The Dimensions of Variability Modeling

or, On the Design of Software and Language Product Lines

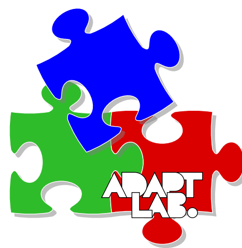
Luca Favalli

Id. Number: R12627

Scuola di Dottorato in Informatica
PhD in Computer Science

PhD School Headmaster: Prof. Roberto Sassi

Advisor: Prof. Walter Cazzola



UNIVERSITÀ DEGLI STUDI DI MILANO
Computer Science Department
ADAPT-Lab

Ciclo XXXV
INF/01 Informatica
Academic Year 2021-2022

COMPUTER SCIENCE DEPARTMENT • ADAPT-LAB

The secret of being a bore is to tell everything

—Voltaire

Contents

Preface 1

1 Introduction 3

2 Background 9

2.1 Software Product Line Engineering . . . 9
2.1.1 Features and Variability Modeling . . . 10
2.1.2 Software Product Line Development with FeatureIDE . . . 12
2.2 Domain-Specific Languages . . . 13
2.2.1 Internal and External DSLs . . . 13
2.2.2 The Implementation Dimensions of DSLs . . . 14
2.3 Language Product Line Engineering . . . 15
2.4 Language Workbenches . . . 15
2.4.1 Xtext . . . 16
2.4.2 LISA . . . 16
2.4.3 Melange . . . 16
2.4.4 Meta Programming System . . . 17
2.4.5 JastAdd . . . 17
2.4.6 MontiCore . . . 18
2.4.7 Rascal . . . 18
2.4.8 Spoofax . . . 18
2.4.9 Racket . . . 19
2.4.10 Neverlang . . . 19
2.4.11 Summary . . . 21

3 Reconciling Object-Oriented and Feature-Oriented Software Design 23

3.1 The Problem of Conservation of Complexity in SPLs . . . 23
3.2 Devise Pattern . . . 25
3.2.1 Purpose and Scope . . . 26
3.2.2 Intent . . . 26
3.2.3 Motivation . . . 26
3.2.4 Applicability . . . 27
3.2.5 Structure and Participants . . . 29
3.2.6 Collaborations . . . 29
3.2.7 Consequences . . . 30
3.2.8 Implementation and Sample Code . . . 31
3.2.9 Related Patterns . . . 35
3.3 Case Study: MNIST-encoder . . . 36
3.3.1 Neural Networks and the MNIST Dataset . . . 37
3.3.2 Application Overview . . . 40
3.3.3 Variability-aware Encoders . . . 41

3.3.4 Comparison: Non-Functional Properties 43
3.3.5 Comparison: Modeling Effort 46
3.3.6 Threats to Validity 46
3.4 Summary of Chapter 3 47

4 A Design Methodology for Language Product Lines 49
4.1 The Problem of Design Quality in LPLs 49
4.2 LPL Design Methodology in "Five" Steps 50
4.2.1 The LPL Engineering Process—Point 1 51
4.2.2 The LPL Engineering Environment—Point 5 55
4.2.3 Properties of a Well Designed Language Decomposition—Point 2 59
4.2.4 Metrics for the Detection of Design Errors in LPLs—Point 3 . . . 62
4.3 Case Study: Javascript Language Family 70
4.3.1 Family of Javascript-based Languages 70
4.3.2 Incrementally Teaching a Language 71
4.3.3 Growing Javascript Variants—The Deployer's Perspective 72
4.3.4 Refactoring the neverlang.JS LPL—The Developer's Perspective . 73
4.3.5 Comparison of the 15 Language Variants 75
4.4 Evaluation 77
4.4.1 Experimental Setup 77
4.4.2 Results 81
4.4.3 Principal Component Analysis 87
4.4.4 Thresholds 88
4.4.5 Discussion 90
4.4.6 Lessons Learned 91
4.4.7 Threats to Validity 95
4.5 Summary of Chapter 4 96

5 Mutation Testing based on Language Product Lines 99
5.1 The Problem of Quality in Languages Test Suites 99
5.2 Mutation Testing Overview 101
5.3 The Language Extension Problem 101
5.4 The Language Mutation Problem 102
5.4.1 Problem Overview 102
5.4.2 Resolution Meta-model 103
5.4.3 Consequences and Limitations 109
5.5 Solving the Language Mutation Problem in Practice 111
5.5.1 Mutation Testing in Neverlang 111
5.5.2 Neverlang Mutation Operators 112
5.6 Case Study: ECMAScript Conformance Test Suite 117
5.6.1 Setup 118
5.6.2 Results 119
5.6.3 Discussion 122
5.6.4 Lessons Learned 125

5.6.5 Threats to Validity 125
5.7 Summary of Chapter 5 127

6 Related Work **129**
6.1 Design Patterns for the Definition of SPLs 129
6.2 Language Product Lines Design Methodology 130
6.3 Mutation Testing of Language Implementations 131
6.3.1 Comparing the LMP with other Mutation Testing Techniques . . 132

7 Conclusions **137**

Bibliography **139**

Preface

What is a representation? First and foremost, a representation can be seen as some sort of model of the thing it represents [167]. In other words, a representation establishes the existence of a *represented* world and a *representing* world, as well as the existence of a relation between the two worlds. Such a relation maps objects of the represented world to objects of the representing world and, most importantly, determines what aspects of the objects from the represented world are being modelled by objects of the representing world. Equivalently, objects of the representing world express information about the world they represent in a structurally-preserving way [159]. According to representationalism, also known as indirect realism or epistemological dualism, the world we see in conscious experience is not the real world itself, but merely a miniature virtual-reality replica of that world in an internal representation. Representationalism is often disregarded due to it causing infinite regress fallacies such as the homunculus fallacy [116]: mental representations of the world are themselves symbols and thus they require interpretation. Although, it is the only alternative that is consistent with the facts of perception, that seem to suggest that our experience is limited by our senses and that senses may differ from the world itself, due to dreams and hallucinations.

Contemporary forms of representationalism escape the infinite regression fallacies by deploying self-interpreting representations, using computers and their programming languages as a powerful analogy. Fodor brings this analogy forth in his *The Language of Thought* [78]. Computers use at least two languages—*i.e.*, two abstraction levels with regards to representation: one (the high-level language) to communicate with the environment and the other (the machine-level language) to communicate with themselves, using a *compiler* as the mediator. A program written in the high-level language is a representation for a solution to a problem in the real world which is being represented. Then, the compiler establishes the relation between the represented high-level abstraction and the representing machine-level program. The machine-level language does not need further representation because it is self-interpreted: each of its instructions correspond directly to computationally relevant states and operations of the machine. In other words, the machine-level language is understood by the computer without further compilation, stopping the regression.

This dissertation deals with this analogy and its implications. Language is a vehicle of thought and programming languages are vehicles used to model problems and solutions of the real world. The way we think of programs and the way we write them are deeply connected. What if we bring this analogy one step further and try to model problems and solutions of the development of programming languages by introducing an additional layer of representation? How does modeling programming languages and their commonalities in a systematic way affect the way we develop them? Let us discuss the dimensions of variability modeling or, in other words, the design of software and language product lines.

1 Introduction

Modular software development is taken for granted. Modular programming dates back to the 1970s [170] and is a technique that emphasizes the need to separate different concerns of a software system into an independent module, so that the module itself contains all that is needed to express its concern with the least possible amount of dependencies to other concerns. Modules usually represent an interface that can be used by other modules to access the implementation of a concern without requiring any actual knowledge on the implementation details. Each programming language brings forth its own flavour of modularization to empower developers with better abstractions and an increased ability to reuse existing assets in a different context. Some languages may even provide several modularization techniques to further support reusability at different levels of granularity. The choice of a modularization technique is a primary decision with regards to the design of a programming language and it is often deeply connected to the programming paradigm and the intended development approach in general. As a result, the concept of modular programming lacks a set definition and it is more generally used to refer to the process of decomposing a complex software systems into smaller and simpler pieces.

Object-oriented programming has probably been the most commonly adopted programming paradigm in recent decades, due to the popularity of languages such as C++[1], Java[2] and Python[3]. In object-oriented programming, code modularization and reuse is based on the concept of objects, which contain both data and code in the form of fields and methods respectively. Most object-oriented programming languages adopt a manyfold modularity structure that entails the concept of classes as the primary unit of code organization, but also modules (in Python), namespaces (in C++), or other similar concepts. Java supports packages that are akin to modules in other languages, although a new module concept was introduced since Java 9 to describe collections of packages with enhanced access control, including how they can be discovered and loaded.

Functional programming is a programming paradigm in which programs are the result of the definition and composition of functions. Functions are treated as first-class citizens that can be bound to identifiers and passed as arguments to manage the execution flow of a program, for instance by using the continuation approach. As such, functions are the primary unit of reuse in functional programming languages

[1] https://isocpp.org/
[2] https://www.java.com/
[3] https://www.python.org/

and they are often grouped into modules. Functional programming languages such as OCaml[4] allow for the dynamic definition of modules through the usage of functors—*i.e.*, modules that are parametrized by another module, just like functions are parametrized by their arguments. Functors are commonly used to customize the behavior of a module with regards to a specific abstract data type, for instance by turning a module for generic set operations into a module for operations on set of strings.

Actor-based programming is a model for concurrent computation in which each individual concurrent process is an actor that can make local decisions and communicate with other actors by means of messages. A normal actor-based program can be made of several hundreds of small actors running concurrently and thus actors are the primary unit of reuse. Among the most well-known actor-based programming languages, Erlang[5] and Elixir[6] adopt a hybrid paradigm mixing actor-based and functional programming and leverage the power of the BEAM virtual machine to deploy lightweight concurrent actors.

This draw towards hybridity is shared amongst all modern languages. Language designers are constantly looking for ways to extend and enrich programming languages with new constructs and abstractions to close the gap between human thought and machine behaviour. Languages that were born as purely object-oriented, now include functional constructs, such as the *functional interface* introduced in Java 8, and vice-versa. When a paradigm is not supported by the base language, the community comes in to produce language extensions towards their support. For instance, AspectJ[7] is a seamless **aspect-oriented** extension to the Java programming language used to model crosscutting concerns such as logging and error-checking whereas Akka[8] is an API for the development of actor-based distributed systems in Java and Scala[9]. As a result of this trend, programming languages are nowadays complex software systems that tend to contaminate one another. Despite their similarities, programming language implementations often lack any modularization and hardly share any code: programming language development is still a top-down monolithic activity in which extensibility is a mere afterthought. More recently, the trend has shifted towards the development of programming languages targeting a specific application domain. For the users to be able to speak their mind, programming languages must be designed so that they do not get in the way of their thought. **Domain-specific languages** (DSLs) try to answer to this need by supposedly limiting comprehensiveness in favor of comprehensibility. The goals are to decrease the development time and to improve the quality of the final product by overcoming the communication barrier between the domain experts and the developers: many of the failures in the development of software projects are caused by the difficulty of translating requirements into specifications and specifications into implementations, due to the lack of a common vocabulary. Instead,

[4] https://ocaml.org/
[5] https://www.erlang.org/
[6] https://elixir-lang.org/
[7] https://www.eclipse.org/aspectj/
[8] https://akka.io/
[9] https://www.scala-lang.org/

DSLs employ terms and concepts that are specific to an application domain, so that domain experts can participate in the development activity by validating specifications written by others or even by expressing new ones themselves: this makes it easier to communicate with domain experts, providing both a description and a solution for the problem [81]. Moreover, due to the simplicity of DSLs, it is generally considered much easier and faster to train the developers on how to use a DSL rather than a general-purpose programming language (GPL). Finally, DSLs enforce the adoption of shared code style, development conventions and best practices across the whole team by means of the compiler. DSLs are often designed as an extension to a general-purpose base language. In this case, the GPL is called *host* and *embeds* the *guest* DSL. DSLs come in two main forms: internal and external [81]. An *internal* DSL is a particular style of API, often referred to as *fluent interface* [79], in which API calls are designed to be easily readable. Instead, an *external* DSL is a language that is parsed separately from the host language.

This shift in the requirements of language development calls for a shift in the programming paradigm. A paradigm in which software is built around a set of DSLs; rather than solving problems using a GPL, the system is first split into its application domains according to such problems and then a DSL is implemented for each domain. Finally, each problem is assigned the most suitable application domain and is solved by using the corresponding DSL. In other words, a **language-oriented programming** paradigm. The term language-oriented programming was firstly introduced by Martin Ward in 1994 [223] following the Unix tradition of *little languages* [20], but such a style of development was ultimately deemed unsuccessful due to the complexity of computer languages and their ecosystems. Programming languages are integrated across an ecosystem of different applications that are usually modelled separately. This ecosystem includes not only the language compiler or interpreter but also an integrated development environment comprehensive of syntax highlighting, code completion, error recovery and a debugger. Moreover, changes to the language may render the entire application ecosystem obsolete, thus requiring each component to be updated. More recently, there has been an attempt to support language-oriented programming through a new breed of tools, by easing the development of programming languages and their ecosystems in a modular way; following the seminal work of Martin Fowler [80], we dub such a breed of tools **language workbenches**. While the original definition focused on projectional editing [80, 219], research on language workbenches is currently focusing on their promise of supporting the efficient definition, reuse and composition of languages and their IDEs. Modern language workbenches evolved according to many different design philosophies, but they all share the same goal: to assist the creation as well as the usage of programming languages in a unified environment. Language workbenches typically offer meta-languages and abstraction that facilitate the development of languages and the reuse of software artifacts. A key aspect of language workbenches is their ability to support **modular language development**—*i.e.*, the development of languages in a componentized way. Composition can be modelled according to several different reuse models, like inheritance [154] and superimposition [139].

Language workbenches benefit from the application of a **feature-oriented program-**

ming approach to the development of DSLs. Firstly introduced as part of the FODA method [114], feature-oriented programming is an approach in which software systems are described in terms of the features they provide—*i.e.* product characteristics that a customer would feel are important in describing the product itself [7]. Such software systems are then modelled using formalisms such as the *feature model* (FM). The main goal is to achieve separation of concerns—*i.e.*, features are implemented separately. However, feature-oriented programming and feature models also fit an engineering approach that is a staple in industrial production: **product line engineering**. A product line is a family of similar products, whose members are differentiated by their features. Similarly, a **software product line** (SPL) is a product line of software products. More recently, researchers gained interest in applying software product line engineering principles to the development of programming languages, thus originating the concept of **language product lines** (LPL)—*i.e.*, SPLs in which the software product is a programming language interpreter or a compiler, together with its ecosystem. The motivations behind SPLs and LPLs are both qualitative and economic. From a qualitative standpoint, complex software systems are hard to develop, maintain and comprehend: SPL engineering tries to control such a complexity with high-level design techniques. From an economics standpoint, the developers are able to amortize the costs of building variants of a program to satisfy several different customers.

This dissertation focuses on several design aspects of SPLs, with a particular attention given to the topic of LPLs. Most of the discussion will be based on the Neverlang language workbench, as an example used throughout this work to present the LPL engineering approach, its challenges and solutions. Neverlang was originally born to explore feature-oriented modularization in language design and implementation [37, 35] and was later overhauled to support a highly-dynamic componentized approach to language development framework [38, 209]. More recently, Neverlang has evolved into a full-fledged language workbench with integrated development environment services support [131]. The goal of our research was to complement such a collection of tools and techniques with a proper design methodology. The goal is to drive the development of LPLs so that their quality can be assessed at any moment, to ensure a truly modular structure of language-oriented software systems and ultimately to maximize reuse of syntactic and semantic assets across several language definitions.

Contribution. Our contribution with this work involves several design aspects of LPLs and SPLs in general and includes:

1. an approach to the realization of SPLs based entirely on design patterns without the support of any external tool, so that the developers can produce software features using abstractions they are used to;
2. an LPL design methodology that encompasses an engineering process to determine the order in which decisions are made, an integrated development environment for LPL designers, the properties of a well-structured language decomposition and the metrics for assessing the quality of a language decomposition;

3. the characterization of the language mutation problem and a framework based on LPLs for its resolution, to enable the quality assessment of test suites for programming languages through mutation testing, but without encompassing re-compilation.

Outline. This work is structured as follows. Chapter 2 introduces any concepts and terminology that are relevant throughout this work. From Chapter 3 onward we will discuss several design aspects of software and language product lines. Chapter 3 discusses the design of SPL at large by overviewing a design pattern for their implementation. The proposed pattern answers some of the limitations of annotative and composition approaches traditionally used for their definition and tries to solve *the problem of conservation of complexity in SPLs*. Then, we will deviate from the topic of SPLs in general to focus specifically on LPLs and their design methodology. Chapter 4 takes on the works of Parnas, Briand and Coleman on design methodologies for object-oriented systems and adapts their contribution to the topic of LPLs to solve *the problem of design quality in LPLs*. A design methodology for LPLs should improve the quality of language decompositions and, in turn, the reusability of language assets. Chapter 5 tries to solve *the problem of quality in test suites for programming languages*: it defines and tackles the language mutation problem using LPLs and language workbenches as a tool to improve mutation testing approaches on programming languages. Chapter 6 discusses any related work with regards to the SPL and LPL design aspects discussed throughout chapters 3, 4 and 5, including works that try to solve similar problems with a different technique or that use similar techniques to solve different problems. Finally, in Chapter 7 we will draw our conclusions on this work and outline some future directions with regards to the research on the design of SPLs and LPLs.

2 Background

This work builds on top of the existing literature on SPLs, language workbenches and LPLs. To better discuss our contribution, we provide any background information, including definitions, terminology and concepts that will be referred to throughout this dissertation. First, in Sect. 2.1, we provide an overview on the field of software product line engineering, as well as the concept of features, their variability and feature-oriented programming. Then, in Sect. 2.2, we introduce DSLs as the main building blocks of language-oriented software systems. In Sect. 2.3, we combine the concepts of SPLs and DSLs to introduce the field of language product line engineering. Finally, in Sect. 2.4, we will overview the topic of language workbenches and discuss some concrete implementations with regards to their approaches to modularization and language composition.

2.1 Software Product Line Engineering

Variability of products is very common in industrial production. Take a car factory as an example: the base car model can be optionally customized with different colors, a navigation system, a cooling system and parking sensors. Such a kind of industrial production, called *variability-rich* production, is a problem that has been dealt with in classical engineering environments through the creation of product lines. Software product line engineering (SPLE) [214, 213] tries to apply the same ideas and concepts to the scope of variability-rich software, following the dream of massive software reuse. The main goal of such an extensive reuse effort is to leverage the similarities among software products pertaining the same domain. Eventually, SPLE should capture these similarities to minimize the costs of deploying similar products that suit the needs of several different customers or to ease the evolution of a single product. As a result, SPL is a paradigm to create a design for families of programs [15]: all members of the *software family* will share a set of commonalities called *core features*, whereas any non-core features will determine the differences between two members of the family, which will be called *variants*. A tempting approach to the creation of software variants is to *clone-and-own* [185]: the cost of making copies of software source code is close to zero, hence one may be lead to slightly change a duplicated piece of code to create a variant of the original implementation in which the functionality is slightly modified according to the customer's requirements. However, *clone-and-own* quickly leads to maintenance hell, since any update applied over any of the variants also has to be applied to the

duplicated code of all other variants. Therefore, researchers and practitioners struggle to improve the theory behind feature reuse and always try to provide new technology to drive software variability management and deployment of software variants with minimal code duplication.

Whether SPLs are developed by means of clone-and-own or through more sophisticated techniques, SPLE follows one of three different approaches [128]:

- the *proactive approach*, in which the SPL is created from scratch;
- the *extractive approach*, in which an existing code base with different software products sharing some commonalities is converted into an SPL;
- the *reactive approach*, in which the development starts from an initial set of core features and then the SPL is developed incrementally by adding more features to the initial set.

In each of the three approaches, the development of an SPL is generally divided into four phases [7]: domain analysis, requirement analysis, domain implementation and product derivation. Domain analysis and requirement analysis pertain the *problem space* of the SPL whereas domain implementation and product derivation pertain its *solution space*. During the domain analysis phase, the goal is to capture the scope of the application domain and to derive its description—usually in terms of a model representing its features. The requirements analysis phase is concerned with the assessment of the concrete requirements of the customers and the definition of the relevant product variants. The domain implementation phase turns desired functionalities into concrete implementations, by mapping features from the domain to reusable software artifacts. Finally, the product derivation phase joins the requirements collected during the second phase with the software artifacts developed during the third phase to yield a member of the software family. This process is usually performed automatically by a software called *composer*.

2.1.1 Features and Variability Modeling

SPLs rely on the notion of *features* and are often developed using a feature-oriented programming (FOP) paradigm. Each feature represents some (either functional or non-functional) characteristics of a subset of the members of the software family. However, there is still no commonly accepted formal definition of software features. Instead, a feature is informally described as an increment [15]—*i.e*, a characteristic or end-user-visible behavior of a software system [7]. Given this informal definition, an SPL can be viewed as the collection of all the available features whereas a *product configuration* is a valid subset of the SPL; each configuration always corresponds to a member of the software family. While many approaches to express SPLs exist in literature, *feature models* (FM) are considered the de facto standard for variability modeling [57].

