 shows that CBM has low impact on the variance of the results: the evaluation of CBM can be avoided altogether if the time requirements become prohibitive.

A shared design methodology. In our work, we focused only on Neverlang and the metrics defined in Section 3.4 are based on Neverlang concepts. However, other language workbenches could apply the same approach we propose with minor changes. For instance, JastAdd aspects contain keywords for inherited (`inh`) and synthesized (`syn`) attributes which can be used to evaluate coupling and cohesion metrics. Similarly, in Melange the semantics are defined with Kermeta[8] aspects: if an aspect refers class attributes that were defined in a different aspect then the two aspects are coupled. As long as the language workbench provides tools to extract this information, our approach should be applicable with minimal effort. However, we suggest performing a preliminary empirical study on a case by case basis before applying our design methodology in production environments. For instance, the metrics might need to be adapted to suit the specific quirks of the language workbench. A major challenge in this regard is the level of granularity: Parnas' work on design methodologies emphasizes the concept of modularization (Parnas 1972). In Neverlang the translation is natural because modules are a core concept in the development of Neverlang LPLs, but each language workbench has a different approach to modularity. In MPS (Völter and Pech 2012) the user manipulates the AST directly and each AST node is an instance of a *Concept*. In Melange the aforementioned Kermeta aspects are woven with Ecore[9] meta-models into languages that can then be extended. Spoofax (Wachsmuth et al. 2014) is a collection of meta-languages, each dealing with a different aspect of language development and each with a different approach to modularization. Given this premise, it may be hard to define a design methodology shared among all language workbenches since some of the concepts may not translate well from one another. Instead, the research should focus on the definition of a shared baseline that is then instantiated differently for each language workbench.

4.7 Threats to validity

Construct validity—the degree to which the independent variables and dependent variables accurately measure the concepts they purport to measure (Wohlin et al. 2003)**.** Part of the metrics we propose are adaptations from object-oriented metrics. It is debatable that the Neverlang modularity model fits that of object-oriented programming,

[8] http://diverse-project.github.io/k3/
[9] https://wiki.eclipse.org.Ecore

i.e., that the proposed metrics actually measure what they purport to measure. Their definition was kept as close as possible to the original metrics to limit the discrepancy. Intuitively Neverlang modules fit the parallel with classes from object-orientation: classes are modules, methods are semantic actions, and attributes are nonterminal attributes from the attribute grammar. The evaluation shows reasonable results and the experiment was designed to include both projects attempting to optimize those metrics and a control group of *legacy* projects to which we did not apply any change before performing the evaluation. Most of the metrics lack normalization and require comparison to assess anything about the quality of software while others are not always applicable. In this work, we sticked as close as possible to the parallel between language feature and class: any inapplicability was addressed in the evaluation and should non impact the results. Finally, our framework focuses on modules and does not address other components of the Neverlang development process, such as slices, endemic slices, and Java source code. As previously stated, this should not influence the results since slices are just glue code with no intrinsic dependency whereas endemic slices and Java source are considered as black-box libraries on which we cannot improve. CC theoretical validity is discussed (Ebert et al. 2016) but CC is used in industrial production nonetheless and its value is needed to compute MI which was applied in the past to the evaluation of SPLs (Aldekoa et al. (2006, 2008). We could measure only a lower bound in the number of configurations thus one may question the validity of feature-oriented metrics—whose results also highly depend on the application domain. However little can be done to improve this since the number of configurations is known to scale exponentially with the number of features. Instead we can leverage this limitation to suggest a refactoring opportunity for LPLs by using a multi-LPL approach.

Internal validity—the degree to which conclusions can be drawn about the causal effect of the treatments on the outcomes (Wohlin et al. 2003)**.** Using a single framework for both the development and the evaluation of software may indeed cause internal validity issues. As already stated, this problem is mitigated by the presence of a control group and from the high variety of different LPLs we present. Due to the focus on maximizing reusability, some of the LPLs are extremely small and could be classified as libraries rather than LPLs but all the reported results are always weighted with respect to the number of modules in the project, hence outliers should not excessively impact the results. It should be noted that most projects were created by the same group of developers and that *legacy* projects were implemented in previous versions of Neverlang hence some changes were needed to adapt those projects to the current standard and to generate a FM using AiDE 2. This could cause some bias in the results but we always applied the minimum required changes without affecting neither syntactic definitions nor semantic actions.