A FM is usually represented as a hierarchical tree structure called *feature diagram* that contains all possible features that exist in an SPL uniquely identified by a name, as well as any dependencies between each other. For the purpose of simplicity, in this dissertation we will refer to both the representation and the underlying model with the

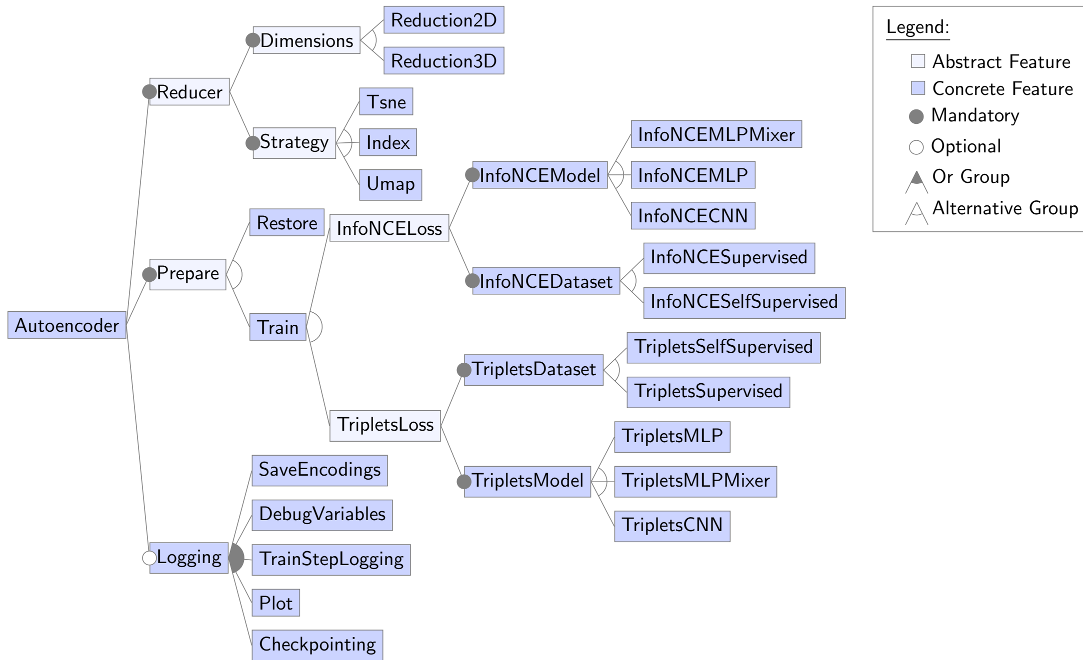


Figure 2.1: *FM of a family of neural networks used to encode the MNIST[1] dataset, taken from [21].*

term FM, even though the two concepts are technically different. An example for a FM is shown in Fig. 2.1.

Given a FM and a valid configuration for that model, 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 using specific types of features. For instance, a FM can contain *abstract features*—such as `Reducer` and `Prepare` in Fig. 2.1—and *concrete features*—such as `Logging` and `Umap` in Fig. 2.1. Abstract features are structural features that are used to collect groups of child features under the same abstraction, but have no implementation associated to them. Concrete features are actual features with an implementation that can be composed into a software product as part of the product derivation process. Features are *mandatory*—such as `Strategy` in Fig. 2.1—if they must always be selected if the parent feature is selected in the same configuration. All non-mandatory features are called *optional*. Multiple features that share the same parent feature can also express additional constraint with one another. *Or* groups—such as the children of the `Logging` feature in Fig. 2.1—manifest the intent that, if the parent feature is active, then one or more of the children must also be active in the same configuration. Similarly, *alternative* groups—such as the children of the `Dimensions` feature in Fig. 2.1—manifest the intent that, if the parent feature is active, then exactly one of the children must also be active in the same configuration.

[1] `http://yann.lecun.com/exdb/mnist/`

In addition, dependencies can be defined explicitly using *cross-tree constraints*, included in the form of logical formulas whose 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. Cross-tree constraints support general Boolean operators such as AND, OR, NOT, IMPLIES and IFF with the traditional meaning. The expressiveness of cross-tree constraints is used to enable the definition of constraints that span the entire feature model instead of being limited to the relation between parent and child features. 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 [19] and behavioral [18] approaches.

2.1.2 Software Product Line Development with FeatureIDE

FeatureIDE[2] [203, 151, 150] is an SPL development environment that copes with all aspects of the development of SPLs. It supports the FM construction, the management of software artifacts, the products configuration and their 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 [173];
3. various *Composers*—which can be customized and extended through dedicated insertion points—for the derivation of product variants from a given valid feature configuration.

As an example, Fig. 2.1 represents a FM as shown by the Feature Model Editor of 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 by translating the feature diagram and its cross-tree constraints to a propositional formula, so that a satisfiability solver can be used to reason about it. Finally, the chosen composer generates the final product variant for a given configuration by combining the active features from that configuration.

[2] `https://featureide.github.io/`

2.2 Domain-Specific Languages

A domain-specific language (DSL) is a *computer programming language of limited expressiveness focused on a particular domain* [81]. Albeit general, this definition by Martin Fowler includes all the key elements of DSLs. DSLs are tailored to a particular application domain and are therefore also referred to as application-oriented, special purpose, specialized, task-specific, or application languages in literature [155]; regardless of naming conventions, DSLs offer appropriate notions and abstractions to express solutions with respect to the problems of a particular application domain [64]. Using DSLs intuitively brings several advantages to the development of complex systems:

- *encapsulation*—DSLs hide implementation details behind proper abstractions;
- *productivity*—powerful abstractions smooth the coding phase;
- *communication*—domain experts can read, understand and validate code, or even participate in its development;
- *quality*—DSLs provide a natural mapping between specification documents and their implementation.

It is apparent that the main focus of this discussion has been *abstraction*. In fact, a library or a framework are no different from a DSL since they both share the same purpose, but libraries are usually interacted with through an *application programming interface* (API) whereas DSLs provide a style of manipulation that may be more appropriate for a non-programmers, by using natural languages instead of function calls and parameters. A library can even be designed as a internal DSL, as we will discuss in the next section.

2.2.1 Internal and External DSLs

There are two main types of DSLs: internal (or embedded [81]) and external. *Internal DSLs* are embedded into an existing language called *host language* and can only operate within its boundaries. The main advantage of internal DSLs is their ease of development, since they can be developed as an easily-readable *fluent interface*. On the other hand, developers have to face reduced flexibility, since an internal DSL is limited to the constraints of the host language, including programming paradigm, type system, tooling, or more generally both its syntax and semantics. While semantics limitations may not effect the expressiveness of the DSL, since the host language is usually a GPL, syntactic limitations may limit the usefulness of an internal DSL due to the inability to express terms and idioms of a particular domain within the boundaries of the host language. Some prominent examples of host languages that are suited to the development of internal DSLs are Lisp with its macros [80], Ruby [81] and Scala [10].

On the contrary, creating an external DSL involves the construction of a new language from scratch. This process includes writing a complete specification for the DSL discussing the needed abstractions, the language syntax and its semantics. External DSLs are usually more suitable for the involvement of domain experts in the development activity since they can be customized according to the syntax used in that particular domain but are also hard to develop since they require a custom compiler.

While grammar-based compilers and interpreters are the most common pattern for the development of external DSLs [123], developers sometime rely on existing tools and their syntax (such as, XML, JSON an other configuration and serialization languages [81]) to create an external DSL while avoiding the need for a custom grammar. Some prominent examples of external DSLs are SQL for querying databases, LaTeX for typesetting documents and Make for build automation.

2.2.2 The Implementation Dimensions of DSLs

DSLs are usually defined with regards to three different dimensions: *abstract syntax*, *concrete syntax* and *semantics*.

The abstract syntax defines the language constructs that are used to represent the application domain. To enable further processing by means of an interpreter or a compiler, parsing tools convert a source file into a representation called *abstract syntax tree* (AST) based on the abstract syntax of the language. The abstract syntax can be defined by means of a grammar comprised of production rules [3] or through a meta-model. The most commonly used grammars for the definition of programming languages are *context free grammars*. A context-free grammar is a tuple $G = (\Sigma, N, P, S)$ where Σ is an alphabet of terminal symbols, N is an alphabet of nonterminal symbols, P is a set of production rules (or productions, for simplicity) and $S \in N$ is the start symbol. A production is written as $A \rightarrow \omega$ where $A \in N$, and $\omega \in (\Sigma \cup N)*$, with $(\Sigma \cup N)*$ being the transitive closure of set $(\Sigma \cup N)$ with respect to symbol juxtaposition. The generated language $L(G)$ of a grammar is the set of all the words that can be derived from a grammar G. A language for a grammar G is said to be empty if $L(G) = \emptyset$ and, conversely, non-empty when $|L(G)| > 0$. A *meta-model* is generally used in model-driven engineering and represents a given domain in terms of its concepts, its properties and their relations [60]. Although meta-models can be used for the definition of DSLs, we will not discuss meta-model-based DSL implementation in this work.

The *concrete syntax* represents the AST as it is viewed and manipulated by the user [60]. The most common representation of the concrete syntax is in textual form, although purely graphical representations are possible. Finally, a mixed representation of textual and graphical concrete syntax of a language is also possible by means of *projectional editing* [219].

The *semantics* of a DSL are necessary to attach a meaning to language constructs. Semantics can either be *static* if they cannot be attached to the abstract syntax such as the concepts of scope, variables and exceptions. *Dynamic semantics* can be attached to the AST and express the runtime behavior of the language. One way to associate dynamic semantics to the abstract syntax of a language is the *syntax directed translation* [3] technique, based on *attribute grammars* [121]. Intuitively, a *syntax-directed definition* can attach attributes to the grammar symbols representing the construct. This is done by program fragments embedded within production rules and called *semantic actions*.

2.3 Language Product Line Engineering

The idea of applying SPL concepts to the creation of families of programming languages and DSLs in particular has gained popularity among researchers and practitioners [86, 132, 35], thus introducing *language product lines* (LPLs) [211, 130, 129]. To enable the development of an LPL, a modular structure of the members of the family in the form of language components (or language modules) is necessary. As discussed in Sect. 2.2.2 a DSL and all its language components must be defined with regards of the three implementation dimensions of abstract syntax, concrete syntax and semantic descriptions. The definition of any of these three elements may cause dependencies to other components, limiting the reusability of the same component across all members of the language family. Just as for SPLs, a main aspect of LPL engineering (LPLE) is to capture commonalities and differences among members of the language family, by defining the variation points and the granularity of language components, as well as any composition mechanism. To be applicable to the development of DSLs and programming languages in general, LPLE must encompass several design aspects, including proper tooling and methodologies.

2.4 Language Workbenches

The LPL approach may prove useful to the creation of variants of a domain-specific language [55, 207, 210] and *dialects* of a general-purpose programming language [34]. LPLE benefits from the creation of *sectional compilers* that support the development of language features separately, including their syntax, their semantics and meta-data for reusable IDE specifications. Most recent *language workbenches* [71] embrace this philosophy to improve reusability and maintainability of language assets. The term *language workbench* was firstly introduced by Fowler [80] to describe tools suited to the *language-oriented programming* paradigm [223], in which complex software systems are built around a set of domain-specific languages to properly express domain problems and their solutions. While the original definition focused on projectional editing [80], research on language workbenches is currently focusing on their promise of supporting the efficient definition, reuse and composition of languages and their IDEs. Current language workbenches evolved according to many different design philosophies, but they all share the same goal: to facilitate the development of languages and the reuse of software artifacts through better abstractions.

Following the feature-oriented programming paradigm, a reusable piece of a language specification is called *language feature*. A language feature is formed by a syntactic asset and a semantic asset and represents a language construct together with its behavior. A language feature can omit the semantic asset or the syntactic asset; these corner cases represent a language construct without semantics and semantics that are not associated to any syntax respectively. For instance, comments can be implemented as a language feature in which the semantic asset is omitted. Languages and their features can be composed to form new language variants according to five forms of

language composition: language extension, language restriction, language unification, self-extension, and extension composition [70]. The goal of a language workbench is to support all five forms of language composition through dedicated abstractions.

Language workbenches have been developed for various technological spaces. The rest of this section presents an incomplete summary on current language workbenches in no particular order. For a complete overview and comparison of language workbenches, their capabilities, and their support to LPLE, please refer to the works of Erdweg *et al.* [71] and Méndez-Acuña *et al.* [153]. Instead, we discuss Neverlang (Sect. 2.4.10) in particular detail, since most of the research discussed in this dissertation will use Neverlang as a running example.

2.4.1 Xtext

Xtext [22] is a DSL development framework developed by the Eclipse Foundation and intended to be used with the Eclipse IDE. The grammar for the DSL is written using the Xtext language, that is then translated into a ANTLR-based parser, an Eclipse Modeling Framework (EMF) [199] and eventually into Java classes, each representing an AST node type. Xtext is fully integrated with Eclipse and can generate a plugin for the language being developed. Semantics are defined using Xtend, a Java-like language, or the Xsemantics [22] language for formal semantics specifications. Despite this premise, Xtext flexibility options are limited, since ANTLR requires the entire grammar of the language to be defined and compiled at the same time.

2.4.2 LISA

LISA [157] is an interactive environment for the development of programming languages developed at the University of Maribor. LISA leverages concepts derived from object-oriented programming, such as templates and multiple attribute grammar inheritance [156] to overcome the limitations of ordinary attribute grammars. LISA consists of several tools, both textual and graphical, to ease the development of programming languages and their ecosystems. Several language-based tools can be derived from a language specification [97]: editors, inspectors, evaluators and a graphical algorithm animator and program visualizer, to improve the user ability to understand the meaning of the source program that is being processed.

2.4.3 Melange

Melange [61] is a language workbench developed by the DiverSE research team at the Institut National de Recherche en Informatique et en Automatique (INRIA). The main design goal is the modular development of DSLs. Melange is based on various tools from the Eclipse Modeling Framework (EMF). The Ecore modeling language is used to define the language abstract syntax, Xtext to define textual concrete syntax and Sirius [220] to define graphical concrete syntax. Semantics are attached to the syntax in an aspect-oriented fashion using Kermeta 3 [107] aspects. These components are

put together using a Melange specification file that declares which elements have to be composed and how. Other composition mechanisms include language extension and language merging. The resulting composition is translated into an Eclipse plugin by the Melange compiler. Melange answers three design needs [60]:

- the introduction of a *language interface* to hide the complexity of language implementations by exposing only meaningful information;
- the usage of polymorphism to provide a model-based type system;
- language implementation reuse through extension and composition of existing operators.

Melange supports some flexibility in the composition of assets by allowing for the redefinition of Ecore elements, such as packages and classes, but with no support towards composition of incompatible concrete syntax elements.

2.4.4 Meta Programming System

Meta Programming System (MPS) [217] is a language workbench developed by the JetBrains[3] company, which has become very popular due to their high-quality IDE tools. MPS is not designed for the development of textual DSLs and rather it provides tools for the developer to directly interact with the AST of a program. Such an approach is called *projectional editing* [219]. Projectional editing was the main focus of Martin Fowler's essay «Language Workbenches: The Killer-App for Domain Specific Languages?» [80], which helped bringing language workbenches and language-oriented programming to a mainstream audience. According to Fowler, projectional editing was one of the key factors in the definition of language workbenches. Although this requirement relaxed over time, leading to a more diverse plethora of language workbenches with different design philosophies, projectional editing remains one of the most interesting approaches to the development of DSLs that are suitable to be used by non-programmers. In MPS, projectional editing is based on views over the AST called *editors*. Each user can interact with the program though any compatible editor, while all the available views are kept up to date through serialization of the underlying AST. The building blocks of the AST data structure are called *concepts* whereas their semantics are implemented by means of attachable components called *behaviors*. Behaviors and concepts can be extended through simple and multiple inheritance to achieve some degree of flexibility over language implementation.

2.4.5 JastAdd

JastAdd [68] is a modular compiler construction system developed at the Computer Science department of the Lund University. Language development in JastAdd revolves around the modeling of the AST as an object-oriented class hierarchy, whose semantics are implemented in an aspect-oriented fashion, using methods and aspects. The JastAdd compiler translates input to Java source files, building a Java class for each AST node.

[3] https://www.jetbrains.com/

The compiler instruments those classes with the proper methods based on its aspects. JastAdd does not support concrete syntax definition, but it can be added by means of an external Java-based parser generator such as ANTLR[4] and JavaCC[5]. Additional tools, such as JastAddParser[6] and Concrete [88] fill the gap of any additional language tooling support. Flexibility in the composition of language components is limited by them needing the same signature.

2.4.6 MontiCore

MontiCore [126] is a language workbench developed by the Software Engineering group at the RWTH Aachen University. MontiCore uses a unique DSL for the definition of both abstract and concrete syntax, that are then translated into Java classes by the compiler. Each Java class represents either a node type of the AST or an ANTLR-based parser. Semantics are implemented as visitors [82] that define the actions to be taken when each node type is encountered during the AST visit (either preorder or postorder). Attribute access is granted by means of getter and setter methods generated by the compiler. While this eases the generation and type checking aspect of the attribute grammar, it also limits its flexibility, since new attributes cannot be added to an existing grammar at a later time.

2.4.7 Rascal

Rascal [120] is a language for the development of language processing tools. Rascal is developed by the Software Analysis and Transformation (SWAT) research group at the Centrum Wiskunde & Informatica (CWI) in Amsterdam. It provides context-free grammar primitives, Algebraic Data Types and templates in a unified language [119]. The abstract syntax is expressed with abstract data types, each with its constructors that are used by the parse library to transform textual input into an AST. However, the usage of constructors limits the flexibility of the tools, since constructors for a given abstract data type must match across apparently unrelated modules.

2.4.8 Spoofax

Spoofax [115] is a language workbench developed at the Delft University of Technology. Spoofax combines several DSLs into a unique Eclipse-based solution. It includes Syntax Definition Formalism (SDF3) for grammar specifications and Stratego to perform transformations over the AST through tree rewriting *rules* and *strategies*. Tree rewriting rules define a transformation from a term to another based on pattern matching. If the input AST matches the pattern, then the tree is transformed according to the rule, otherwise no change is performed over the input. Strategies define a way to apply rules to the terms of a tree, without manually specifying the visit algorithm, leaving it to

[4] http://www.antlr.org/
[5] https://javacc.org/
[6] http://jastadd.org/web/tool-support/jastaddparser.php

be defined by the strategy itself. Some examples are *innermost*, which applies a rule to all the nodes in a tree starting from the leafs, *all*, which applies it to all the direct children of the root and *one*, which applies it to the first one for which a matching pattern is found. Such an approach provides a lot of flexibility, since semantic reuse can be achieved through tree rewriting and the application of rules is limited only by their arity.

2.4.9 Racket

Racket is presented as a "programmable programming language" [73] designed to explore the emerging trend of language-oriented programming. Racket empowers programmers with the ability to create new languages and to add them to a codebase, so that extra-linguistic mechanisms are turned into linguistic constructs [73]. Language extension is supported through several mechanisms such as syntactic abstractions and syntactic extensions [77]. Some tasks require only a small extension to the core language, while other benefit from the introduction of an entire new DSL. The definition of a language goes through the definition of its syntax, its static semantics and its dynamic semantics, usually by mapping new syntactic constructs to elements of the host language via a foreign-function interface. Racket also ensures that the invariants of each language in the multilingual system are respected. Each language implemented in Racket extend the Racket ecosystem, effectively bridging the gap between libraries and extra-linguistic mechanisms. Nonetheless, Racket has some limitations such as, for instance, with regards to type checking: it is currently implemented as a complete recursive descent algorithm, whereas developers want to attach type-checking rules to linguistic constructs.

2.4.10 Neverlang

Neverlang [31, 39, 209] is a language workbench for the modular development of programming languages developed at the University of Milan. 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 [3]. 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

```
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 2.1: *Syntax and semantics for the backup task.*

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 2.1 illustrates the implementation of the `Backup` feature of the `LogLang` DSL, a language for log rotating tools similar to the *logrotate* Unix utility. The `Backup` module declares a reference syntax for the backup task (lines 2-10). The reference syntax of a module also piggybacks [131] 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.[7] 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,

[7] 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.

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:

1. between modules, which yields slices;
2. between slices, which yields a language implementation.

Composition is also supported through bundles that behave just as languages, except for the fact that 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. 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). Please see [209] for a full Neverlang overview.

Neverlang supports LPL engineering thanks to AiDE [211, 210]. AiDE is a variability management tool tailored for the development of LPLs. It extracts information provided by Neverlang modules (lines 3-4 of Listing 2.1) to determine the language features and their dependencies and synthesizes the corresponding FM for a given language family [209]. Through its graphical user interface, the user can explore the FM, choose language features and create language configurations. 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. Based on the current configuration, AiDE generates the corresponding **`language`** descriptor automatically to ease the deployment phase.

2.4.11 Summary

Table 2.1 summarizes the contents of this section. For each of the discussed language workbenches, we report the key elements of each language workbench, including the composition mechanisms being used, their granularity and any IDE support. In particular, a language workbench marked with ● provides native IDE generation support, whereas ○ indicates that automatic IDE generation is not supported. Melange provides IDE generation support, but only through a third party tool (EMF). Finally, for each language workbench, Table 2.1 reports any relevant notes with regards to the flexibility of the composition mechanisms and techniques that can be used to bridge the gap between incompatible language components.

Workbench	Composition	Granularity	IDE generation	Notes
Xtext	Rule overriding	Syntax rule	●	Only supports single grammar inheritance
LISA	Multiple grammar inheritance and templates	Any syntactic and semantic elements	●	Due to templates, semantic rules are independent from grammar productions
Melange	Renaming	Ecore classes and aspects	EMF-based	Supports renaming of Ecore elements
MPS	Multiple inheritance	Concepts and behaviors	●	Behavior methods can compose incompatible concepts
JastAdd	Matching data structures	AST node with concrete syntax	○	Composing syntax and semantics requires the same signatures
MontiCore	Overriding and delegation	Grammar rule	●	Supports multiple grammar inheritance but cannot add attributes
Rascal	Multiple module inheritance	Any syntactic and semantic elements	●	Identifiers must match across unrelated modules
Spoofax	Rewrite rules and strategies	Any syntactic and semantic elements	●	Parse tree rewriting can compose incompatible strategies
Racket	Language macros	Any syntactic and semantic elements	●	Macros are rules that rewrite custom syntax into Racket expressions
Neverlang	Slices	Reference syntax and semantic action	●	Renames and mappings can compose incompatible modules

Table 2.1: *Comparison among language workbenches capabilities.*

3 Reconciling Object-Oriented and Feature-Oriented Software Design

Software product lines can be implemented with many different ways. However, most implementations can be classified into the same common underlying design philosophies: compositional and annotative. The *annotative* approach uses macros such as the `#ifdef` macro in C to highlight portions of a system intended to implement a software feature. The *compositional* approach uses variability-aware preprocessors called *composers* to generate a program variant from a set of *features* and a *configuration*. Both approaches have disadvantages. Most notably, these approaches are usually not supported by the base language; for instance Java is one of the most commonly used languages among researchers in the context of FOP, but it does not support macros and rather it relies on the C preprocessor or a custom one to translate macros into actual Java code. As a result, developers must struggle to keep up with the evolution of the base language, hindering the general applicability of SPL engineering. Moreover, to effectively evolve a software configuration and its features, their location must be known. The problem of recording and maintaining traceability information is considered expensive and error-prone and it is once again handled externally through dedicated modeling languages and tools.

One possibility to properly convey the FOP paradigm is to treat software features as first-class citizens using concepts that are proper to the host language, so that the variability can be expressed and analyzed with the same tools used to develop any other software in the same language. In this chapter, we overview a design pattern-based approach that fits these requirements without loss of generality. The proposed design pattern—dubbed *devise* pattern—can be used to express feature behaviors and constraints with a light-weight syntax similar to `#ifdef` macros. The same abstraction can be used to express both the modeling code and the implementation code and therefore suits both the domain analysis and at domain implementation activities.

3.1 The Problem of Conservation of Complexity in SPLs

Ideally, software product line engineering (SPLE) should provide variability mechanisms to accommodate the introduction and removal of crosscutting and non-crosscutting features, as well as their transformation without invasive changes and ripple effects. State-of-the-art SPL development environments—such as FeatureIDE [203, 150]—can

cope with all the aspects of the development of an SPL, including construction, management of software artifacts, configuration and product derivation. However, such tools and techniques are not natively supported by the base language and thus the developers have to struggle to keep up with the evolution of the base language—for instance, Java has a 6-month release cycle since March 2021. Moreover, there is no general consensus on how the composition mechanism should be performed, thus the source code of the core application and its features are structured differently depending on the chosen composer tool. Composer tools are preprocessors that translate feature-oriented code into Java code with regards to a chosen configuration. Possible composers are (among others) FeatureHouse [203], AHEAD [17], Antenna[1] and AspectJ [158]. However, it is usually possible to avoid using preprocessors thanks to the Java Virtual Machine (JVM) abstractions [76]. To change a composer is usually unfeasible as the SPL has to be rewritten. The tool chain may not support the new composer so the developers have to learn new syntax, tools and a specific development environment. A closely related problem is that of feature *traceability*: recording and maintaining the potentially scattered locations of features in the software artifacts for evolution and maintenance purposes is tedious and error-prone [1] especially when changes to the specification cause changes to the implementation and vice versa. While several feature location and variability mining strategies have been proposed [65, 183, 56] and evaluated [147] in the literature, they must be complemented by ad-hoc refactoring strategies to evolve software into a variability-aware SPL. These problems may obstacle the adoption of SPLs as a more wide-spread engineering technique [102] and solving them requires dealing with their inherent complexity. SPLE involves aspects of domain analysis and implementation, requirements analysis and product derivation; the possible configurations are exponential in the number of features and SPLs, e.g., the Linux kernel [194] has several thousands of features whereas the Neverlang.JS implementation of Javascript [33] has hundreds of language features.

As Larry Tesler stated in an interview for Dan Saffer [186]'s «Designing for Interaction» book: «Systems have an inherent amount of complexity that cannot be reduced». This is known as *the law of conservation of complexity* and leaves one question open with regards to complexity: if it cannot be reduced or hidden, then who should be exposed to such a complexity?

We present an approach in which managing the complexity of highly-variable software systems is a matter of *design* rather than a matter of *tooling*, in contrast to the typical approaches that largely focus on composer tools and preprocessors. In this approach, software features are modeled through concepts the software developers are familiar with, such as composition, inheritance and design patterns. Feature development and their recording are the same development activity, so that tracing is done with the same tools used to analyze normal code: Eclipse and JetBrains' IntelliJ IDEA, as well as most other modern Java IDEs support finding usages of classes and methods, and class hierarchy inspection and refactoring—including any external dependencies. Most developers are already familiar with these tools: using the same

[1] `http://antenna.sourceforge.net`

abstractions to implement both features and normal classes makes their expertise applicable to FOP at no additional cost. Should the development environment be changed, the same code can be reused with no changes. The same approach can be used as a refactoring framework to complement variability mining techniques or to avoid the feature location activity by explicitly declaring the variability points when an SPL is developed from scratch. To show how this change of perspective can impact the development of SPLs, we present a design pattern for FOP—dubbed *devise pattern*—and a variability-aware MNIST-encoder developed based according to a concrete implementation of this pattern.

3.2 Devise Pattern

To properly design SPL features as first-class citizens in traditional object-oriented software systems, we went back to Prehofer's seminal work [180, 181] and to the origin of FOP. According to Prehofer, FOP is a model for object-oriented programming which generalizes inheritance. Instead of using a rigid class structure, features are similar to mixins [24] and implement services that can be used by other objects. Therefore, objects behaviors are implemented by leveraging the aggregation of several features whereas more modern software composition tools such as FeatureHouse [9] give up aggregation in favor of superimposition—*i.e.*, the process of composing software artifacts by merging their substructures. In order to be able to design software features without any need for external tools and preprocessors, we stick to the original vision of FOP and take a more naïve approach, in which classes are the result of feature aggregation. This process is eased by changes recently introduced in object-oriented programming languages—such as lambdas in Java 8. We propose the *devise pattern*, to hopefully achieve the following FOP design goals:

- *separation of concerns*—the modeling code is separated from the implementation code;
- *light* on the domain analyst—the modeling code of a feature and of the FM is minimal, it contains no semantics and can be automatically generated if another representation of the FM is already available;
- *light* on the developers—the implementation code of a feature takes little to no boilerplate code (the same magnitude of a `#ifdef` macro in C);
- *flexibility*—the implementation code of feature behavior can either be embedded in the application or separated from it to support information hiding and reuse;
- *statically-checked*—both the modeling code and the implementation code are checked by the stock language compiler.

We discuss the *devise pattern*, including its participants and its application, following Gamma *et al.*'s work on design patterns. Therefore, this section follows the same structure used in the book «Design Patterns: Elements of Reusable Object-Oriented Software» [82]. The devise pattern is not the first attempt at developing SPLs using a design pattern-based approach. The variability modules in Java architectural pat-

tern [192] is based on variability modules and delta-oriented programming [187]: each feature is implemented using the Java modules and decorators [82] applied over the base feature. Seidl *et al.* [191] presented a generative SPL development method using variability-aware versions of the observer, strategy, template method and composite [82] patterns and introduced the Family Role Model as a notation to capture constraints on the variable application. In Apel *et al.*'s book «Feature-Oriented Software Product Lines» [8] an entire chapter is dedicated to «Classic, Language-Based Variability Mechanisms», ranging from traditional `if` statements to more sophisticated and flexible programming patterns to support variability. However, compared to other existing works, the devise pattern expresses variability in a standardized manner whereas other approaches the must be tailored to solve specific problems.

3.2.1 Purpose and Scope

The devise pattern is a *class behavioral* pattern [82]. It deals with the relationship among classes implementing crosscutting concerns (features) and with how these classes and their instances (feature actions) interact and set responsibility.

3.2.2 Intent

Explicitly express the variability points of an algorithm at source level so that they can later be traced and refactored. Keep the FM and its implementation aligned by means of the compiler. Plan the feature semantics ahead and defer their execution until they are ensured to be active in a valid configuration. Untangle feature-specific code from the main application. Render the main application unaware of the underlying configuration.

3.2.3 Motivation

Consider a variability-aware machine learning application in which two different loss functions can be used: Triplets [189] and InfoNCE [165][2]. These two specific loss functions are not interchangeable and choosing one over the other in a configuration affects the preparation of the training set and the graph of the model to be trained. In both cases, the code is scattered across the main application. The loss function is then a crosscutting concern and can be modeled as a feature. Different features may have constraints with each other: no loss function is needed if the model is restored from memory. Otherwise, either one of them must be active, but not both in the same configuration. At each point of the execution in which a configuration choice is relevant, the main application must explicitly declare a variability point and any dependencies among features which the variability point is concerned with. To summarize, to solve the problem of the variability of loss functions it means to solve four sub-problems:

1. *features declaration*—to declare the cross-cutting concerns;

[2]Please refer to Sect. 3.3.1 for additional background information on neural networks and loss functions.

2. *variability points declaration*—to declare a variability point in the application;
3. *constraints declaration*—to declare constraints among features;
4. *configuration management*—to configure product variants.

A solution to the *features declaration* problem is to separate the class hierarchy of the main application from the feature hierarchy, so that class instances (objects) and feature instances (feature actions) can be combined at will through aggregation. With this structure, cross-cutting concerns can be identified simply by inspecting the class hierarchy. In our example, each of the two loss functions will inherit from the same `Feature` abstract class. Any other class that does not inherit from `Feature` will not be identified as a cross-cutting concern. A common solution to the *variability points declaration* problem is the usage of conditional compilation with `#ifdef` macros [138]. While this solution is extremely simple, it is usually considered error-prone due to the low level of abstraction. To maintain the benefits of an `#ifdef` while improving the abstraction, a solution would be to separate the declaration of a variability point from its implementation. For example, both Triplets and InfoNCE are implemented in their own classes and the main method only declares the variability point in which one of the two must be chosen in a configuration. A common problem with *constraints declaration* is that feature constraints are usually declared at FM level, so it is hard to ensure that dependencies expressed in the source code align with those declared in the FM. A solution to this issue would be to declare the feature constraints directly at source level: the alignment between the representation at FM level and the implementation can then be checked automatically. For instance, the alternative nature between Triplets and InfoNCE that we discussed earlier should be expressed both at FM level and at source level. Any inconsistencies can be revealed by analyzing the source code against the FM. To solve the *configuration management* problem, the application needs an activation mechanism that handles the execution of each feature: feature actions must be executed if and only if the corresponding feature is active in the current configuration. In this example, the main application is a client for two possible services provided by the alternative Triplets and InfoNCE features. The service that is actually provided when the application is run is determined by a configuration, whose validity is checked against the FM.

3.2.4 Applicability

The devise pattern should be used to manage the variability of SPLs without preprocessors, as discussed in the motivational example above. In particular, the devise pattern can explicitly declare variability points in an application and untangle code from different concerns by refactoring them into features. The scattered locations of features implemented with the devise pattern can be retrieved automatically with common tools such as an IDE. The devise pattern can also be used to defer the execution of a block of code until the validity of a configuration is checked. Finally, the devise pattern offers a viable solution whenever configuration choices are subject to complex constraints.

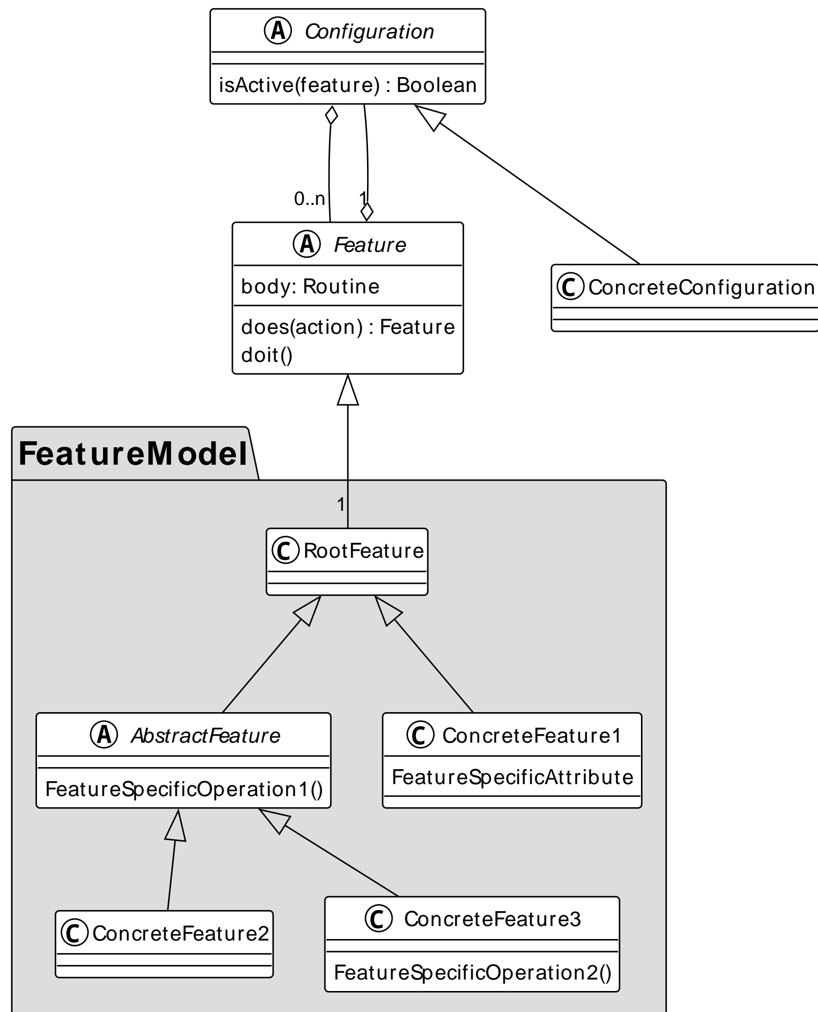


Figure 3.1: *Minimal class diagram of the devise pattern containing only its key elements without any additional abstraction.*

3.2.5 Structure and Participants

Fig. 3.1 shows the class diagram of an essential implementation of the devise pattern, representing the hierarchy of the feature classes. Each feature class is used by a client application (not shown in the diagram). There are five main participants to the devise pattern.

- *Feature*: the root of the feature hierarchy. Classes in the feature hierarchy represent the modeling code of the SPL whereas instances of those classes represent its implementation. The semantics of a feature action are devised using the `does` method and stored in a `body` field. The `does` method also returns the feature action to allow method call chains. The execution is deferred until the `doit` method is called.
- *Abstract and Concrete Features*: sub-classes of the `Feature` class determine the FM of the SPL. Each direct subclass of `Feature` (`RootFeature` in Fig. 3.1) is the root of a FM. The complete FM is equivalent to the sub-hierarchy of `RootFeature`, with abstract classes being abstract features and concrete classes being concrete features. Each feature can be enriched with feature-specific attributes (such as `FeatureSpecificAttribute` in Fig. 3.1) and operations (such as `FeatureSpecificOperation1` in Fig. 3.1). Notice that feature attributes are used to support the extended FM formalism and not as a means to implement any feature semantics since the feature hierarchy is only used as modeling code.
- *Configuration*: declares an interface to determine if features are active or inactive, *i.e.*, whether their devised action should be executed when its `doit` method is called.
- *ConcreteConfiguration*: implements the `isActive` interface. It stores the activation status of features, checks the validity of a variant and preempts the execution of inactive features.
- *Variant* (or main application): implements feature actions by creating instances of the feature classes, defines the variability points and the dependencies between feature actions.

3.2.6 Collaborations

Fig. 3.2 shows the sequence diagram of an exemplary variability-aware application implemented with the devise pattern. The participants are the same as in Fig. 3.1, with the addition of the `Variant` main application and a `PreMain`. The `main` method stored in the `Variant` is unaware of the current configuration which is set by the `PreMain`. In this example, `ConcreteFeature1` is inactive and `ConcreteFeature2` is active. Then, the `PreMain` launches the actual `Variant main`. The `Variant` declares two variability points, one for each of the two concrete features. In the case of `ConcreteFeature1`, the execution is devised and deferred to a later time whereas a feature action for `ConcreteFeature2` is devised and executed sequentially by calling the `does` and `doit` methods respectively. When the `doit` method is called, each feature action messages the

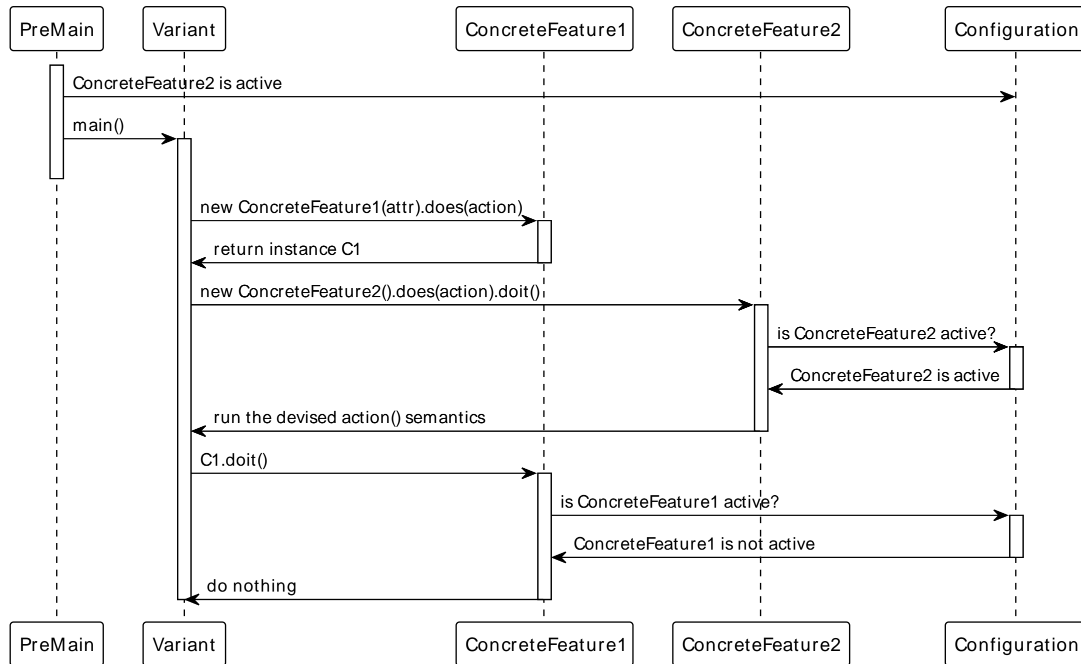


Figure 3.2: *Sequence diagram of an exemplary variability-aware application devising two feature actions: one instance of `ConcreteFeature1` (which is inactive) and one of `ConcreteFeature2` (which is active). In both cases, the execution is deferred until the `doit` method is called. `Configuration` preempts the execution of inactive feature actions (`ConcreteFeature1` in this example).*

`Configuration` to check if it is active. In this example, only `ConcreteFeature2` is active and therefore executed, whereas `ConcreteFeature1` is not executed. Notice that the `Variant` only has to devise the semantics of `ConcreteFeature1` and `ConcreteFeature2` whereas their execution or preemption is entirely handled by the `Configuration`.

3.2.7 Consequences

The devise pattern has the following benefits (+) and drawbacks (-).

+ It leads to an inverted control structure referred to as *the Hollywood principle* [201]: the `Configuration` handles the execution of the `Variant` and preempts the execution of inactive features and invalid configurations.
+ It makes the variability points of the application explicit: given a configuration, an active feature action could be replaced with its body without changing the semantics.
+ Feature actions are predictable and their body is a function that cannot cause side effects over variables in the scope.
+ Devising feature actions eliminates the need for conditional statements because alternative behaviors are selected based on the configuration; for instance, an