External validity—the degree to which the results of the research can be generalized to the population under study and other research setting (Wohlin et al. 2003)**.** In this study, we only used LPLs created with the Neverlang language workbench and the AiDE LPL framework. We focused on concepts which are specific to Neverlang, such as, modules and semantic actions. Hence the same concepts may not be applicable to other language workbenches. However we tried to stick to elements of the attribute grammar formalism, which should be applicable to several other language workbenches such as JastAdd (Hedin and Magnusson 2003), for the definition of cohesion and coupling metrics. Instead, feature variability aspects of our evaluation are mostly shared among the product line engineering. If the language family is described by a FM then the same metrics can be applied with no changes. The time limit imposed by FeatureIDE on the evaluation of the metrics may cause different results to be obtained with respect to the number of configurations in subsequent experiments or in different research settings. However, the improvement that a different research setting could bring are limited due to the exponential nature of the quantity we are trying to measure. The same result may change considerably also if renames were to be considered in the computations of valid configurations in a future work, but renames can only increase the number of valid configurations since no rename can turn a valid configuration invalid. The number of configurations is a lower bound for most considered LPLs, thus it would still hold true if renames were added. The thresholds we used to answer RQ_2 are based on a limited set of LPLs, which design quality is not determined independently from the metrics by a domain expert. Therefore the results may not be generalizable to other LPLs. To address this threat to validity we took Neverlang LPLs created by different authors across several years without a shared vision or approach to language design. This should ensure our sample is fairly representative of the real world population.

5 Related Work

Language workbenches and the development of DSLs are established research topics. Most recent language workbenches (Erdweg et al. 2015)—such as, Spoofax (Wachsmuth et al. 2014), MPS (Völter and Pech 2012), MontiCore (Krahn et al. 2010) and Melange (Degueule et al. 2015)—provide *tools for system designers* by addressing the problem of IDE support for modular DSLs. Monticore, Spoofax and MPS directly or indirectly provide LPL engineering capabilities. Monticore supports language embedding and language inheritance for compositional development of language families. Butting et al. (2018) presented an approach to manage syntactic variability of extensible LPLs using Monticore. Spoofax supports generation of a wide variety of IDE tools for Eclipse and IntelliJ. Liebig et al. (2013) used Spoofax alongside FeatureHouse for the representation and composition of language features. MPS offers full IDE support and customizable abstract syntax tree manipulation. MPS was used to develop mbeddr (Völter et al. 2012), a set of integrated and extensible languages based on C for embedded software engineering with an IDE support. Mendez-Acuña et al. presented several contributions to the topic of reuse in language workbenches: in Jézéquel et al. (2014) the authors address the problems of programming languages evolution and maintenance, as well as their verification and validation; in Méndez-Acuña et al. (2016) they introduce PUZZLE as a tool for the detection of duplicates in syntactic and semantic definitions in Melange. Melange is also supported by GEMOC Studio (Combemale et al. 2017), used by language designers to build and compose DSLs and by domain designers to coordinate their models. In most of these works LPLs are not directly addressed but emerge from the development of language features, language variants and the tools they provide for system designers. For instance, we could not find any work directly addressing tool-supported LPL capabilities in JastAdd.

Thüm et al. (2011) indirectly address point 4 of the design methodology—*i.e.*, a *specification technique* for SPL products—by proof decomposition into features: all the features in a configuration are associated with a partial proof in Coq, then the proof assistant checks if the composed proof is valid thus verifying that the program variant is valid. Their work is complementary to ours since they address an aspect of design methodologies that we did not address in this contribution. On the same topic, CBS (Mosses 2019) provides an extensible library of reusable language specification components based on fundamental programming constructs (funcons).

The topic of *methods of detecting errors in design decisions* is one as old as software engineering itself and has been addressed in different ways in the framework of SPLs. IncLing (Al-Hajjaji et al. 2016) and MoSo-PoLiTe (Oster et al. 2011) address the problem of exponential increase in product configurations and apply pairwise testing to find a minimal subset of configurations covering 100% pairwise interactions. Perrouin et al. (2012) apply automated generation of test products using the t-wise SPL test adequacy criteria. Several works (Bagheri and Gasevic 2011; Her et al. 2007; van der Hock et al. 2003; Zhang et al. 2008) evaluate SPLs using structural or service utilization metrics and Aldekoa et al. (2006, 2008) used the maintainability index to evaluate the maintainability of SPLs.

However, to the best of our knowledge, no previous work performs a similar evaluation on LPLs nor defines a design methodology for LPLs. Despite being tailored to bottom-up LPLs and attribute grammars, our approach introduces the definition of a design methodology for LPLs, comprehensive of *the order in which decisions are made*, *what constitutes good structure for a system*, *methods of detecting errors in design decisions* and *tools for system designers*.