```java
new Hello().does(() ->
    System.out.println("Hello"))
.implies(new World().does(() ->
    System.out.println(" World")))
.doit();
```

Listing 3.1: *Declaring constraints among feature actions.*

alternative group is equivalent to a **`switch`** statement with a **`break`** on each case. For this reason, there is no need for constant flags in source code to control the execution flow.

+ Moves feature-specific attributes and methods from the classes' source code to the features' source code.
- Features increase the number of classes in an application: each feature is an additional class and the body of each feature action is translated into an anonymous class by the Java compiler.
- The code of the feature action's body is embedded in the main application, thus the resulting code may be hard to comprehend and analyze. As we will show later in this section, refactoring the implementation of a feature action out of the main application requires additional abstractions.

3.2.8 Implementation and Sample Code

The base pattern shown so far can be adapted to fit application-specific requirements by changing the interface described in Fig. 3.1 accordingly. We hereby discuss some implementation details that can improve the applicability of the devise pattern, as well as the corresponding changes made to the base model, as shown in Fig. 3.3.

Constraints among software features. In our implementation, we chose to enrich the `Feature` class with one method for each of the most commonly used Boolean relations, which are often used to define the propositional formulas that declare FM cross-tree constraints: `and`, `or`, `not`, `implies`, `xor`. An example of usage of this API is shown in Listing 3.1. In this example, if the `Hello` feature is active, then the `World` feature must also be active. This constraint can be expressed by using the `implies` method. To render this constraint possible, the devise pattern leverages the Java functional interface to defer the execution of all feature actions until the `doit` method is called (on line 5). Calling the `doit` method checks the validity of the current configuration with respect to the declared constraints before executing any of the semantics.

Embedded code and refactored code. In most cases it is beneficial to decouple the declaration of the variability point and its implementation, otherwise the devise pattern acts identically to *`#ifdef`* macros. In our implementation, we chose to provide an `@Action` annotation and an `isFeature` method that returns **`false`** if the class is

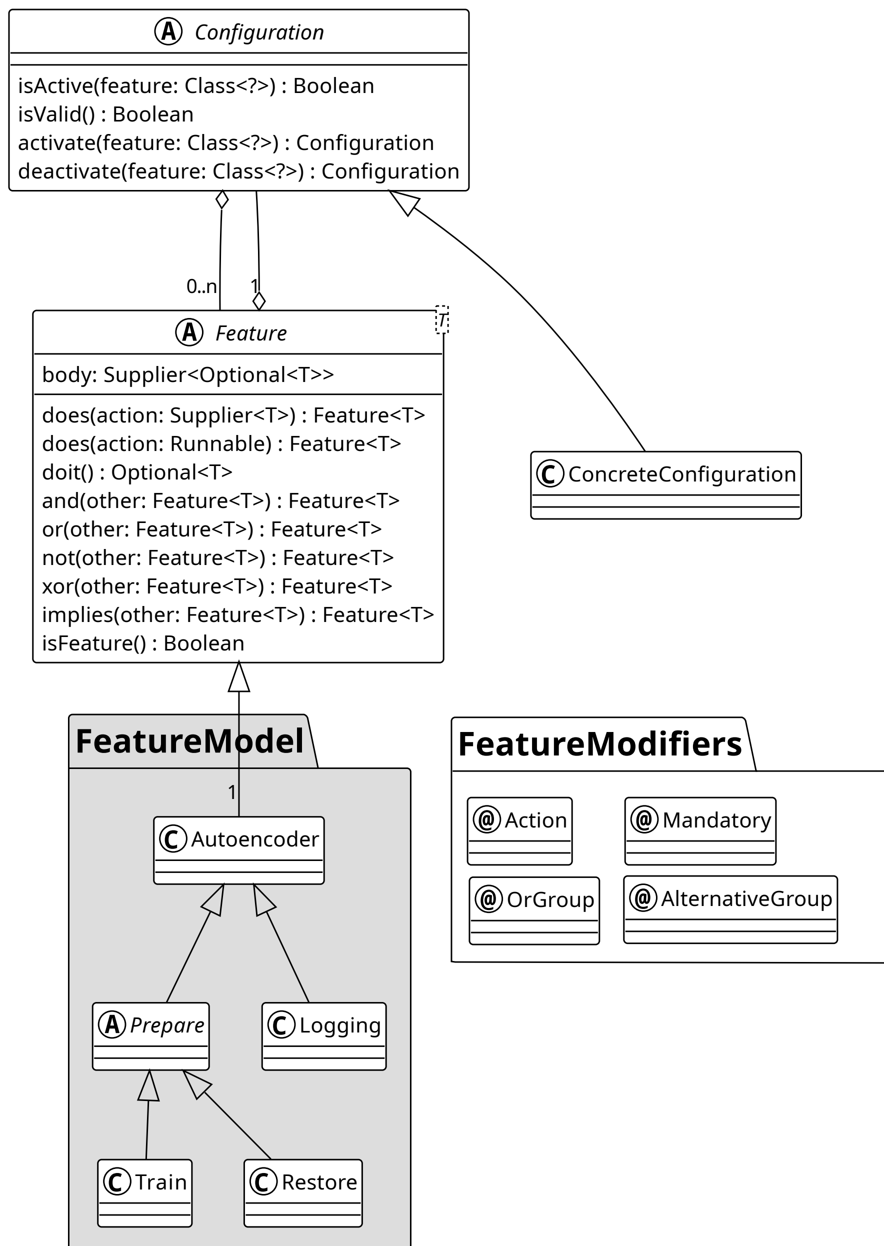


Figure 3.3: *Extended class diagram of the devise pattern, including part of the FM of a concrete variability-aware application.*

```
void main() {
  new Hello.does(() ->
    System.out.println("Hello")
  ).implies(
    new WorldAction()
  ).doit();
}
@Action
public class WorldAction extends World {
  public WorldAction() {
    this.does(() -> System.out.println(" World"))
  }
}
```

Listing 3.2: *Embedded and refactored feature actions.*

annotated, so that annotated classes are not considered as part of the FM and instead their activation status is determined based on their super-class. Consider refactoring Listing 3.1 so that the implementation of the `World` feature action is decoupled from the variability point declaration. The result of the refactoring is shown in Listing 3.2, in which the embedded feature action for the `World` feature was moved to the `WorldAction` annotated class. Thanks to this refactoring, the main method is unaware of the `World` feature implementation and the refactored `WorldAction` can also be reused in different parts of the application without code duplication. This was not possible in Listing 3.1.

Feature-specific class fields. A common application of feature-oriented programming is to add or to remove feature-specific fields from classes based on the product configuration. For instance, a product usually needs a logger only if the `Log` feature is active. It is usually impossible to remove fields from a class without using conditional compilation, although the devise pattern can be used to move feature-specific fields outside of the main application and within the most pertinent feature classes. Such an example is shown in Listing 3.3. The initial application (lines 1-14) does not use the devise pattern. Instead it contains a `PrintStream` field used to store the output channel for the logger and a boolean flag to determine whether any logging activity should be performed or not. The second application (lines 16-36) was refactored using the devise pattern. The `PrintStream` field was moved from the main application to the `Log` feature. The channel is then set and used in two separate feature actions (lines 25 and 32 respectively). The boolean flag is no longer needed since it is encapsulated in the configuration logic of the devise pattern.

Extended features parametrization. To implement extended features and their parameters, consider using configuration methods instead of constructors: non-default constructors must be overridden by child classes, causing unnecessary overhead for the domain analyst writing the model. In Listing 3.4, `Hello1` and `Hello2` are devised with the same semantics, but the second one does not require sub-classes to override the

```
1  //Base application
2  public class Main {
3      private PrintStream channel;
4      public static final boolean LOG = true;
5      public Main() {
6          if(LOG) channel = System.out;
7          //More initialization
8      }
9      public void routine() {
10         //DO STUFF
11         if(LOG) channel.println("Logging message");
12         //DO STUFF
13     }
14 }
16 //Application using the devise pattern
17 public class Log extends Feature {
18     public PrintStream channel;
20 }
21 public class Main {
22     private final Log logger = new Log();
23     public Main() {
24         logger.does(() ->
25             logger.channel = System.out
26         ).doit();
27         //More initialization
28     }
29     public void routine() {
30         //DO STUFF
31         logger.does(() ->
32             channel.println("Logging message")
33         ).doit();
34         //DO STUFF
35     }
36 }
```

Listing 3.3: *Moving feature-specific fields outside of the main application using the devise pattern.*

non-default constructor.

Non-void feature actions. Listing 3.1, 3.2, 3.3 and 3.4 show `void` feature actions, implemented using the `Runnable` interface. A more flexible implementation may allow feature actions to return values. In our implementation, the body of a feature action returns an `Optional` type. Feature actions can be devised by providing either a `Supplier` (with return value) or a `Runnable` (without return value) to the overloaded `does` method (see Fig. 3.3). In the latter case, executing the feature action will return an `Optional.empty()` value.

Passing the context to refactored feature actions. Code pertaining a specific feature may need some contextual information to be executed. Java's `Supplier` and `Runnable`

```java
void main() {
  new Hello1(42).doit();
  new Hello2().config(42).doit();
}
public class Hello1 extends Feature {
  private int param;
  public Hello1(int param) { //Must be overridden by subclasses!
    this.param = param;
    this.does(() ->
      System.out.println(param + " is the answer")
    );
  }
}
public class Hello2 extends Feature {
  private int param;
  public Hello2 config(int param) { //No need to override
    this.param = param;
    return this.does(() ->
      System.out.println(param + " is the answer")
    );
  }
}
```

Listing 3.4: *Configuring extended features.*

functional interfaces have access to the scope they are defined, *i.e.*, **this** object, its fields, its methods and any other variable in scope. Such an example is shown in Listing 3.5, in which the original application (lines 1-10) that uses a boolean flag to log the coordinates of a point is refactored using the devise pattern (lines 11-22), with a very similar syntax. However, **this** and any other contextual information is not accessible if the feature action is refactored outside of the main application with the `@Action` annotation. In these cases any contextual information that is relevant for the execution of the feature action should be passed to its constructor, as shown in the third version of the same application in Listing 3.5, lines 23-40. Such a refactoring does not only decouple the feature-specific code from the main application, but also allows the developers to tweak the amount of information that should be provided to a feature action.

3.2.9 Related Patterns

A builder [82] can greatly benefit from using the devise pattern to configure the creation of complex object variants. The `doit` method of the Feature class is structured as a template method [82]. The separation between the feature abstractions and their implementation through a `Runnable` or a `Supplier` functional interface is akin to a bridge pattern [82]. The enforcement of the same configuration across all features and the main application can be achieved with a singleton object [82]. The devise pattern can be used in conjunction with decorators [82] to implement the delta-oriented programming paradigm [187].

```
//Base application
public record Point3D(int x, int y, int z) {
    public Canvas canvas;
    public static final boolean LOG = true;
    public void plot() {
        canvas.plot(this);
        if(LOG) System.out.printf(
            "Point position: (%d, %d, %d)\n", this.x, this.y, this.z);
    }
}
//Application using the devise pattern
public class Log extends Feature {}
public record Point3D(int x, int y, int z) {
    public Canvas canvas;
    public void plot() {
        canvas.plot(this);
        new Log().does(() ->
            System.out.printf(
                "Point position: (%d, %d, %d)\n", this.x, this.y, this.z)
        ).doit();
    }
}
//Application using the devise pattern and @Action annotations
public class Log extends Feature {}
@Action
public class LogPointPosition extends Log {
    public LogPointPosition(Point point) {
        this.does(() ->
            System.out.printf(
                "Point position: (%d, %d, %d)\n", point.x, point.y, point.z)
        );
    }
}
public record Point3D(int x, int y, int z) {
    public Canvas canvas;
    public void plot() {
        canvas.plot(this);
        new LogPointPosition(this).doit();
    }
}
```

Listing 3.5: *Three different implementations of a logger for points in 3D.*

3.3 Case Study: MNIST-encoder

In this section, we will discuss a case study that compares three different implementations of a MNIST-encoder in which the variability is handled at source level without using external preprocessors: using JSON configuration files, the Variability Modules in Java (VMJ) [192] architectural pattern, and the devise pattern. Sect. 3.3.1 contains any relevant background information with regards to this case study.

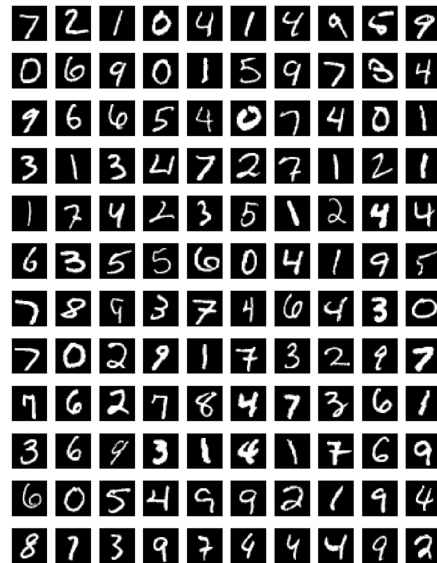


Figure 3.4: *A sample of 100 images taken from the MNIST dataset.*

3.3.1 Neural Networks and the MNIST Dataset

Neural Networks (NNs) are computational graphs with trainable parameters designed to solve specific tasks. When trained, these parameters are optimized so that a *loss* function is minimized. By minimizing the loss, the NN can learn specific patterns that are useful to solve the intended task. NNs are highly-customizable software systems: the type of the architecture and of the loss function, as well as the number of trainable parameters, are all factors that can substantially affect the NN performance. NNs learn hidden internal representations of the data on which they are trained. These representations can be explicitly trained to satisfy certain properties. For example, *contrastive learning* [43] is a technique in which a NN is trained so that semantically similar data points have close hidden representations. This can be achieved by means of a well-designed loss function. Contrastive learning can be both supervised and self-supervised. In the first case, semantically close data points are known in advance and the NN is trained so that their hidden representation is also close. For example, data points with the same label are trained to be close to each other, whereas data points with different labels are trained to be far apart. In the latter case, semantically close data points are not known beforehand and are instead generated using augmentation pipelines. These kinds of architectures are usually referred to as *encoders*.

We trained a set of NN encoders using contrastive learning on the popular MNIST dataset [136]. It contains $60,000$ gray scale images of 28×28 pixels. Each image represents a numerical digit from 0 to 9, as shown in Fig. 3.4. For this reason, we called this application MNIST-encoder. Figure 3.5 schematizes an execution of the model, in which an image from the MNIST dataset is fed to a convolutional neural network (CNN) model to obtain an embedding in the output space.

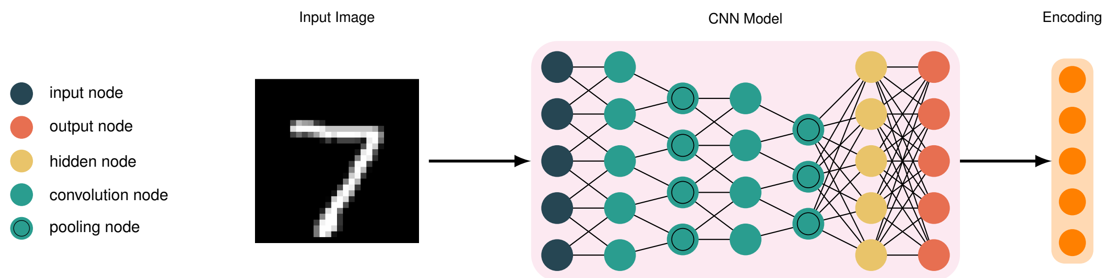


Figure 3.5: *Process for turning an image from the MNIST dataset into an encoding in the output space.*

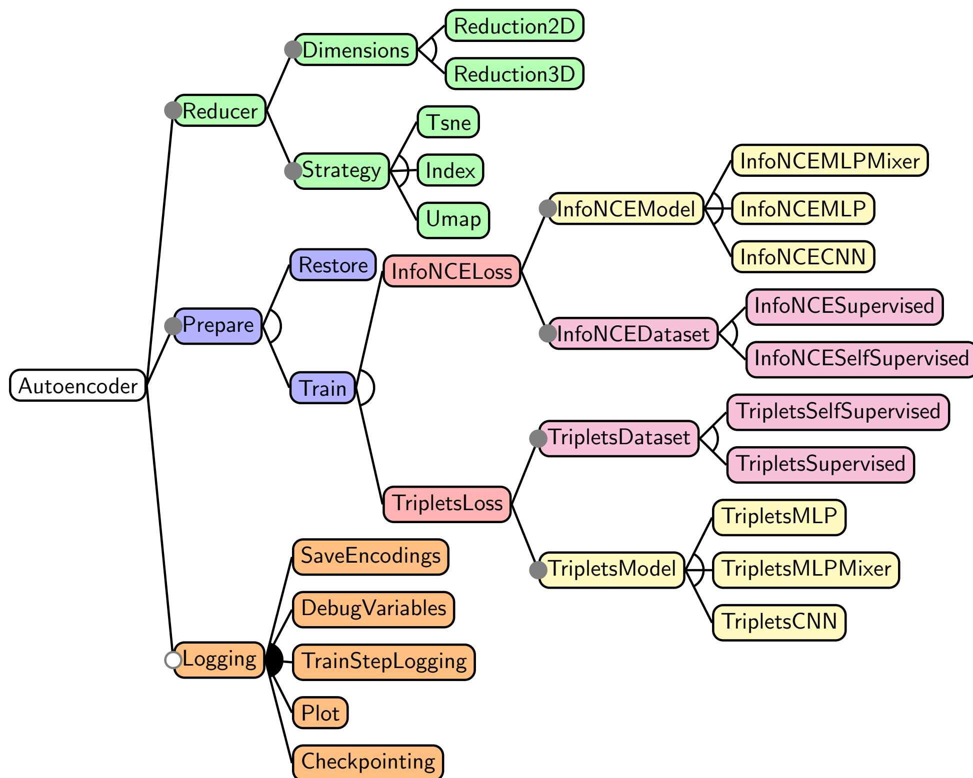
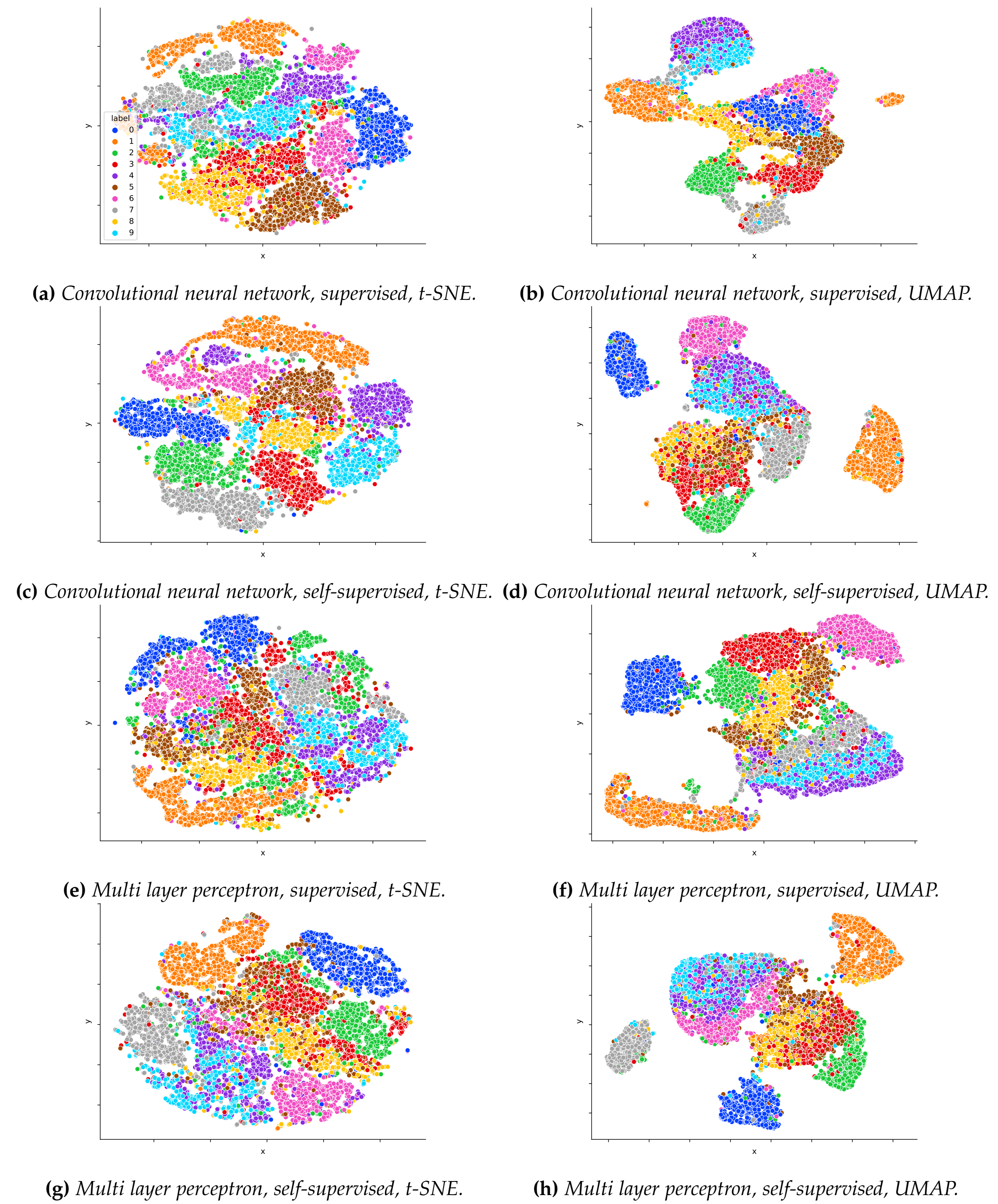


Figure 3.6: *FM of the MNIST-encoder software family. Each feature was colored based on the variability concern it addresses.*



(a) *Convolutional neural network, supervised, t-SNE.*

(b) *Convolutional neural network, supervised, UMAP.*

(c) *Convolutional neural network, self-supervised, t-SNE.*

(d) *Convolutional neural network, self-supervised, UMAP.*

(e) *Multi layer perceptron, supervised, t-SNE.*

(f) *Multi layer perceptron, supervised, UMAP.*

(g) *Multi layer perceptron, self-supervised, t-SNE.*

(h) *Multi layer perceptron, self-supervised, UMAP.*

Figure 3.7: *Embeddings obtained by eight variants of the* MNIST-encoder *using the InfoNCE loss. The legend reported in Fig. 3.7a maps each color to the label of the 10,000 data points.*

3.3.2 Application Overview

The design of deep learning applications often offers huge challenges in terms of variability. Several aspects of NNs, including architecture, training procedure and dataset, can be modeled differently to achieve different results. SPLE represents a valuable asset to model the variability of this kind of applications and to produce a family of related but different NNs. In this work, we embraced this approach to develop a family of MNIST-encoders. First, we analyzed the application domain and produced the FM previously shown in Fig. 2.1. Hereby, we report a version of the same FM highlighting the several variability concerns of the MNIST-encoder SPL, each represented by a different color.

- *Logging* (orange): tweak the output information that is provided to the user during training, including the value of debug variables, the loss and the model checkpoints; logging can also optionally plot the resulting encodings.
- *NN architecture* (yellow): we consider three kinds of architectures. The multi-layer perceptron (MLP) [87], the convolutional neural network (CNN) [87] and the MLP mixer [206]. All these NNs can be trained according to different loss functions. We consider only the Triplets [189] and the InfoNCE [165] loss functions.
- *Supervised* or *self-supervised* learning (pink): depending on the approach, a different dataset has to be generated.
- *Loss function* (red): the model creation and its training differ based on the chosen loss function.
- *Dimensionality reduction techniques* (green): usually, hidden NN representations are high dimensional vectors. To visualize these vectors in 2D scatter plots, it is necessary to project them into a low-dimensional space. This can be achieved with various techniques such as t-SNE [143] and UMAP [149]. Instead, when the hidden representation is already low dimensional, it can directly be plotted without projection.
- *Training mode* (blue): determines whether the model is loaded from memory or trained from scratch.

Overall, the FM contains 34 features, 15 of which are extended features that can be parameterized, allowing for additional customization options. We considered and evaluated 8 out of the total 55,296 valid configurations. Each variant was trained and used to produce the encodings of 10,000 data points from the MNIST dataset. We chose eight specific variants for a better comparison: we kept a shared base configuration and only changed a few features to better showcase the effect that each feature has on the results. The results are shown in Fig. 3.7. Each color represents a data point labeled with a different digit (from 0 to 9). All variants learned a meaningful representation: data points with the same label are generally clustered together. The first two rows are NN variants using the CNN architecture; the last two rows are NN variants using the MLP architecture. Odd rows use a dataset for supervised learning whereas even rows use a dataset for self-supervised learning. The left column shows NN variants in which dimensionality reduction is performed using t-SNE. On the right column those

in which dimensionality reduction is performed using UMAP. Notice that the CNN variants show better clustering on average.

3.3.3 Variability-aware Encoders

In this section, we overview the three approaches that we chose to turn the base MNIST-encoder implementation into an SPL that is aware of the variability concerns discussed earlier. Since we are focusing on approaches that do not require external tools, the configuration is performed manually by the developer in all these implementations. However, consider that the generation of configuration files can be automated with additional tooling.

JSON. The Javascript object notation (JSON) is commonly used for serialization and deserialization of objects; it is also used as a format for configuration files. In this version of the MNIST-encoder, the JSON configuration files are deserialized into factories [82]. The MNIST-encoder can be customized by editing one or more of the configuration files: a different JSON configuration will instantiate a different factory and eventually a different variant.

VMJ Pattern. VMJ [192] is an architectural pattern for the generation of SPLs. VMJ is based on the DOP paradigm in which features are expressed as deltas over a core module. Each delta is implemented as a decorator [82]. In VMJ a product is expressed using factories that instantiate a core module and all the required deltas depending on the configuration. Feature selection happens in a module declaration that lists all its requirements. Configurations are expressed as different main methods in which the core modules are configured by applying all the necessary deltas. Please refer to [192] for a complete overview.

Devise Pattern. The MNIST-encoder implementation based on the devise pattern follows the framework discussed in Sect. 3.2. Each feature in the FM from Fig. 3.6 is a Java class that directly or indirectly inherits from `Feature`. The effects that the activation of each feature has on a variant are expressed as feature actions—*i.e.*, instances of feature classes—whose semantics are devised by passing executable code to the `does` method. For instance, Listing 3.6 shows two features: `Restore` and `Train`, which are part of an alternative group—only one of them can be active at the same time. If both are active or both are inactive, the configuration is considered invalid. This is expressed at source level using the `xor` method. When `Train` is active the differentiation engine is created as a clear instance. Instead, when `Restore` is active the differentiation engine is instantiated by loading a previously saved model. In this case, the semantics are devised using Java method references. In the comments on lines 2, 4 and 6 of Listing 3.6, the preprocessor annotations that would be used to achieve the same result without using the devise pattern. Instead, with the devise pattern, creating a new configuration should be as effortless and reusable as possible. The MNIST-encoder uses

```java
SameDiff engine =
    new Restore<>().does(      // #ifdef RESTORE
        Model::load
    ).xor(new Train<>().does( // #elif defined(TRAIN)
        SameDiff::create
    )).doit();                 // #endif
```

Listing 3.6: *`Restore` and `Train` are alternative features.*

```java
public class DerivedConfiguration extends BaseConfiguration {
    public DerivedConfiguration() {
        super();
        this.activate(
            InfoNCELoss.class,
            InfoNCEDataset.class,
            InfoNCEModel.class,
            Reduction2D.class,
            Tsne.class,
            InfoNCEMLP.class,
            InfoNCESupervised.class);
    }
}
```

Listing 3.7: *Creating a `DerivedConfiguration` is eased by extending the `BaseConfiguration`.*

a `BaseConfiguration` class as a template for all eight aforementioned NN variants and leverages inheritance to minimize the required changes. For instance, Listing 3.7 shows a `DerivedConfiguration` which is obtained by activating seven additional features over the `BaseConfiguration`. The effort of creating a new configuration is minimized by sticking to a declarative approach in which active and inactive features are simply listed with no mention of the control flow of the application. This implementation is based on the extended version of the devise pattern interface shown in Fig. 3.3, supporting all the modeling techniques provided by mainstream feature modeling tools:

- cross-tree constraints are expressed using Boolean operators over feature actions (`and`, `or`, `not`, `xor` and `implies` methods);
- alternative (`xor`) groups are expressed by an `@AlternativeGroup` annotation as a feature class modifier;
- or groups are expressed by an `@OrGroup` annotation as a feature class modifier;
- mandatory features are expressed by a `@Mandatory` annotation as a feature class modifier.

The `BaseConfiguration` class collects all this information with regards to each feature class in the feature hierarchy and evaluates its validity before running the main application. The execution is preempted if the configuration is invalid with respect to the FM. The full implementation of the devise pattern and its application is available at

Zenodo[3].

Summary. In JSON the variability is handled using configuration files and factories. In VMJ the variability is handled by applying different decorators over the base component class. In the devise pattern the variability is handled by declaring variability points and devising feature actions and managed by a configuration class.

3.3.4 Comparison: Non-Functional Properties

The semantics of each of the eight considered variants of the MNIST-encoder do not change depending on the mechanism used to express the variability: JSON, VMJ or devise pattern. However, the three approaches are substantially different with regards to their non-functional properties. In this evaluation, we identified 13 non-functional properties supported by at least one of the approaches. Then, we classified each non-functional property into one of four categories. The *feature dependencies* category collects all properties dealing with the expressiveness with regards to the base FM formalism: alternative groups, or groups, mandatory features and cross-tree constraints. The *implementation extension* category collects the properties dealing with the capability of changing the behavior of an existing class [192]: adding and removing fields and methods. The *Other FM formalisms* category collects properties dealing with the expressiveness with regards to variants of the base FM formalism: the extended FM formalism and the multi-dimensional FM. The *Quality of life* category collects all other properties that can improve the usability of the approach by providing support to the verification and maintenance of SPLs: static checking capabilities, configuration inheritance, traceability of feature location, separation between modeling code and implementation code, and compatibility with other approaches. This section discusses each of the 13 properties. Table 3.1 summarizes this discussion.

Feature Dependencies. In most common variability modeling frameworks, such as FeatureIDE, the FM formalism can be properly expressed by enriching features with additional information—*i.e.*, if features are either *optional* or *mandatory* and if siblings are part of an *alternative group* or an *or group*. JSON cannot express any of these feature dependencies. VMJ can properly support *mandatory* features by combining module requirements and well-designed factories. However, to the best of our knowledge, it is not possible to declare different deltas as part of an *or group* or an *alternative group*, because each delta is modeled as a decorator over the same base component class. In our implementation of the devise pattern, *mandatory* features, *alternative groups* and *or groups* are expressed as simple annotations and checked by the configuration abstraction. Instead, the devise pattern supports cross-tree constraints only partially, because feature dependencies are expressed at source level and evaluated when the feature action is instantiated; therefore any invalidity with regards to cross-tree constraints is captured,

[3] https://doi.org/10.5281/zenodo.6624848

	Property	JSON	VMJ	Devise
Feature dependencies	Alternative groups	○	○	●
	Or groups	○	○	●
	Mandatory features	○	●	●
	Cross-tree constraints	○	○	◑
Implementation extension	Add fields & methods	○	●	◑
	Remove fields & methods	○	●	○
FM formalisms	Extended features	●	●	●
	Multi-SPL	◑	●	◑
Quality of life	Statically checked	○	◑	●
	Configuration inheritance	○	◑	●
	Traceability support	○	●	●
	Model and implementation independence	○	○	●
	Intercompatibility	●	●	●

Table 3.1: *Support to VM modeling in different approaches.*
○: not supported, ◑: partially supported, ●: fully supported.

but only at runtime. Statically detecting cross-tree constraints would require an external control flow analysis tool.

Extension of a Base Implementation. The VMJ approach natively supports addition and removal of both fields and methods. Addition is simply performed by decorators. Removal of methods is done by throwing a runtime exception in the overridden method. Similarly, removal of fields is done by overriding their getters and setters. Notice that this is possible only if the removed fields are private and never accessed through reflection. The devise pattern can emulate the addition of fields and methods through aggregation (as shown in Listing 3.3) and does not support fields and methods removal. However, according to the *additive universe conjecture* «every FM that uses subtractive features can be transformed to a new FM that uses only additive features; the two FMs share the same set of products» [15]. The additive universe conjecture was later proved with the goal of achieving monotonic reasoning on delta-oriented software product lines [58]. JSON does not support any of the above functionalities.

Other FM Formalisms. All three approaches fully or partially support the variants of the base FM formalism: extended feature model and multi-product lines. The parameters of extended features are implemented as fields but are handled differently depending on the approach: in JSON, the fields are simply added to the factories and are deserialized when the configuration is loaded; in VMJ, the parameters are added as fields of the decorator classes and then set by the factory methods called by the main; in the devise pattern, parameters are static fields of the classes from the feature hierarchy. Multi-product lines can be partially achieved in JSON by nesting configurations and in the devise pattern by using multiple configuration classes in the

Modeling effort	JSON	VMJ	Devise
Variability	597	1528	1066
Configuration (1 configuration)	43	112	109
Configuration (8 configurations)	344	896	293

Table 3.2: *Modeling effort in terms of LoC required to turn a core implementation into a variability-aware one using different approaches. Also the effort to create the first configuration and all the eight configurations from Fig. 3.7.*

same product which are combined through aggregation; only VMJ directly addresses the problem of multi-product lines and it is designed to fully support the formalism.

Quality of Life. Finally, we consider the aspect of quality of life for variability modeling. This includes the capability of statically checking SPLs and their configurations, the support to the extension of existing configurations, the traceability of feature implementations and the compatibility with other approaches. In the devise pattern all elements of variability are statically checked: *mandatory* features, *or* and *alternative groups*; all product variants coexist and are checked by the compiler, including any return types of the feature actions. VMJ can only check the validity of *mandatory* features using modules. However, decorators over the base module are applied using reflection that can fail at runtime if types mismatch. Similarly, access to removed fields and methods raise runtime exceptions that can cause failures if not properly handled. Both the devise pattern and VMJ support the extension of existing configurations. However, this can be achieved natively with the devise pattern using the `Configuration` class, whereas changing a configuration in VMJ requires refactoring of the existing product. Most notably, removing a delta may not be feasible depending on the ability to remove decorators from a component. Finally, in both VMJ and the devise pattern the feature traceability issue is trivialized by mapping features and their implementations to language-specific abstractions which usages can be easily traced by any commonly used IDE. However, only the devise pattern can properly separate the model from its implementation, whereas in VMJ the deltas are expressed in the decorator classes together with their implementation. None of the above quality of life improvements are directly supported in JSON. However, it should be noted that the three approaches to variability modeling are not mutually incompatible and can be combined at will depending on the scenario to stem and complement their issues. We argue that this point represents the main advantage of using an in-language approach to variability modeling over external preprocessors: software artifacts that can be handled by a specific preprocessor are usually incompatible other preprocessors, therefore migration between different preprocessors can be hard or unfeasible.

3.3.5 Comparison: Modeling Effort

For each of the three implementations of the MNIST-encoder, we analyzed the effort—in terms of lines of code (LoC)—required to model the variability and generate new products. All implementations depend on a variability-unaware core application of 2236 LoC. Table 3.2 reports the results of our analysis.

– Modeling the variability in JSON required the factory classes discussed earlier, for a total of 597 LoC; then each configuration can be written in JSON for a total of 43 LoC for each configuration. Deploying the eight configurations from Fig. 3.7 costs 344 LoC.
– Modeling the variability in VMJ required adding the component and decorator classes, as well as the same factories used by JSON, for a total of 1528 LoC; then each configuration can be written in a Java main class for a total of 112 LoC for each configuration. Deploying the eight configurations shown in Fig. 3.7 costs 896 LoC.
– Modeling the variability using the devise pattern required adding one class for each feature of the FM and a variability-aware main class, for a total of 1066 LoC; writing the first configuration requires creating the `BaseConfiguration` abstract class (86 LoC) and a concrete configuration subclass (23 LoC), for a total of 109 LoC. Deploying the remaining 7 configurations shown in Fig. 3.7 costs 23 additional LoC for each configuration: the total configuration effort is 293 LoC.

The modeling effort among the three approaches is fairly similar: JSON has a slight advantage in terms of both variability modeling and configuration modeling, but it should be noted that the JSON approach does not express the FM formalism and its constraints. Conversely, VMJ and the devise pattern face an initial overhead to introduce the variability with the advantage of expressing feature constraints. Between the two, each feature written in the devise pattern takes slightly less LoC than the respective feature in VMJ, for a total of 1066 LoC vs 1528 LoC. Finally, configuration inheritance in the devise pattern introduces an initial overhead on the first configuration but then the first configuration can be reused to model subsequent configurations, reducing the configuration effort substantially in the long run.

3.3.6 Threats to Validity

The validity of our results may be threatened by our lack of expertise with the VMJ architectural pattern; our implementation was not reviewed by the original authors [192] and we may have applied the pattern incorrectly, which may lead to different results in Table 3.2. To stem this issue we applied the same black-box approach to the usage of the devise pattern: the implementation of the devise pattern library and its usage for the development of the MNIST-encoder were performed separately by different contributors of this work. To the best of our knowledge the three variability-aware implementations of the MNIST-encoder should be semantically equivalent, but this is hard to properly

verify due to the random nature of the learning process in DeepLearning4J[4]—different runs may result in different NNs. In this regard, we separated the core library that implements most of the functionalities used by each implementation, so that each approach is only concerned with the variability modeling aspect. Our implementation of the devise pattern is written in Java and may not be applicable to other languages. However, the pattern should at least be applicable to any object-oriented language, as we tested by developing minimal implementations in Scala and Kotlin[5]. We do not compare against composers and preprocessors from the literature—they may provide better abstractions for variability modeling. However, a direct comparison may not be applicable because the two approaches to variability modeling tackle different problems.

3.4 Summary of Chapter 3

In this chapter, we discussed the design and implementation of feature-oriented SPLs and introduced the devise pattern as a technique for modeling and implementing SPLs. The devise pattern can express all aspects of the extended FM formalism using tools that are familiar to software developers and with a syntax similar to the `#ifdef` macros in C. Feature actions can also be refactored using dedicated abstractions to avoid the `#ifdef` hell problem and code duplication. The devise pattern prevents the need for feature location techniques because variability points are explicitly declared in the application and can be used to complement variability mining techniques. The pattern is described following Gamma *et al.*'s template. We demonstrated its applicability on the development of a variability-aware MNIST-encoder application. Finally, we compared this application to other two variability-aware alternatives implemented using JSON configurations and VMJ decorators respectively. Our contribution resulted more expressive and can model all aspects of the FM formalism; the validity of the configurations can be checked statically and deploying multiple configurations is eased. When compared to traditional annotative and compositional approaches to the development of SPLs, the devise pattern does not require any additional tools nor expertise and can therefore ease the adoption of SPLs in the industry by reducing the barrier to enter associated with the complexity of dedicated tools and environments.

In the following chapters we will deviate from the topic of traditional SPLs and focus on the niche of LPLs. Although many design and implementation aspects overlap, we will see that LPLs have to face very different problems from standard SPLs; most aspects of the design methodology will need to be adapted to be applicable to the context of compilers and interpreters.

[4] `https://deeplearning4j.konduit.ai/`
[5] The Scala and Kotlin implementations are not discussed in this dissertation.

4 A Design Methodology for Language Product Lines

LPLs and language workbenches enable a flexible approach to language development, in which monolithic implementations are replaced by reusable language assets. 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 highly 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 an LPL. In this chapter, we try to capture these aspects by discussing a design methodology for LPLs. We define the properties of a good language decomposition and the metrics that can be used to assess these properties with regards to language components, as well as the engineering process and the tools supporting the development of LPLs.

4.1 The Problem of Design Quality in LPLs

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 [72]. 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 a 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 [169, 208, 144, 74, 134]. 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 [216] and the automatic generation

of IDE services and debugging [28, 131, 72]. 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 [52]. In language engineering, separate constructs should be either independent or implemented as a unique feature. Yet, we argue that such a 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. In this context, we define the qualities of a good language decomposition developed following that process. The output of this work 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 an LPL engineering process to guide the design of language decompositions.

The contribution presented in this chapter is validated by answering the following research questions:

$RQ_{4.1}$ What are the properties of a language decomposition in Neverlang?
$RQ_{4.2}$ 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.

4.2 LPL Design Methodology in "Five" Steps

We hereby describe the concepts that enable the development and evaluation of well designed Neverlang-based language families. This topic deals with several design aspects:

1. the bottom-up LPL engineering process;
2. the properties of a well designed language decomposition;
3. the metrics to measure the quality of LPLs;
4. a design methodology that encompasses all other aspects, as well as a dedicated IDE, under a unified vision by following Parnas' steps.

According to Parnas [169], a design methodology for any software system should account for five design aspects of that system:

1. the order in which decisions are made;

2. what constitutes good structure for a system;
3. methods of detecting errors in design decisions;
4. specification techniques;
5. tools for system designers.

Notice that the *system* (points 2 and 5) is an LPL in our case. Point 1 can be addressed by an engineering process. Point 5 is addressed by a suitable LPL engineering environment. 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. In this dissertation, we do not discuss the topic of specification techniques (point 4) for Neverlang LPLs, which should be tackled on with a different approach in future works.

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 (Sect. 4.2.1), followed by the LPL engineering environment (Sect. 4.2.2), the properties of a well designed language decomposition (Sect. 4.2.3) and finally the metrics for the detection of design errors in LPLs (Sect. 4.2.4).

4.2.1 The LPL Engineering Process—Point 1

The LPL engineering process establishes the order in which decisions are made—*i.e.*, point 1 of Parnas' vision. Although the general SPL engineering process presented in [177] and illustrated in [150] also applies for LPLs, it is still too coarse-grained to disclose the relevant users and views of an LPL. Similarly, the LPL development process, described in [129], is not detailed enough as it neglects the language user. The *Business Process Model and Notation* (BPMN) model[1] [48] in Fig. 4.1 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. Moreover, it shows the artifacts created or refined by each task. Distributing the engineering process over several areas of responsibility allows for the concurrent development of LPLs while minimizing conflicts. Here, only the language components, the FM and the language variants are highlighted, because they are shared between several tasks of different roles, therefore they must be kept consistent between the views of the three roles.

The process tries to grant continuous flexibility and responsiveness in the development and extension of LPLs while maximizing the separation of concerns among

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

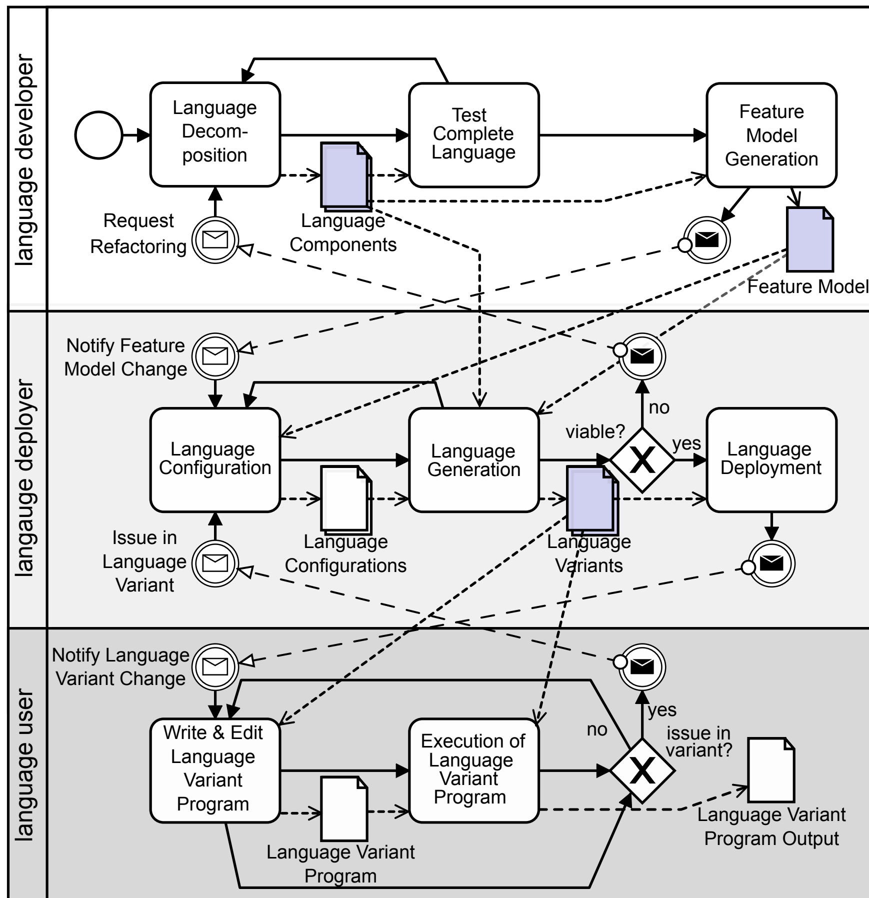


Figure 4.1: *BPMN model describing the language product line engineering process.*

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.

Language Development. 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 [129], the language decomposition triggers the generation of a description of the variability space of the system, which is typically 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.

Language Deployment. The *language deployer* (second layer) is notified about the latest FM to choose and pick [130] language features from the FM, thus creating or revising a *language configuration*. The deployer incrementally generates language variants and tests their viability, *i.e.*, whether it includes the desired language constructs and concepts. 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* [224], *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. 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 [131], e.g., an editor with syntax highlighting, code completion and a debugger, making them available to the language users.

Language Usage. 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.

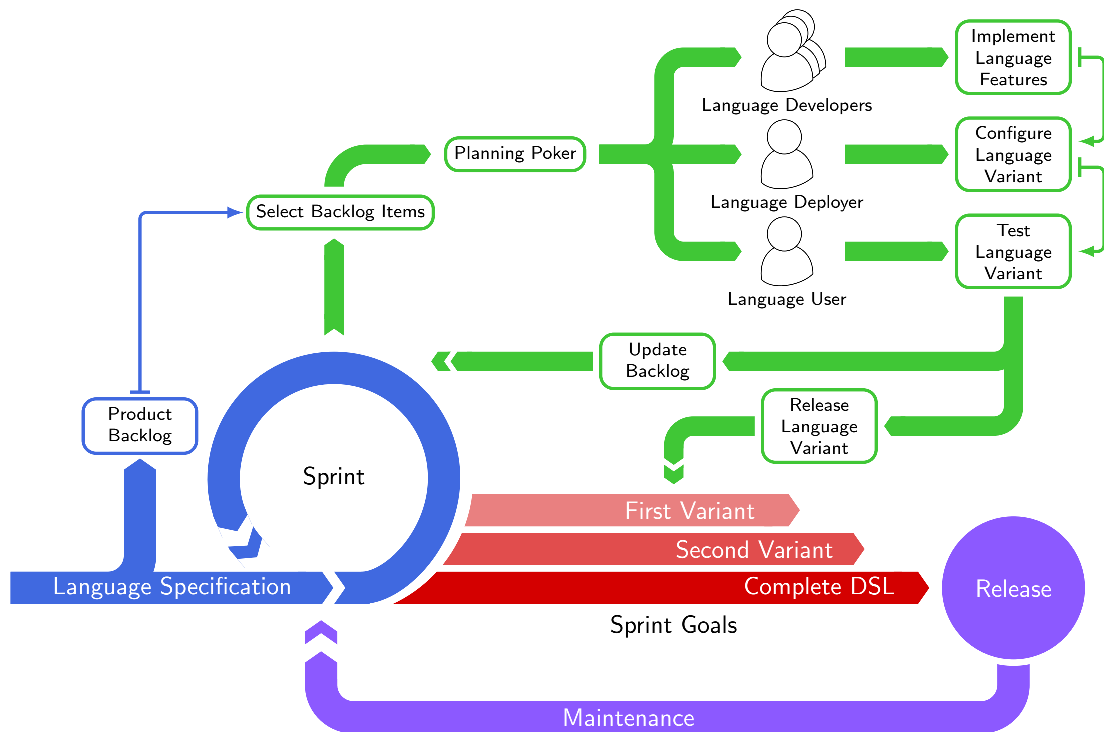


Figure 4.2: *Agile LPL development applying the engineering process on each sprint.*

Agile LPL Development. The 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. 4.1 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 overlap 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:

1. requests from *language user* to *language deployer* and from *language deployer* to *language developer* when the BPMN model is traversed from bottom to top;
2. 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.

Due to its iterative nature, the engineering process presented in this section is suited to be part of an agile language product line development process, as shown in Fig. 4.2. We will not discuss all aspects of agile development frameworks in this dissertation,

although, each iteration of the engineering process as detailed in Fig. 4.1 can be viewed as the sprint of an agile framework such as SCRUM[2]. Following this approach, the development team can set a language variant as the *sprint goal* of the current sprint, so that each iteration releases a new interpreter or a compiler. The result is an LPL comprised of all the language variants released on each sprint.

The proposed LPLE process, both in its base version and in agile context, 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.

4.2.2 The LPL Engineering Environment—Point 5

Each role of the engineering process presented in Sect. 4.2.1 requires very different views and services that must be provided by a proper LPL development environment. Granted, it may be challenging to implement all these views in one development environment, yet this enables distributed, incremental development of LPLs with tight feedback loops and rapid deployment, whereas the LPL development environment maintains the consistency between the shared artifacts, *i.e.*, the language components, the FM, and language variants. With tooling support, the engineering process can address any conflicts in the requirements: requests from different language users can be balanced by configuring additional language variants, while feature conflicts are translated into FM constraints by the environment. To summarize, each and every phase of the engineering process must be supported by a dedicated *tool for system designers*, according to point 5 of Parnas' vision.

Our approach is based on the combination of the state-of-the-art in SPL and LPL engineering. Instead of implementing the tools for system designers from scratch, we opted to create the novel AiDE 2 LPL development environment by marrying two established frameworks: Neverlang and AiDE for LPL development and FeatureIDE for feature-oriented SPL development. 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. For instance, Neverlang itself is bootstrapped and its IDE is generated within AiDE 2 using **`categories`**, **`in-buckets`** and **`out-buckets`** as in Listing 2.1.

Integrating AiDE into FeatureIDE. FeatureIDE and AiDE are standalone development environments for the development of product lines. AiDE 2 is an integrated LPL development environment born by marrying them. The FeatureIDE core library was used to implement the *layered language feature model* as an extension of the default FeatureIDE FM class and a set of additional abstractions to represent the different syntactic and semantic features of an LPL, as well as the cross-tree constraints. The IDE

[2] `https://www.scrum.org/`

was implemented as an Eclipse plugin and provides several extension to FeatureIDE and to the native Eclipse environment:

1. the AiDE project nature for LPL projects;
2. a Neverlang incremental builder for the AiDE project nature;
3. the AiDE composer for the creation of language artifacts from configuration files;
4. wizards for the creation of new LPL projects and language variants extending the *New Feature Project Wizard* and the *New Configuration Wizard* respectively;
5. the *Neverlang Configuration Editor* extending the FeatureIDE *Configuration Editor* for the deployment of language variants.

In addition, we updated the *Neverlang Editor* introduced in [131] to support automatic and dynamic reloading of language variants and their editors.

AiDE 2. The bottom-up generation of a FM from the source code of language components is based on the algorithm presented in [130]. The novel AiDE 2 algorithm extends the original algorithm to expand the FM one level deeper; the goal is distinguishing between syntactic and semantic language features. Algorithm 1 accepts the FM generated by the original AiDE algorithm from [130] as an input. First, only abstract features are present, then our extension creates the corresponding syntactic features as leaves. Finally, all semantic actions, attached to a syntactic feature are added as their leaves. This enables a more fine-grained customization of languages, as it enlarges the variant space. For instance, it is possible to generate a syntax checker with no semantics by only selecting syntactic features from the FM during the configuration process.
Fig. 4.3 shows the FM for LogLang generated by AiDE 2. It contains abstract features generated from the tags defined in `modules` (Listing 2.1 lines 3-4) and two layers of concrete features. The first holds the syntactic features of the LPL, whereas the second contains the corresponding compatible semantic features. To seamlessly support the LPL engineering process, the AiDE FM generation algorithm has been integrated with

Algorithm 1: ExpandFeatureModel (FM: Feature Model)