6 Conclusion

In this article, we presented the introductory study of a design methodology for language decompositions. Our original work (Favalli et al. 2020) focused on the order in which decisions are made and on tools for system designers. With this contribution we substantially extend the original paper by defining what constitutes good structure for a Neverlang LPL and methods of detecting errors in design decisions. We validated our research by answering the research questions RQ_1 and RQ_2 through an empirical study. The results show that AiDE 2 can be leveraged to compute several metrics adapted from the literature and supports the early detection of design flaws in language decompositions and their components. The results match our expectations: fine-grained language decompositions using abstractions in their syntactic definitions show better cohesion, coupling, complexity and maintainability results. Limiting the number of features in a LPL improves the accuracy of FM analysis tools and eases the configuration process of language variants. Contrary to our expectations, the PCA revealed that coupling is not a principal component in determining the variance of the results and that CBM can be expressed in terms of other quantities.

Nonetheless, to improve the usability of our LPL development environment the implementation must be further optimized for better performance and extended to include more Neverlang features. The metrics results that are now dumped to disk in CSV format should be fully integrated with the LPL engineering environment. Finally, our future work will focus on the evaluation of other Neverlang constructs, such as slices and endemic slices and on point 4 of Parnas' design methodology: *specification techniques* for Neverlang LPLs. Our work will focus on the integration of theorem proving and contracts support for Neverlang semantics, as well as their usage to determine the validity of a language specification.

Acknowledgments This work was partly supported by the MUR project “T-LADIES” (PRIN 2020TL3X8X).

Declarations

Conflict of Interests The authors declare that they have no conflict of interest.

References

Aho AV, Sethi R, Ullman JD (1986) Compilers: Principles, techniques, and tools. Addison wesley, Reading

Al-Hajjaji M, Krieter S, Thüm T, Lochau M, Saake G (2016) Incling: Efficient Product-Line Testing Using Incremental Pairwise Sampling. In: Schaefer I (ed) Proceedings of the 15th International Conference on Generative Programming and Component Engineering (GPCE'16). ACM, Amsterdam, pp 144–155

Aldekoa G, Trujillo S, Mendieta GS, Díaz O (2006) Experience Measuring Maintainability in Software Product Lines. In: Riquelme Santos JC, Botella P (eds) Proceedings of the XI Jornadas de Ingenieria del Software y Bases de Datos (JISBD'06). Barcelona, Spain, pp 173–182

Aldekoa G, Trujillo S, Sagardui G, Díaz O (2008) Quantifying Maintainability in Feature Oriented Product Lines. In: Tjiortjis C, Winter A (eds) Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08). IEEE, Athens, pp 243–247

Apel S, von Thein A, Wendler P, Größlinger A, Beyer F (2013) Strategies for Product-Line Verification: Case Studies and Experiments. In: Chang BH, Pohl K (eds) Proceedings of the 35th International Conference on Software Engineering (ICSE'13). IEEE, San Francisco, pp 482–491

Bagheri E, Gasevic D (2011) Assessing the maintainability of software product line feature models using strutural metrics. Softw Quality J 19(3):576–612

Basalaj W (2008) How to select a programming language subset to maximise software quality. In: Redmill F, Anderson T (eds) Proceedings of the 16th Safety-Critical Systems Symposium (SSS'08). Springer, Bristol, pp 43–56

Batory D, Cardone R, Smaragdakis Y (2000) Object-Oriented Frameworks and product lines. In: Donohoe P (ed) Software product lines, SECS, vol 576. Springer, pp 227–247

ter Beek MH, Damiani F, Lienhardt M, Mazzanti F, Paolini L (2019) Static analysis of featured transition systems. In: Duchien L, Thüm T (eds) Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC'19). ACM, Paris, pp 39–51

Benavides D, Segura S, Ruiz-cortés A (2010) Automated Analysis of Feature Models 20 Years Later: A Literature Review. Inf Syst 35(6):615–636

Bettini L, Crescenzi P (2015) An eclipse IDE for teaching java--. In: Lorenz P, Cardoso J, Maciaszek LA, van Sinderen M (eds) Proceedings of 10th International Conference on Software Technology (ICSOFT'15), Communications in Computer and Information Science 586. Springer, Colmar, pp 63–78

Bezerra CIM, Andrade RMC, Monteiro JM (2015) Measures for quality evaluation of feature models. In: Proceedings of the 9th International Conference on Software and Software Reuse (ICSR'15), Lecture Notes in Computer Science 8919. Springer, Miami, pp 282–297

Briand L, Morasca S, Basili VR (1994) Defining and validating High-Level design metrics technical report CS-TR, vol 3301. University of Maryland, Baltimore, MD, USA

Briand LC, Daly J, Porter V, Wüst J (1998) A comprehensive empirical validation of design measures for Object-Oriented systems. In: Proceedings of the 5th International Symposium on Software Metrics (METRICS'98). IEEE, Bethesda, pp 246–257