```
begin
    P := {p | p is a node in FM};
    for p ∈ P do
        S := {f | f ∈ syntactic_features(p)};
        for f ∈ S do
            generate concrete syntactic node n for f;
            children(n) := ∅;
            R := {f′ | f′is a semantic feature compatible with f};
            for f′ ∈ R do
                generate concrete semantic node n′ for f′;
                children(n) := children(n) ∪ {n′};
            end
            children(p) := children(p) ∪ {n};
        end
    end
    return FM;
end
```

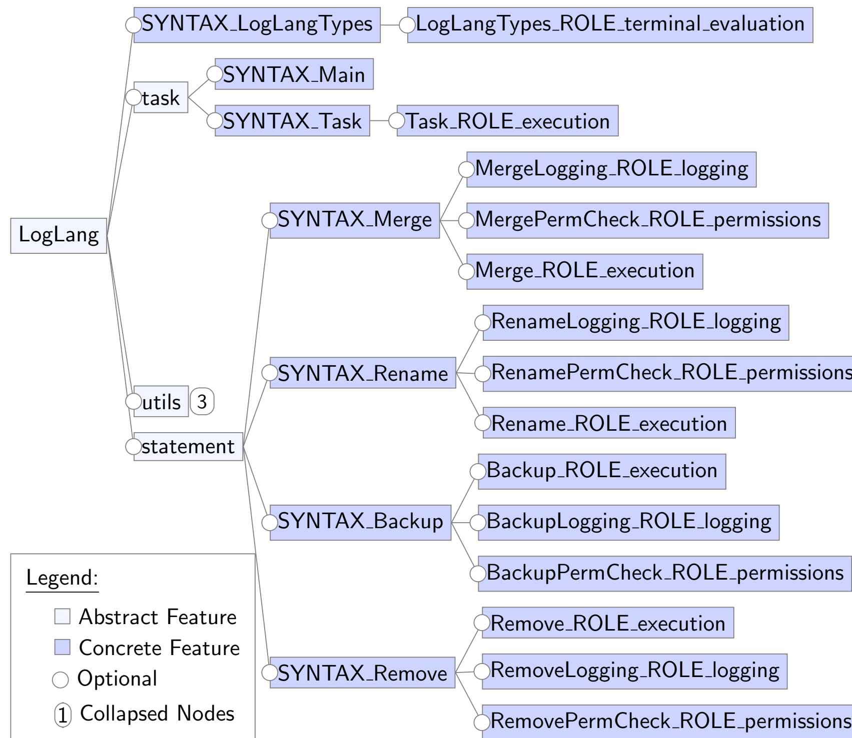


Figure 4.3: *FM of the LogLang language family based on the code presented in Listing 2.1 and generated using AiDE 2 and Algorithm 1.*

the Eclipse build process. Whenever a **`module`** is added, deleted or changed in the workspace, the incremental Neverlang compiler compiles the most recent version of the file and triggers an update of the FM, according to the latest pool of available language features.

Neverlang Editor. The *Neverlang Editor* (Fig. 4.4, language developer and language user layers) is an LPL-driven editor for the development of language features, introduced in [131]. It collects and integrates IDE services specified in **`modules`** to deploy a tailored editor for language variants.

Since the Neverlang compiler is bootstrapped, the *Neverlang Editor* serves as an environment for both the development of language components and the usage of language variants. Furthermore, it provides syntax highlighting and code-completion services by cross-referencing the IDE specifications within the language components in language variants, e.g., **`categories`** hold stylistic information for a grammar fragment, **`out-buckets`** are fed with text from a terminal or nonterminal symbol and can be retrieved to provide suggestions for code-completions using the **`in-buckets`** directive (Listing 2.1 lines 7-9). This work contributes to the *Neverlang Editor* by integrating dynamic reloading of language implementations within the same Eclipse running instance to better suit the LPL engineering process and the incremental development of

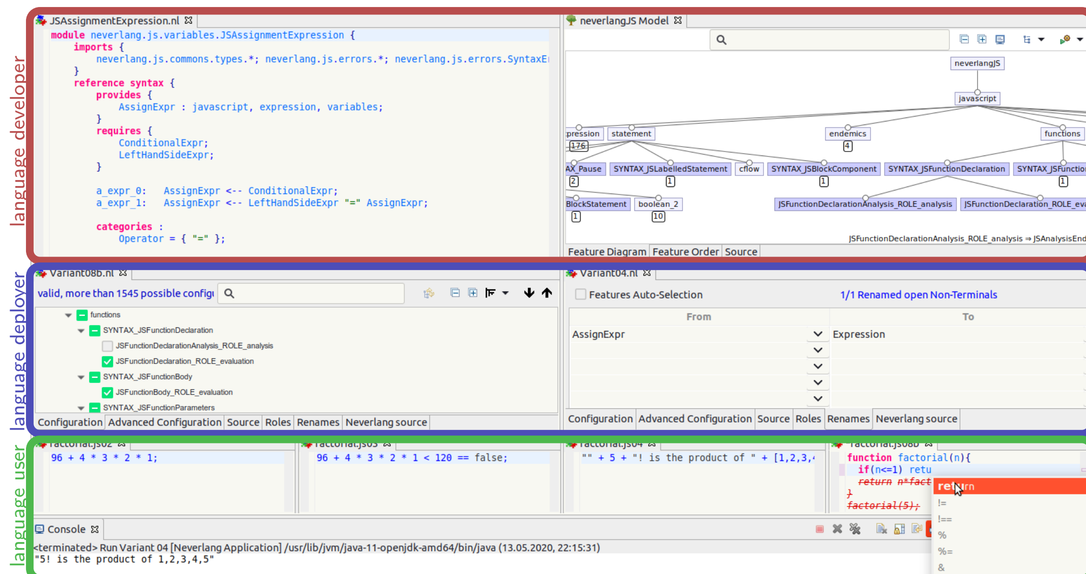


Figure 4.4: *Overview on the Neverlang LPL development environment highlighting the views provided to each role, according to the engineering process presented in Sect. 4.2.1.*

multiple language variants.

Language Configuration Editor. Language features can be combined into a language configuration using the Neverlang `language` construct. AiDE supports the automatic generation of `language` and `slice` files through the *Neverlang Language Configuration Editor* (Fig. 4.4, language deployer layer). It extends the default FeatureIDE *Configuration Editor* to support the creation of language variants. The variability space of an LPL can be further expanded with regards to its SPL equivalent by allowing for language restrictions that would normally lead to invalid configurations. Due to the *domino effect*, removing a language feature requires all features dependent on it to be removed as well. In case of language grammars, this is often due to open nonterminals. Neverlang permits `renaming` to stop the domino effect, *i.e.*, an open nonterminal can be renamed to a provided nonterminal to fill the gap and obtain a viable language variant although the feature configuration is invalid. The *Neverlang Language Configuration Editor* adds a *Renames* tab (Fig. 4.4, language deployer layer, right side) to incorporate this functionality into FeatureIDE. In addition, the compilation phases for the interpreter/compiler can be specified in the *Roles* tab by defining the succession and traversal of semantic actions. The source generation process is triggered whenever changes are saved. This will automatically generate all `slices` by collecting the selected syntactic and semantic features and compile the corresponding language variant. The generated code for the language variant can be inspected in the *Neverlang source* tab. Since the Neverlang compiler translates `language` files into Java classes, the language variant's interpreter can be immediately tested within Eclipse by running the generated Java class as a *Java application*.

Deployment of a Language Product. FeatureIDE can deploy languages and language families using the AiDE 2 library and Gradle. Upon creation of an LPL project, AiDE 2 optionally generates a `build.gradle` file with a `distribution` task, which can be used to generate a Java archive containing the required project binaries and their dependencies. Binaries of language variants can be registered by referring to either binary project folders or `jar` archives and specifying the fully-qualified name of the language class file within the Neverlang `config.json` file.

Once a language is registered, the *Neverlang Editor* can load its basic IDE services. Programs compliant with any registered language variant can be executed inside the Eclipse console (Fig. 4.4, language user layer, bottom part) by producing a *Neverlang run configuration* for that file specifying the desired language variant.

Consistency Preservation. AiDE 2 preserves consistency between the LPL artifacts by orchestrating the Neverlang compiler, the AiDE algorithm and FeatureIDE. Any change to a Neverlang source file (language components) in the workspace triggers the Neverlang compiler. The compiler is responsible for both the translation of Neverlang into the target language (Java by default) and the synchronization of an *environment* holding all the language features relevant to the FM creation. AiDE 2 issues the regeneration of the FM according to Algorithm 1 whenever the Neverlang source update causes an update to the environment. The language developer reviews the updated version of the FM inside the FM Editor and decides to either discard or accept the changes by canceling or performing the save operation. Finally, the language developer commits the LPL. Upon checking out the latest FM, FeatureIDE notifies the language deployer by applying a warning on any inconsistent configuration with regards to the current FM. The language deployer reviews the language configurations, to solve any inconsistency in the FM, and commits any change. The *Neverlang Language Configuration Editor* automatically reestablishes a consistent language configuration whenever the FM change is not substantial—*i.e.*, no concrete features are renamed or removed. In this way FeatureIDE supports the incremental development of LPLs with little to no side effects.

Summary. Neverlang and AiDE 2 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. This toolchain constitutes a set of tools for system designers in support of the LPL engineering process. All elements of the toolchain coexist in the same development environment to allow for an iterative and agile development process with a tight feedback loop.

4.2.3 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 corresponding metrics (discussed in Sect. 4.2.4) are based on the works of

Briand *et al.* [26] and Coleman *et al.* [50], 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 an 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 [188]. 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 [173]. 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 [117] or split complex LPLs into smaller LPLs towards a multi-LPL approach [184]. Dealing with smaller LPLs also stems the problem of increased connectivity associated to programmers using information they should not possess about other modules [169]. 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 [104].
– *encapsulation*— concept that access to the names, meanings, and values of the responsibilities of a class is entirely separated from access to their realization [104].

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 [104].
– *complexity*—degree to which a system's design or code is difficult to understand because of numerous components or relationships among components [104].
– *modifiability*—degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality [104].

In particular, an 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 discussed in this section might be incomplete, as it represents the foundations of a design methodology for Neverlang LPLs that could later be extended.

Properties of well designed language components. A well designed modularization brings several benefits to the entire software system [170]: 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 [170]. 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 in a bottom-up fashion. 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 [148, 182, 59, 75] and that automated software maintainability analysis can be used to guide software-related decision making [50]. 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 [26]. The correlation between those properties was supported by empirical evidence [25, 30, 29]. Cohesion and coupling can be used to evaluate several aspects of modules design. Coupling is strongly related to the probability of fault detection [27]. On the other hand lack of cohesion—despite not being directly associated to faults empirically—can hinder the design of the system. Parnas [170] 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 eventually improves development time, flexibility and comprehensibility [170]. 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 [104].
- *coupling*—degree of interdependence between software modules [104].
- *complexity*—degree to which a system or component has a design or implementation that is difficult to understand and verify [104].
- *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 [104].

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 discussed in this section might be incomplete, as it represents the foundations of a design methodology for Neverlang LPLs that could later be extended.

4.2.4 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 Sect. 4.2.3. 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 Sect. 4.2.1 and the LPL engineering environment presented in Sect. 4.2.2 are designed to work in conjunction with the evaluation of these metrics. More precisely, in Sect. 4.2.1 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. This step is streamlined by leveraging the usage of metrics for assessing the level of granularity: the LPL engineering environment provides tools for the evaluation of these metrics, thus enabling the detection of design errors without requiring manual inspection by the *language deployer*. When an 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. In agile terms, the sprint goal can be achieved only when the product meets the required quality standards in terms of the used metrics.

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 Sect. 4.2.3. 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 (Sect. 4.4) 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 [23]. 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 [204]. 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. There is a large variety of metrics can be perform the quality assessment of the FM in literature [69], therefore a thorough evaluation of all the metrics proposed in literature will not be discussed in this dissertation. Instead, this work focuses on a subset of the existing metrics to show how literature fits our vision of an LPL design methodology. For each property of the variability space from Sect. 4.2.3, 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 an LPL is nonnegative because it is impossible to understand an 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 [146] 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 [127].

• *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 [98]. 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:

1. classes are mapped to Neverlang modules;
2. methods are mapped to semantic actions;
3. class fields are mapped to grammar attributes.

Table 4.1 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

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 [45] (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 [46]
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 [100]
Co (connectivity)	Let $\|V\|$ be the number of vertices in the graph G from LCOA3, and $\|E\|$ the number of edges. Then $$\text{Co} = 2\frac{\|E\| - (\|V\| - 1)}{(\|V\| - 1)(\|V\| - 2)} .$$	Co or C [26, 100] (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 $$M_j = \{s \in S \mid s \text{ references } a_j\}$$ and $\mu_j = \|M_j\|$ then $$\text{LCOA5} = \frac{\frac{1}{m}\left(\sum_{j=1}^{m} \mu_j\right) - n}{1 - n}$$	LCOM5 [26]
Coh (cohesion)	Given n, m and μ_j as in LCOA5, $$\text{Coh} = \frac{\sum_{j=1}^{m} \mu_j}{nm}$$ is a normalized representation of individual references to attributes in actions.	Coh [27] (cohesion)
CBM (coupling between modules)	A module is coupled with another if actions in either 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 [46] (coupling between objects)

Table 4.1: *Cohesion and coupling metrics for language product lines.*

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 [174]. Now we contextualize the metrics from Table 4.1 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.

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 = ∞ 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 4.1).
• *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 = ∞ 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 4.1) 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 = ∞ 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 = ∞ 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 and 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) [148], Halstead's complexity metrics [90]—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) [49] and the normalized derivative used in Visual Studio (VS) [41]. 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.

4.3 Case Study: Javascript Language Family

This section demonstrates the applicability of the design methodology, with a focus on the iterative engineering process presented in Sect. 4.2.1. We will discuss how each of the three roles involved in the LPL engineering process interact with the tools for systems designers provided by AiDE 2. The demonstration case study is replication of the gradually teaching programming experiment presented in [129]. The experiment was undertaken in a distributed environment and versioned using `git`[3]. The repository has 3 different contributors, each one embodying one of the three roles: *language developer*, *language deployer*, and *language user*; the changes made by each author can be reviewed by inspecting the commit history and highlights the underlying distributed, incremental LPL engineering process.

4.3.1 Family of Javascript-based Languages

As a case study, we employed and refactored the LPL for the family of Javascript-based languages introduced in [209] and used for a language evolution experiment in [34]—neverlang.JS. Although Javascript provides a realistic level of complexity and variety of language features, its base implementation only amounts to 3599 *lines of code* (LoC) in 79 slices, 83 modules, and 3 endemic slices and support classes. The base implementation was later extended with additional modules and classes, but those extensions are not discussed in this experiment. The neverlang.JS interpreter mostly conforms to the *ECMAScript 3 Language Specification* (ECMA-262), except for any built-in functions that were not re-implemented in Neverlang. Despite its limited size, the FM generated by AiDE, as partially depicted in Fig. 4.5, comprises 234 language features (including 43 abstract features) and 162 cross-tree constraints [4]. Notably, some of them are redundant, this, however, results from the individual generation of cross-tree constraints. As such, the FM represents a language family of at least 139713 valid feature configurations—*i.e.*, fulfilling all tree and cross-tree constraints—and corresponding language variants—estimated via FeatureIDE's number of products analysis. Note that this number is a lower bound to the actual number of configurations because the Neverlang renaming mechanism could still be used to derive a viable language variant from an invalid feature configuration. Unfortunately, neither FeatureIDE nor AiDE 2 can provide a more accurate estimate on the number of viable language configurations[5] and thus the actual viability of a renaming depends on the language implementation.

---

[3]The repository is available at:`https://cazzola.di.unimi.it/aide/neverlangjs-lpl.git.tgz`. To setup a copy of the repository on your machine please download it, extract it, switch to the new directory and run: **`git clone .git neverlangjs-lpl`**.

[4]The full feature model is available: `https://cazzola.di.unimi.it/aide/neverlangJS-fullfm.png`.

[5]The number of possible configurations is exponential with regards to the number of features in the FM, thus most IDE tools impose a timeout over the estimation. This limitation improves the user experience by avoiding delays and actually provides useful information with regards to the variability space, as will be discussed in Sect. 4.4.

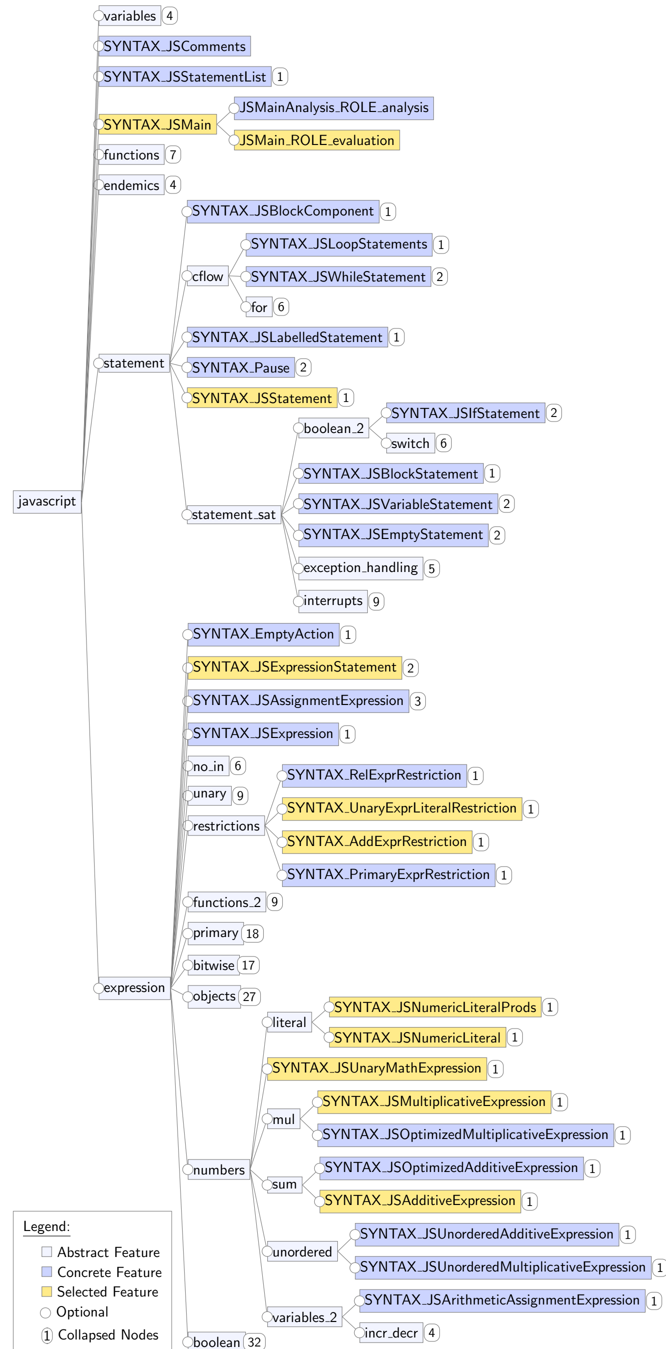


Figure 4.5: *FM generated by AiDE 2 for the Javascript family.*

4.3.2 Incrementally Teaching a Language

For our demonstration case study, we adapted the teaching schedule, proposed in [34], for gradually teaching programming to students. Fig. 4.6 highlights the 14 language

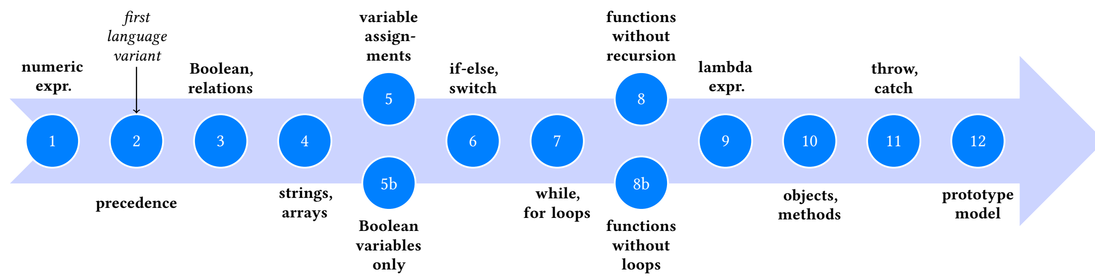


Figure 4.6: *Course schedule and language products for teaching* Javascript, *adapted from [34].*

variants (circles) which gradually introduce new language features over the duration of the programming course. In particular, we included three additional language variants. The first, Variant 5b, is a language specialization that only permits Boolean expressions and the declaration of Boolean variables. It mirrors Variant 5, yet focuses on propositional logical formulas. In contrast, Variant 8 and Variant 8b were introduced to teach recursion or lack thereof. While the former permits function calls yet prohibits recursion, the latter supports recursive function calls yet removes loops from the language variant. Henceforth, we will take the perspective of the teacher as the language deployer tasked to configure and deploy the 14 language variants to students that act the role of language users.

4.3.3 Growing Javascript Variants—The Deployer's Perspective

To create the increasingly complex language variants, the language deployer clones the neverlang.JS git project provided by the language developer and opens it within AiDE 2. Then the deployer uses the *Neverlang Language Configuration* wizard to create a new language configuration and opens it in the *Neverlang Language Configuration Editor*. Initially the configuration is empty and the desired language features must be selected from the *Configuration* tab, as in the second layer of Fig. 4.4. Henceforth, we use a dot-notation to denote the path to a feature from the root of the FM, shown in Fig. 4.5.

In our case, we started with Variant 2, a language variant only allowing numerical expressions. For this example, Fig. 4.5 highlights the relevant features with a yellow background. The language deployer starts by selecting the desired language features for this language specialization found under `expression.numbers`, *i.e.*, `sum.JSAdditiveExpression` and `literal.JSNumbericLiteral`. The corresponding `evaluation` semantic features are also selected. The configuration editor automatically selects `endemics.JSMathEndemic` but yields an invalid configuration due to unsatisfied cross-tree constraints. However, it suggests to select `mul.JSMultiplicativeExpression` and `literal.JSNumeric` and `literal.LiteralProd` from the `expression.numbers` subtree. Although this yields a valid configuration, the resulting language variant is still empty, as the selected productions are never reached. The language deployer can now select the language features leading from the start symbol (`Program` by convention) to the numerical expressions to make them reachable. Such features include

`JSMain` and `statement.JSStatement` (top of Fig. 4.5) and `JSExpressionStatement` and `JSPrimaryExpressions` (from the `expression` subtree). At this point, the configuration editor suggests the `JSExpression` feature which cascades into additional dependencies that the language variant should not include. In fact, at this point the deployer requests help from the language developer who, in turn, assesses that in this case renaming is not enough to solve the dependencies and introduces two alternative language components, *i.e.*, `AddExprRestriction` and `UnaryExprLiteralRestriction`, that introduce the missing productions needed to make this language variant viable. Immediately, after adding the new language components the FM is regenerated. The language developer can now commit the newly created Neverlang modules and the changes to the FM file to the repository. The language deployer pulls the latest changes which cause FeatureIDE to automatically reload the *Neverlang Language Configuration Editor*, *i.e.*, publishing the corresponding language features within the `expression.restrictions` subtree. Finally, the configuration is valid and the deployer only needs to declare the sequence of the semantic actions and corresponding traversal, *i.e.*, *evaluation* with *preorder*, in the *Roles* tab. Once the configuration is saved, the language variant is automatically generated and ready to be published to the repo for testing. To do so, the *language user* can simply update the repository and run the generated Java program within the `gencode.aide` package in the `gen-src` folder as a *Java application*. This starts an interactive interpreter of the language variant in the *Console*. After evaluating that the language variant correctly parses and evaluates the desired numerical expressions, they can proceed to package the language variant via gradle and registering it to Neverlang selecting a unique file extension, e.g., `js02`. As a result, language users can now use the *Neverlang Editor* to edit all `js02` files with the correct syntax highlighting and code-completion. They can also run the files using the *Neverlang run configuration* with the corresponding language variant, *i.e.*, *Variant 2* and file path, as showcased in [131]. Fig. 4.4 (language user layer) showcases the editors for the first four language variants with syntax highlighting and code-completion, and the execution of the `factorial.js04` program with the fourth language variant.

Building on these language variants, the language deployer can reuse previous language configurations to derive the other language variants. Moreover, with feedback from the language developer they can also derive language variants, which require renaming, such as, Variant 4 to exclude assignments and Variant 5b to exclude bitwise operations and relations, or diverging semantics, such as, Variant 1 requiring language components violating the precedence rules and Variant 8 which prohibits recursive function calls. Henceforth, we change the perspective to the language developer detailing how Variant 5b and 8b were facilitated.

4.3.4 Refactoring the neverlang.JS LPL—The Developer's Perspective

Variant 5b was derived restricting variant 5 by removing support for numerical values, strings and arrays to allow the language deployer to configure a language variant of only Boolean assignments and expressions. Introducing the new variant in the LPL highlighted a refactoring opportunity. Listing 4.1 showcases the

implementation of assignments before the refactoring process. The syntax definition for the `JSAssignmentExpression` **`module`** was too coarse-grained and required the `AssignOperator` nonterminal to be defined leading to invalid language configurations when numerical values and arithmetic assignment operators are not present in the variant. To fix this we refactored the `JSAssignmentExpression` into two **`modules`** (cf. Listing 4.2) to distinguish standard assignments from arithmetic assignments, e.g., `+=`. In total, the added `JSArithmeticAssignmentExpression` amounts to 61 additional LoC. After completing the refactoring process, the LPL development environment automatically compiles all changed language components and regenerates the FM. Then, the language developer only needs to review and save the FM via the FM Editor. As previously outlined, this change is propagated to all *Language Configuration Editors*. Additionally, all variants 5–13 that were using the modified `JSAssignmentExpression` feature are marked with a warning indicating that they need to be reviewed by the language deployer.

A similar refactoring process was required to derive Variant 8 from Variant 8b to introduce functions. Recall, Variant 8 adds functions with loops but prohibits recursive calls whereas Variant 8b permits recursive functions but lacks loops. For the former, this required selectively changing the language semantics. Hence, the language developer opted for preceding the evaluation with an additional semantic role, *i.e.*, the `analysis` role. This role was added to `JSFunctionDeclaration`, `JSFunctionCalls`, and `JSMain` to retrieve the static call graph from the parsed program and prevent its evaluation when a cycle is detected.

Listing 4.3 shows the implementation of the semantic actions added to function declarations and function calls. The former uses a stack to keep track of the current function in scope whereas the latter adds call edges from the current function to the called function. Excluding the employed third-party graph library, this extension only introduced 52 new LoC. The impact of this refactoring on the FM is limited to the added language features without changing its structure. Thus no other language configuration was affected by the change. Hence, to configure Variant 8, the language deployer only needs to pull the FM with the new features, add those features to the configuration, select the corresponding analysis role and list it as preceding the *evaluation* phase in the *Roles* tab.