Briand LC, Daly J, Wüst J (1998) A Unified Framework for Cohesion Measurement in Object-Oriented Systems. Empir Softw Eng 3(1):65–117

Butting A, Eikermann R, Kautz O, Rumpe B, Wortmann A (2018) Controlled and extensible variability of concrete and abstract syntax with independent language features. In: Proceedings of the 12th International Workshop on Variability Modelling of Software Intensive Systems (VAMOS'18). ACM, Spain, pp 75–82

Card DN, Church VE, Agresti WW (1986) An empirical study of software design practices. IEEE Trans Softw Eng 12(2):264–271

Card DN, Page GT, McGarry FE (1985) Criteria for software modularization. In: Lehman MM, Hünke H, Boehm B (eds) Proceedings of the 8th International Conference on Software Engineering (ICSE'85). IEEE, London, pp 372–377

Cazzola W (2012) Domain-Specific Languages in few steps: The neverlang approach. In: Gschwind T, De Paoli F, Gruhn V, Book M (eds) Proceedings of the 11th International Conference on Software Composition (SC'12), Lecture Notes in Computer Science 7306. Springer, Prague, pp 162–177

Cazzola W, Chitchyan R, Rashid A, Shaqiri A (2018) μ-DSU: A Micro-Language Based Approach to Dynamic Software Updating. Computer Languages. Syst Struct 51:71–89. https://doi.org/10.1016/j.cl.2017.07.003

Cazzola W, Olivares DM (2016) Gradually Learning Programming Supported by a Growable Programming Language. IEEE Trans Emerg Top Comput 4(3), 404–415. https://doi.org/10.1109/TETC.2015.2446192. Special Issue on Emerging Trends in Education

Cazzola W, Poletti D (2010) DSL Evolution through composition. In: Proceedings of the 7th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'10). ACM, Maribor

Cazzola W, Vacchi E (2013) Neverlang 2: Componentised language development for the JVM. In: Binder W, Bodden E, Löwe W (eds) Proceedings of the 12th International Conference on Software Composition (SC'13), Lecture Notes in Computer Science 8088. Springer, Budapest, pp 17–32

Chen C, Alfayez R, Srisopha K, Boehm B, Shi L (2017) Why Is It Important to Measure Maintainability, and What Are the Best Ways to Do It? In: Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-companion'17). IEEE, Buenos Aires, pp 377–378

Chen Z (2000) Java card technology for smart cards: Architecture and programmer's guide. Addison-Wesley, Reading

Chidamber SR, Kemerer CF (1991) Towards a metrics suite for Object-Oriented design. In: Paepcke A (ed) Proceedings of the 6th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'91). ACM, Phoenix, pp 197–211

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493

Coleman D (1992) Assessing maintenaibility. In: Proceedings of the HP Software Engineering Productivity Conference (SEPC'92). HP, Palo Alto, pp 525–532

Coleman D, Ash D, Lowther B, Oman P (1994) Using Metrics to Evaluate Software System Maintainability. IEEE Comput 27(8):44–49

Colyer A, Rashid A, Blair G (2004) On the separation of concerns in program families technical report, vol 107. Lancaster University, Lancaster

Combemale B, Barais O, Wortmann A (2017) Language engineering with the GEMOC studio. In: Proceedings of the International Conference on Software Architecture Workshop (ICSAW'17). IEEE, Sweden, pp 189–191

Combemale B, De Antoni J, Baudry B, France RB, Jézéquel JM, Gray J (2014) Globalizing Modeling Languages. IEEE Comput 47:68–71

Cordy M, Classen A, Heymans P, Schobbens PY, Legay A (2013) Provelines: A Product Line of Verifiers for Software Product Lines. In: Proceedings of the 17th International Software Product Line Conference (SPLC'13 Workshops). ACM, Tokyo, pp 141–146

Crane ML, Dingel J (2005) UML Vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal. In: Briand L, Williams C (eds) Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05), Lecture Notes in Computer Science 3713. Springer, Montego Bay, pp 97–112

Degueule T, Combemale B, Blouin A, Barais O, Jézéquel JM (2015) Melange: a Meta-Language for Modular and Reusable Development of DSLs. In: Di Ruscio D, Völter M (eds) Proceedings of the 8th International Conference on Software Language Engineering (SLE'15). ACM, Pittsburgh, pp 25–36

Ebert C, Cain J, Antoniol G, Counsell S, Laplante P (2016) Cyclomatic Complexity. IEEE Softw 33(6):27–29