```
1 module neverlang.js.variables.JSAssignmentExpression {
2   reference syntax {
3     provides { AssignExpr: expression, variables; }
4     requires { LeftHandSideExpr; AssignOperator; }
5     AssignExpr ← LeftHandSideExpr "=" AssignExpr;
6     AssignExpr ← LeftHandSideExpr AssignOperator AssignExpr;
7   }
8   role (evaluation) {/*...*/}
9 }
```

Listing 4.1: *Too coarse-grained* **`module`** *defining assignment.*

```
module neverlang.js.variables.JSAssignmentExpression {
  reference syntax {
    provides { AssignExpr: expression, variables; }
    requires { LeftHandSideExpr; }
    AssignExpr ⟵ LeftHandSideExpr "=" AssignExpr;
  }
  role (evaluation) {/*...*/}
}
module neverlang.js.variables.JSArithmeticAssignmentExpression {
  reference syntax {
    provides {
      AssignExpr: expression, variables, numbers, strings;
      AssignOperator: ..., operators;
    }
    requires { LeftHandSideExpr; AssignExpr; }
    AssignExpr ⟵ LeftHandSideExpr AssignOperator AssignExpr;
    AssignOperator ⟵ "+="; // other rules for *= /= %= -=
  }
  role (evaluation) {/*...*/}
}
```

Listing 4.2: *Assignment split into separate* **`modules`**.

Variant	Description	Roles	Features		Renames	LoC	
1	Numeric expressions and operators (without precedence)	evaluation	37	(11)	-	330	(80)
2	Numeric expressions and operators (correct precedence)	evaluation	42	(5)	-	354	(82)
3	Booleans and relational operators	evaluation	70	(5)	-	780	(215)
4	Strings, arrays and their operators	evaluation	76	(5)	1	866	(229)
5	Variables and assignments	evaluation	53	(3)	-	1356	(350)
5b	Only Boolean assignments and expressions	evaluation	106	(3)	3	641	(154)
6	Conditional statements (e.g., if, else, switch)	evaluation	145	(0)	-	2010	(571)
7	Loop statements (e.g., while & for)	evaluation	162	(0)	-	2479	(821)
8	Loops and functions without recursion	analysis, evaluation	171	(10)	-	2727	(877)
8b	Functions with recursion, but without loops	evaluation	183	(3)	-	2470	(739)
9	Functions and lambda expressions	evaluation	181	(3)	-	2766	(884)
10	Objects and Methods	evaluation	186	(3)	-	2937	(974)
11	Exception Handling	evaluation	193	(3)	-	2990	(984)
12	Constructors and prototype model	evaluation	208	(3)	-	3224	(1054)
Complete	Variant conforming to ECMAScript 3	debug, evaluation	234	(0)	-	3599	(1194)

Table 4.2: *Overview on the 15 NeverlangJS language variants highlighting the selected features including features for language specialization in brackets as well as total lines of code (LoC) including Java code in brackets.*

To recap, these cases illustrate the benefits of a tight feedback loop from the language variant's deployer to the LPL developer in a distributed environment, such that the language developer only needs to make small changes to the LPL to gradually make more of the language deployer's language variants viable.

4.3.5 Comparison of the 15 Language Variants

During the course of this case study, we incrementally created 15 distinct language variants of increasing complexity including the 14 variants for the teaching schedule