El-Sharkawy S, Yamagishi-Eichler N, Schmid K (2019) Metrics for analyzing variability and its implementation in software product lines: a systematic literature review. Inf Softw Technol 106:1–30

Erdweg S, Rendel T, Kästner C, Ostermann K (2011) Sugarj: Library-Based Syntactic Language extensibility. In: Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming (OOPSLA'11). ACM, Portland, pp 391–406

Erdweg S, van der Storm T, Völter M, Boersma M, Bosman R, Cook WR, Gerrtsen A, Hulshout A, Kelly S, Loh A, Konat GDP, Molina PJ, Palatnik M, Pohjonen R, Schindler E, Schindler K, Solmi R, Vergu V, Visser E (2013) The state of the art in language workbenches. In: Erwig M, Paige RF, Van Wyk E (eds) Proceedings of the 6th International Conference on Software Language Engineering (SLE'13), Lecture Notes on Computer Science 8225. Springer, Indianapolis, pp 197–217

Erdweg S, van der Storm T, Völter M, Tratt L, Bosman R, Cook WR, Gerritsen A, Hulshout A, Kelly A, Konat G, Molina PJ, Palatnik M, Pohjonen R, Schindler E, Schindler K, Solmi R, Vergu V, Visser E, van del Vlist K, Wachsmuth G, van der Woning J (2015) Evaluating and comparing language workbenches: Existing results and benchmarks for the future. Comput Lang Syst Struct 44:24–47

Favalli L, Kühn T, Cazzola W (2020) Neverlang and featureIDE Just Married: Integrated Language Product Line Development Environment. In: Collet P, Nadi S (eds) Proceedings of the 24th International Software Product Line Conference (SPLC'20). ACM, Montréal, pp 285–295

Fenton NE (1991) Software metrics: a rigorous approach. Chapman & Hall, London

Fernández-Sáez MA, Chaudron MRV, Genero M (2018) An industrial case study on the use of UML in software maintenance and its perceived benefits and hurdles. Empir Softw Eng 23(6):3281–3345

Fowler M (2005) Language Workbenches: The Killer-App for Domain Specific Languages? Martin Fowler's Blog. http://www.martinfowler.com/articles/languageWorkbench.html

Grönniger H, Rumpe B (2010) Modeling language variability. In: Calinescu R, Jackson E (eds) Proceedings of the 16th Monterey Workshop on Modeling, Development and VErification of Adaptive Systems (WMDVAS'10), Lecture Notes in Computer Science 6662. Springer, Redmond, pp 17–32

Halstead MH (1977) Elements of software science. Operating and programming systems. Elsevier

Hatton L (2007) Language Subsetting in an Industrial Context: A Comparison of MISRA C 1998 and MISRA C 2004. Inf Softw Technol 49(5):475–482

Hedin G, Magnusson E (2003) Jastadd — an Aspect-Oriented compiler construction system. Sci Comput Programm 47(1):37–58

Henry S, Selig C (1990) Predicting Source-Code Complexity at the Design Stage. IEEE Softw 7(2):36–44

Her JS, Kim JH, Hun OS, Rhew SY, Kim SD (2007) A Framework for Evaluating Reausability of Core Asset in Product Line Engineering. Inf Softw Technol 49(7):740–760

Hermans F (2020) Hedy: a gradual language for programming education. In: Robins A, Ko A (eds) Proceedings of ACM International Computing Education Research Conference. ACM, Dunedin

Hitz M, Montazeri B (1995) Measuring coupling and cohesion in Object-Oriented systems. In: Proceedings of International Symposium on Applied Corporate Computing, Monterrey

van der Hock A, Dincel E, Medvidović N. (2003) Using service utilization metrics to assess the structure of product line architectures. In: Proceedings of the 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IWENCHI'03). IEEE, Sidney, pp 298–308

ISO/IEC/IEEE International Standard (2017) Systems and software Engineering—Vocabulary. Standard. 24765–2017. https://doi.org/10.1109/IEEESTD.2017.8016712

Jézéquel JM, Méndez-Acuña D, Degueule T, Combemale B, Barais O (2014) When systems engineering meets software language engineering. In: Boulanger F, Krob D, Morel G, Rousse JC (eds) Proceedings of the 5th International Conference on Complex Systems Design & Management (CSDM'14). Springer, Paris, pp 1–13

Kaiser HF (1960) The Application of Electronic Computers to Factor Analysis. Educ Psychol Measur 20(1):141–151

Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS (1990) Feature-Oriented Domain analysis (FODA) feasibility study. Technical report, CMU/SEI-90-TR-21. Carnegie mellon university, Pittsburgh

Kim CHP, Bodden E, Batory D, Khurshid S (2010) Reducing configurations to monitor in a software product line. In: Barringer H, Falcone Y, Finkbeiner B, Havelund K, Lee I (eds) Proceedings of the First International Conference on Runtime Verification (RV'10), Lecture Notes in Computer Science 6418. Springer, St. Julians, pp 285–299

Krahn H, Rumpe B, Völkel S (2010) Monticore: A Framework for Compositional Development of Domain Specific Languages. Int J Softw Tools Technol Transfer 12(5):353–372

Krieter S, Schröter R, Thüm T, Fenske W, Saake G (2016) Comparing algorithms for efficient Feature-Model slicing. In: Rabiser R, Xie B (eds) Proceedings of the 20th International Systems and Software Product-Line Conference (SPLC'16). ACM, Beijing, pp 60–64

Krueger CW (2001) Easing the transition to software mass customization. In: van der Linden F (ed) Proceedings of the 1st International Workshop on Software Product-Family Engineering (PFE'01), Lecture Notes in Computer Science 2290. Springer, Bilbao, pp 282–293

Kühn T, Cazzola W (2016) Apples and oranges: Comparing Top-Down and Bottom-Up language product lines. In: Rabiser R, Xie B (eds) Proceedings of the 20th International Software Product Line Conference (SPLC'16). ACM, Beijing, pp 50–59

Kühn T, Cazzola W, Olivares DM (2015) Choosy and picky: Configuration of language product lines. In: Botterweck G, White J (eds) Proceedings of the 19th International Software Product Line Conference (SPLC'15). ACM, Nashville, pp 71–80

Kühn T, Cazzola W, Pirritano Giampietro N, Poggi M (2019) Piggyback IDE support for language product lines. In: Thüm T, Duchien L (eds) Proceedings of the 23rd International Software Product Line Conference (SPLC'19). ACM, Paris, pp 131–142

Kühn T, Leuthäuser M, Götz S, Seidl C, Aßmann U (2014) A Metamodel Family for Role-Based Modeling and Programming Languages. In: Combemale B, Pearce DJ, Barais O, Vinju J (eds) Proceedings of the 7th International Conference Software Language Engineering (SLE'14), Lecture Notes in Computer Science 8706. Springer, Västerås, pp 141–160

Lanza M, Marinescu R (2006) Object-Oriented Metrics in practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer

de Lara J, Guerra E (2012) Domain-Specific Textual Meta-Modelling languages for model driven engineering. In: Vallecillo A, Tolvanen JP, Kindler E, Störrle H, Kolovos D (eds) Proceedings of the European Conference on Modelling Foundations and Applications (ECMFA'12), Lecture Notes in Computer Science 7349. Springer, Kgs. Lyngby, pp 259–274

Liebig J, Daniel R, Apel S (2013) Feature-Oriented Language families: a case study. In: Collet P, Schmid K (eds) Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13). ACM, Pisa

Lima LG, Soares-Neto F, Lieuthier P, Castor F, Melfe G, Fernandes JP (2016) Haskell in green land: Analyzing the energy behavior of a purely functional language. In: Hassan AE, Zimmermann T, Di Penta M (eds) Proceedings of the 23rd International Conference on Software Analysis, Evolution and Reengineering (SANER'16). IEEE, Osaka, pp 517–528

Macro A, Buxton J (1987) The craft of software engineering. Addison-Wesley

Mann S, Rock G (2011) Control Variant-Rich models by variability measures. In: Czarnecki K, Eisenecker UW (eds) Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS'11). ACM, Namur, pp 29–38

McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng 2(4):308–320

Meinicke J, Thüm T, Schröter R, Benduhn F, Leich T, Saake G (2017) Mastering Software Variability with featureIDE. Springer

Meinicke J, Thüm T, Schröter R, Krieter S, Benduhn F, Saake G, Leich T (2016) FeatureIDE: Taming the Preprocessor Wilderness. In: Procedings of the 38th international conference on software engineering companion (ICSE'16-companion). IEEE, Austin, pp 629–632

Méndez-Acuña D, Galindo JA, Combemale B, Blouin A, Baudry B (2016) Puzzle: A Tool for Analyzing and Extracting Specification Clones in DSLs. In: Kapitsaki GM, Santana de Almeida E (eds) Proceedings of the International Conference on Software Reuse (ICSR'16), Lecture Notes in Computer Science 9679. Springer, Limassol, pp 393–396

Méndez-Acuña D, Galindo JA, Combemale B, Blouin A, Baudry B (2017) Reverse engineering language product lines from existing DSL variants. J Syst Softw 133:145–158

Méndez-Acuña D, Galindo JA, Degueule T, Combemale B, Baudry B (2016) Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. Comput Lang Syst Struct 46:206–235

Mosses PD (2019) Software Meta-Language Engineering and CBS. J Comput Lang:50 39–48

Ng K, Warren M, Golde P, Hejlberg A (2011) The roslyn project: Exposing the c# and VB compiler's code analysis. White paper, Microsoft

Oster S, Zorcic I, Markert F, Lochau M (2011) Moso-polite—tool Support for Pairwise and Model-Based Software Product Line Testing. In: Czarnecki K, Eisenecker UW (eds) Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS'11). ACM, Namur, pp 79–82

Parnas DL (1971) Information Distribution Aspects of Design Methodology. Inf Process:71 339–344

Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 15(12):1053–1058

Pearson K (1895) Notes on Regression and Inheritance in the Case of Two Parents. Proc R Soc Lond 58:240–242

Pereira JA, Krieter S, Meinicke J, Schröter R, Saake G, Leich T (2016) FeatureIDE: Scalable Product Configuration of Variable Systems. In: Kapitsaki GM, Santana de Almeida E (eds) Proceedings of the 15th International Conference on Software Reuse (ICSR'16), Lecture Notes in Computer Science 9679. Springer, Limassol, pp 397–401

Perepletchikov M, Ryan C, Frampton K, Tari Z (2007) Coupling metrics for predicting maintainability in Service-Oriented designs. In: Proceedings of the 18th Australian Software Engineering Conference (ASWEC'07). IEEE, Melbourne, pp 329–340

Perrouin G, Oster S, Sen S, Klein J, Baudry B, le Traon Y (2012) Pairwise Testing for Software Product Lines: Comparison of Two Approaches. Softw Qual J 20(3):605–643

Pinto G, Castor F, David LY (2014) Understanding energy behaviors of thread management constructs. In: Millstein T (ed) Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'14). ACM, Portland, pp 345–360

Pohl K, Böckle K, van der Linden FJ (2005) Software product line engineering: Foundations, Principles and Techniques. Springer

Pressman RS (2005) Software Engineering: A Practitioner's Approach, 6th edn. McGraw-Hill

Rosenmüller M, Siegmund N, Thüm Saake,G (2011) Multi-Dimensional Variability Modeling. In: Czarnecki K, Eisenecker UW (eds) Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS'11). ACM, Namur, pp 11–20

Schmid K (2002) A comprehensive product line scoping approach and its validation. In: Magee J, Young M (eds) Proceedings of the 24th International Conference on Software Engineering (ICSE'02). ACM, Orlando, pp 593–603

Sundermann C, Nieke M, Bittner PM, Łovasz-bukvova H (2021) Applications of #SAT Solvers on Feature Models. In: Grünbacher P, Seidl C, Dhungana D (eds) Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS'21). ACM, Krems, pp 1–10

Tavares E, Silva B, Maciel P (2008) An environment for measuring and scheduling Time-Critical embedded systems with energy constraints. In: Cerone A, Grüner S (eds) Proceedings of the 6th International Conference on Software Engineerig and Formal Methods (SEFM'08). IEEE, Cape Town, pp 291–300

Thüm T, Kästner C, Benduhn F, Meinicke J, Saake G, Leich T (2014) FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. Sci Comput Programm 79(1):70–85

Thüm T, Kästner C, Erdweg S, Siegmund N (2011) Abstract features in feature modeling. In: Schaefer I, John I, Schmid K (eds) Proceedings of the 15th International Systems and Software Product-Line Conference (SPLC'11). ACM, Münich, pp 191–200

Thüm T, Schaefer I, Kuhlemann M, Apel S (2011) Proof composition for deductive verification of software product lines. In: Proceedings of the International Conference on Software Testing Verification and Validation Workshop (ICSTW'11). IEEE, Berlin, pp 270–277

Tiwari V, Malik S, Wolfe A (1994) Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. IEEE Trans Very Large Scale Integr Syst 2(4):437–445

Tratt L (2008) Domain specific language implementation via Compile-Time Meta-Programming. ACM Trans Programm Lang Syst 30(6):31:1–31:40

Trefethen AE, Thiyagaligam J (2013) Energy-Aware Software: challenges, Opportunities and Strategies. J Comput Sci 4(6):444–449

Troy DA, Zweben SH (1981) Measuring the quality of structured designs. J Syst Softw 2(2):113–120

Vacchi E, Cazzola W (2015) Neverlang: a framework for Feature-Oriented language development. Comput Lang Syst Struct 43(3):1–40. https://doi.org/10.1016/j.cl.2015.02.001

Vacchi E, Cazzola W, Combemale B, Acher M (2014) Automating variability model inference for Component-Based language implementations. In: Heymans P, Rubin J (eds) Proceedings of the 18th International Software Product Line Conference (SPLC'14). ACM, Florence, pp 167–176

Vacchi E, Cazzola W, Pillay S, Combemale B (2013) Variability support in Domain-Specific language development. In: Erwig M, Paige RF, Van Wyk E (eds) Proceedings of 6^{th} International Conference on Software Language Engineering (SLE'13), Lecture Notes on Computer Science 8225. Springer, Indianapolis, pp 76–95

de Vasconcelos JB, Kimble C, Carreteiro P, Rocha A (2017) The Application of Knowledge Management to Software Evolution. Int J Inf Manag 37(1):1499–1506

Visser E, Wachsmuth G, Tolmach A, Neron P, Vergu V, Passalaqua A, Konat G (2014) A language designer's workbench: a One-Stop-Shop for implementation and verification of language designs. In: Black AP, Krishnamurthi S, Bruegge B, Ruskiewicz JN (eds) Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!'14). ACM, Portland, pp 95–111

Völter M (2011) Language and IDE modularization and composition with MPS. In: Lämmel R, Saraiva JA, Visser J (eds) Proceedings of the 4th International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'11), Lecture Notes in Computer Science, vol 7680. Springer, Braga, pp 383–430

Völter M, Pech V (2012) Language modularity with the MPS language workbench. In: Proceedings of the 34th International Conference on Software Engineering (ICSE'12). IEEE, Zürich, pp 1449–1450

Völter M, Ratiu D, Schätz B, Kolb B (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, pp 121–140

Wachsmuth GH, Konat GDP, Visser E (2014) Language design with the spoofax language workbench. IEEE Softw 31(5):35–43

Wende C, Thieme N, Zschaler S (2009) A Role-Based Approach towards Modular Language Engineering. In: van den Brand M, Gašević D, Gray J (eds) Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09), Lecture Notes in Computer Science 5969. Springer, Denver, pp 254–273

White J, Hill JH, Gray J, Tambe S, Gokhale A, Schmidt DC (2009) Improving Domain-specific Language Reuse with Software Product-Line Configuration Techniques. IEEE Softw 26(4):47–53

Wohlin C, Höst M, Henningsson K (2003) Empirical research methods in software engineering. In: Conradi R, Wang AI (eds) Empirical Methods and Studies in Software Enginering: Experiences from ESERNET, LNCS 2765. Springer, pp 7–23

Yau SS, Collofello JS, MacGregor T (1978) Ripple effect analysis of software maintenance. In: Proceedings of the 2nd International Computer Software and Applications Conference (COMPSAC'78). IEEE, Chicago, pp 60–65

Zhang T, Deng L, Wu J, Zhou Q, Ma C (2008) Some metrics for accessing quality of product line architecture. In: Proceedings of the International Conference on Computer Science and Software Engineering (ICCSSE'08). Wuhan, China, pp 500–503

Zschaler S, Kolovos DS, Drivalos N, Paige RF, Rashid A (2009) Domain-Specific Metamodelling languages for software language engineering. In: van den Brand M, Gašević D, Gray J (eds) Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09), Lecture Notes in Computer Science 5969. Springer, Denver, pp 334–353

Zschaler S, Sánchez P, Santos J, Alférez M, Rashid A, Fuentes L, Moreira A, Araújo J, Kulesza U (2009) VML*—A family of languages for variability management in software product lines. In: van den Brand M, Gašević D, Gray J. (eds) Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09), Lecture Notes in Computer Science 5969. Springer, Denver, pp 82–102

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Walter Cazzola is an Associate Professor in the Computer Science Department of the Université? degli Studi di Milano, Italy and the Chair of the ADAPT laboratory. Dr. Cazzola designed the mChaRM framework, @Java, [a]C#, Blueprint programming languages and he is involved in the designing and development of the Neverlang language workbench. He also designed the JavAdaptor dynamic software updating framework and its front-end FiGA. He has written over 100 scientific papers. His research interests include (but are not limited to) software maintenance, evolution and comprehension, programming methodologies and languages. He served on the program committees or editorial boards of the most important conferences and journals about his research topics. He is associate editor for the Journal of Computer Languages published by Elsevier. More information about Dr. Cazzola and all his publications are available at https://cazzola.di.unimi.it and he can be contacted at cazzola@di.unimi.it for any question.

Luca Favalli is currently a Computer Science PhD student at Université degli Studi di Milano. He is involved in the research activity of the ADAPT Lab and in the development of the Neverlang language workbench and of JavAdaptor. His main research interests are software design, software (and language) product lines and dynamic software updating with a focus on how they can be used to ease the learning of programming languages. He can be contacted at favalli@di.unimi.it for any question.