```
module neverlang.js.analysis.JSFunctionDeclarationAnalysis {
  reference syntax from
    neverlang.js.functions.JSFunctionDeclaration
  role(analysis) {
    func_decl: .{
      String foo = $func_decl[1].toTerminalString();
      String args = $func_decl[2].toTerminalString();
      $$CallStackBuilder.declare(foo, args.split(","));
      eval $func_decl[3]
      $$CallStackBuilder.pop();
    }.
  }
}
module neverlang.js.analysis.JSFunctionCallsAnalysis {
  reference syntax from
    neverlang.js.expression.JSFunctionCalls
  role (analysis) {
    c_expr: .{
      String foo = $c_expr[1].toTerminalString();
      String args = $c_expr[2].toTerminalString();
      $$CallStackBuilder.call(foo, argsString.split(","));
    }.
  }
}
```

Listing 4.3: *Semantic actions for creating the call graph.*

and the full-fledged version of Javascript while refactoring the underlying neverlang.JS LPL according to need. Table 4.2 provides an overview on the 15 created language variants. For each language variant Table 4.2 highlights its semantic roles, as well as, it indicates the total number of selected (abstract and concrete) features; in brackets, the number of features introduced and selected for language specialization. The four newly introduced features were needed for all the variants except 6, 7 and the full-fledged Javascript whereas variants 2, 3 and 4 needed two specializations each. `UnaryExprLiteralRestriction` was used eleven times, `RelExprRestriction` was used twice and the remaining two slices were used once. The addition of language components for specialization was mainly needed to deal with the *domino effect*, but in case of Variants 4 and 5 the renaming mechanism sufficed for its resolution, as indicated in the *Renames* column in Table 4.2. In these two cases, the automatic feature selection was disabled. Consequently, all but these two language configurations are considered valid by *FeatureIDE*, yet all 15 configurations produce viable language variants. To sum up, there were a total of eight language features that caused the *domino effect*, four of which were solved by adding dedicated features to the LPL and four by means of nonterminal renaming.

From our experience, we observed that the distributed, incremental development, rapid testing and deployment significantly reduced the effort to create new and provision language variants. Due to the power of FeatureIDE's feature configuration with automatic feature selection and feature suggestion, the creation of viable language

variants was significantly reduced when compared to manually writing Neverlang language files. Additionally, the total number of LoC required to build each language variant from scratch is shown in the last column whereas the lines of included Java code is shown in brackets. Granted, this assumes that each language variant would have been built from scratch using *Neverlang*, still it illustrates how the LPL engineering process could speed up the creation of language variants and improve reuse among members of a family of languages. Last but not least, with this case study we could illustrate the suitability of our LPL development environment for the teaching case, as it simplified the teacher's task to create viable language variants. AiDE 2 is applicable for the distributed, incremental development, configuration and deployment of LPLs, as it directly supports the LPL engineering process. This section shows how AiDE 2 is a suitable collection of tools for systems designers, as it provides all the views and services required/expected by language developers, deployers, and users involved in the engineering process.

4.4 Evaluation

In addition to the demonstration case study, we setup an experiment assessing the quality in the design of LPLs with respect to the properties defined in Sect. 4.2.3, by measuring the metrics presented in Sect. 4.2.4. The experiment tries to answer $\mathbf{RQ_{4.1}}$ by applying the proposed metrics against a wide range of LPLs and $\mathbf{RQ_{4.2}}$ 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.

4.4.1 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 the development environment discussed in Sect. 4.2.2, comprised of AiDE 2.0.1 and FeatureIDE 3.6.1.

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:

1. *legacy* LPLs created before the introduction of the design methodology;
2. *sub-languages* LPLs created applying the design methodology;

Category	Project	From	LoC	(actions)	Modules	Features	(semantic)	Constraints	Configs
legacy	Neverlang	[209]	1650	(349)	40	81	(41)	18	231935 †
	LogLang	[35]	284	(38)	15	28	(18)	19	104532
	Javascript	[34]	4199	(1399)	108	262	(121)	162	172169 †
	State Machines	[211]	948	(247)	24	64	(37)	36	271356 †
	$\text{Tyl}_{\text{legacy}}$	–	4981	(2335)	78	190	(123)	186	185459 †
	Java	[129]	5488	(1233)	113	307	(169)	180	155522 †
	Java Role Extension	[129]	202	(36)	5	19	(10)	0	3156
	Object Teams	[129]	1036	(222)	16	55	(46)	10	288678 †
	PowerJava	[129]	300	(77)	7	26	(14)	3	49193
	Rava	[129]	309	(76)	5	20	(10)	1	2866
	Java Relations	[129]	1026	(284)	17	61	(34)	11	284600 †
	Rumer	[129]	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	[209]	63	(8)	3	8	(4)	2	15
	Lambda	–	107	(20)	4	10	(6)	0	150
	$\text{Tyl}_{\text{refactored}}$	–	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	–

Table 4.3: *Feature model information for* Neverlang *LPLs considered in this experiment. 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.*

3. *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 4.3 along with some of the metrics and general project information. For each LPL, Table 4.3 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. Neverlang implements the translator from Neverlang source to Java. Neverlang is an LPL applying the bootstrapping technique [39].

• 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.

• 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 [132] 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 [3] 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

```
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 4.4: *Accepted Java+SM syntax.*

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.
- JS+Slicing is a *refactored* LPL. JS+Slicing is defined as an 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 4.4).

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 [32], 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 [32]. 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,

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

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 this chapter whereas *legacy* projects were developed prior to 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 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 Sect. 4.4.5.

4.4.2 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 4.3 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 **role**s). 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 Sect. 2.4.10, 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 can be measured by traditional metrics for 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 Sect. 2.1, 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 [16] and that computing the exact number of valid configurations is just not feasible for large projects. Table 4.3 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 for the purposes of the engineering process. Finding the language variant that meets the user's requirements in an LPL with several hundreds of thousands of valid configuration is hard regardless of how close the approximation is [173]. It should also be considered that a high number of configurations affects the viability of solutions to NP-hard problems such as slicing [127]. 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 [184]. 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 [200] and Binary Decision Diagrams (BDDs) [54] should be considered. On a side note, recall that renames—*i.e.*, the Neverlang mechanism used to stop the ripple (or domino) 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: the satisfiability solver used by the configuration editor should be updated to exclude some of the constraints during resolution, based on the available renames.

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 points of interest with regards to refactoring opportunities. Language components in atomic sets should be either merged into a unique feature 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 4.3 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);
- $\text{Tyl}_{\text{legacy}}$ 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 [172] (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. 4.7. 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

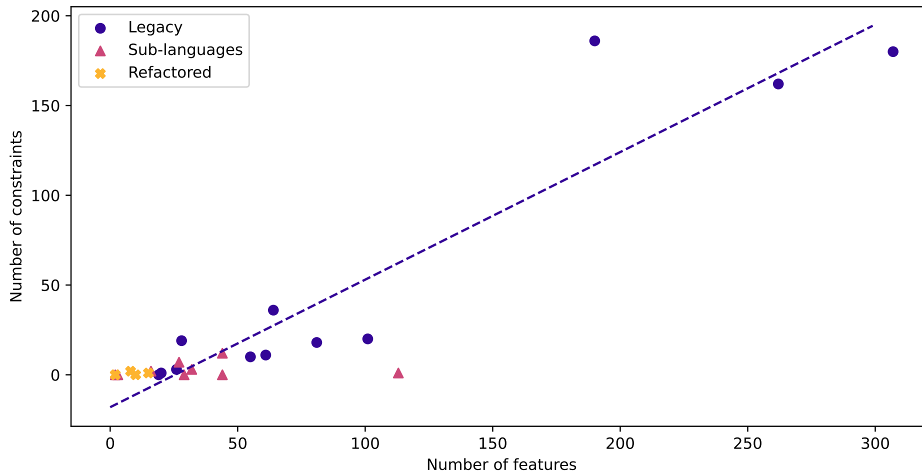


Figure 4.7: *Number of constraints in* Neverlang *LPLs with respect to their number of features.*

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 Sect. 4.2.4) and evaluated in literature [69]. For the same reasons, the results presented in this paragraph are not used to answer $\mathbf{RQ}_{4.1}$ and $\mathbf{RQ}_{4.2}$. Instead, the remainder of this section will focus on the properties of language components introduced in Sect. 4.2.3 and their evaluation.

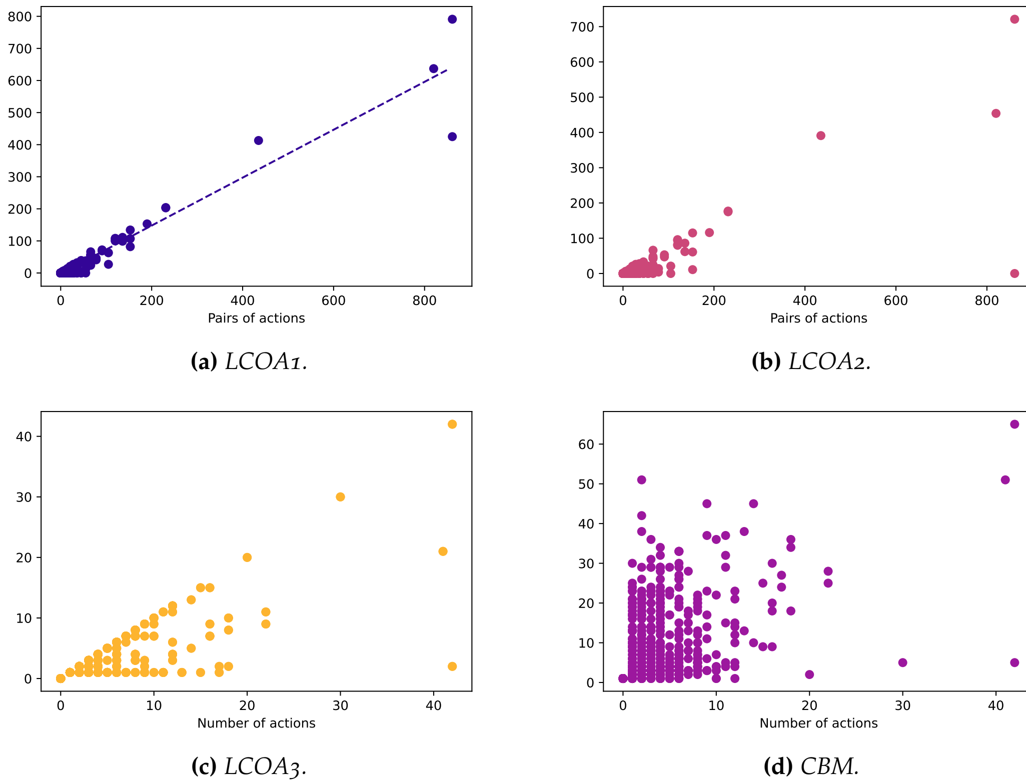
Cohesion and coupling. We assessed each of the LPLs with respect to each of the metrics introduced in Table 4.1. First we can observe that some of these metrics are not applicable to all modules. Co applies only to modules with an LCOA3 value of 1—*i.e.*, completely interconnected actions dependency 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 and thus we report the results only for the sake of completeness. Table 4.4 summarizes the results for each LPL whereas Table 4.5 compares *legacy* and *sub-languages* LPLs. *Refactored* 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 4.5 shows how *sub-languages* consistently perform better on all metrics and highlights how applying a design methodology can improve the quality of language modules.

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
$\text{Tyl}_{\text{refactored}}$	9.00	6.00	2.50	–	0.65	0.66	2.50
$\text{Tyl}_{\text{legacy}}$	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 4.4: *Cohesion and coupling on* Neverlang *modules for each LPL sorted by increasing LCOA1.*

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

Table 4.5: *Comparison between sub-languages and legacy LPLs.*



(a) *LCOA1.* **(b)** *LCOA2.*

(c) *LCOA3.* **(d)** *CBM.*

Figure 4.8: *Cohesion (a-c) and coupling (d) with respect to the number of actions per module.*

Figures 4.8a, 4.8b, and 4.8c 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 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. 4.8d). We can conclude that reducing the size of a module in terms of its semantic actions can increase cohesion but instead increasing the size of a module does not reduce coupling.

Code complexity and maintainability. Table 4.6 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 [67]; 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 4.6 shows that average CC tends to be higher for projects in which average 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

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

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
$Tyl_{refactored}$	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
Tyl_{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	–	–	–	–	–	–	–	–	–

Table 4.6: *Summary of the result of complexity metrics on* Neverlang *LPLs sorted by decreasing MI.*

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 [49] 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 4.6 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.

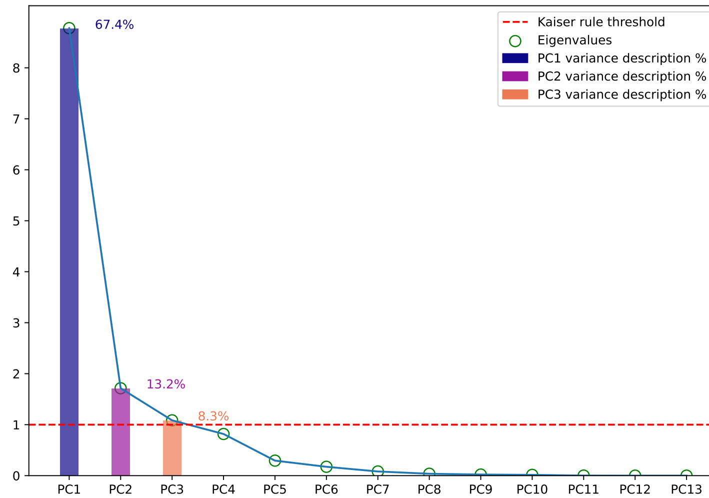


Figure 4.9: *Results of the PCA performed on the subject systems and the considered metrics. Each element on the x-axis is one of the principal components. On the y-axis, their respective eigenvalue.*

4.4.3 Principal Component Analysis

To answer $\mathbf{RQ_{4.1}}$ 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. Fig. 4.9 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 [112]: 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 Sect. 4.2.3 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 described by other dimensions. We can conclude that we are interested in only 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 standalone language component, but only the CBM of a language component within an LPL, which takes more computation time and is less significant for small project. Instead, this result shows that the evaluation of the coupling property can be avoided with regards to the design methodology, since it does not have a big impact on the variance of the results.

4.4.4 Thresholds

To answer $\mathbf{RQ_{4.2}}$ 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 Sect. 4.4.2. However, cohesion and coupling metrics were first defined in Sect. 4.2.4 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. 4.10. 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 which values above Q3 indicate good design and values below Q1 indicate bad design. This analysis reveals the following thresholds.

LoC (total)

good design:	$\text{LoC} \leq 19$
average design:	$19 < \text{LoC} \leq 50$
bad design:	$\text{LoC} > 50$

LoC (actions)

good design:	$\text{LoC} \leq 2$
average design:	$2 < \text{LoC} \leq 14$
bad design:	$\text{LoC} > 14$

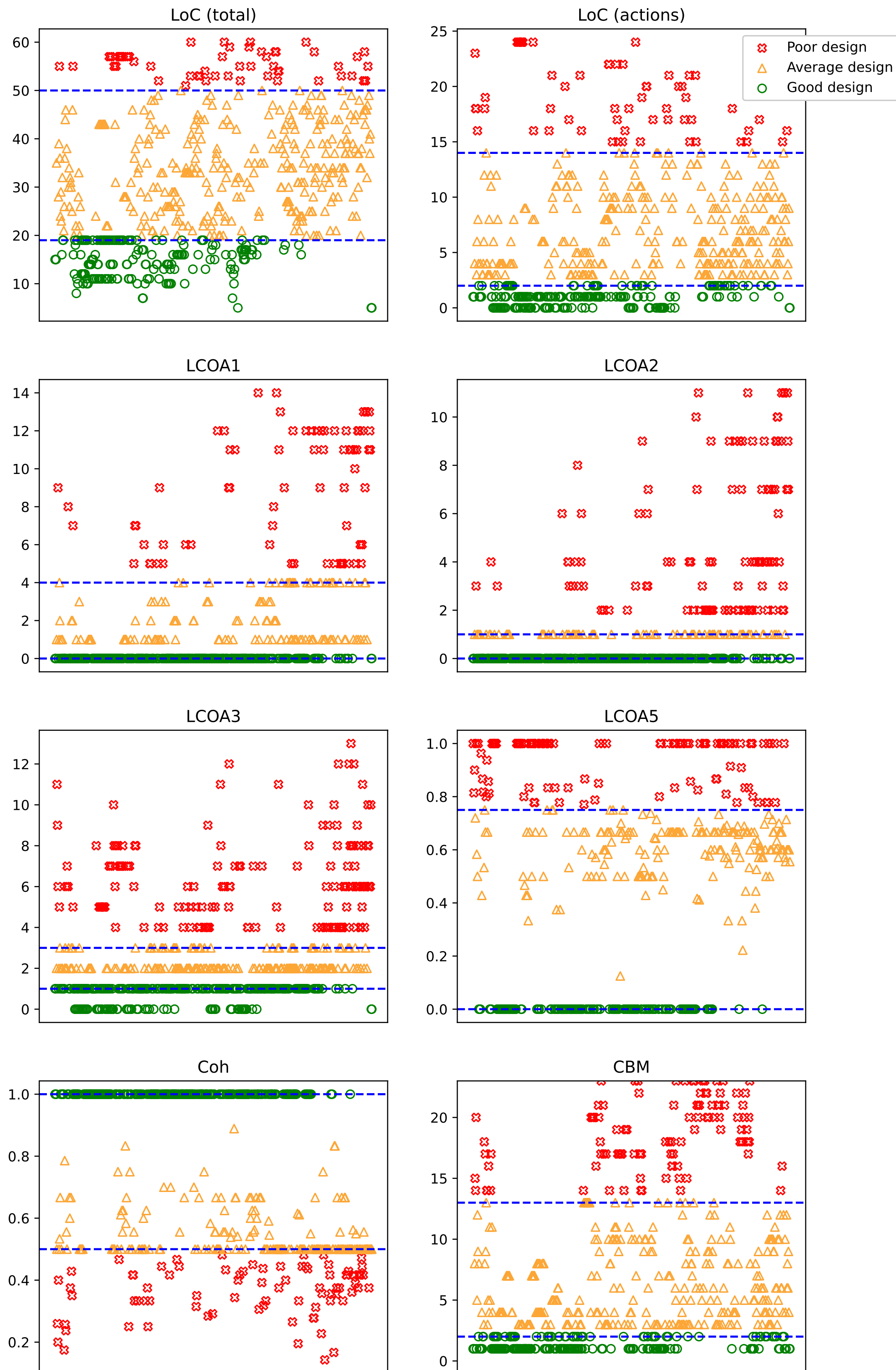


Figure 4.10: *Evaluation results of several metrics and their quartiles. Some results are omitted for better readability. Modules in the IQR is considered to have average design. Modules below Q1 are well designed and modules above Q3 are badly designed or vice versa depending on the metric.*

LCOA1
good design: $LCOA1 \leq 0$
average design: $0 < LCOA1 \leq 4$
bad design: $LCOA1 > 4$

LCOA2
good design: $LCOA2 \leq 0$
average design: $0 < LCOA2 \leq 1$
bad design: $LCOA2 > 1$

LCOA3
good design: $LCOA3 \leq 1$
average design: $1 < LCOA3 \leq 3$
bad design: $LCOA3 > 3$

LCOA5
good design: $LCOA5 \leq 0$
average design: $0 < LCOA5 \leq 0.75$
bad design: $LCOA5 > 0.75$

Coh
good design: $Coh \geq 1$
average design: $0.5 \leq Coh < 1$
bad design: $Coh < 0.5$

CBM
good design: $CBM \leq 2$
average design: $2 < CBM \leq 13$
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.

4.4.5 Discussion

The experimental results outline the amount of data concerning the design quality that can be inferred from an LPL with relative ease. Now we summarize these results with respect to our research questions.

$RQ_{4.1}$. 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.

$RQ_{4.2}$. 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 [69]. 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.

4.4.6 Lessons Learned

Now we provide an overview of the lessons learned from this evaluation and how the design properties translate into best practices when programming Neverlang LPLs. In this context, we also share show our opinion on how this contribution could 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 4.3 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. Larger LPLs should be rather be split into several LPLs, each describing the variability of a family of micro-languages [32].

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, $\mathsf{Tyl}_{\mathsf{legacy}}$

```
1  module BackupSyntax {
2    reference syntax {
3      provides { Backup: backup, statement; Cmd: statement; }
4      requires { BackupSource; BackupTarget; }
5      Backup ← "backup" BackupSource  BackupTarget;
6      Cmd    ← Backup;
7    }
8  }
10 language LogLang {
11   slices BackupSlice RemoveSlice
12          RenameSlice MergeSlice Task
13          Main LogLangTypes bundle(Expressions)
14   endemic slices FileOpEndemic PermEndemic
15   roles syntax < terminal-evaluation < permissions : execution
16   rename {
17         BackupSource → String;     BackupTarget → Expression;
18   }
19 }
```

Listing 4.5: *Refactoring Listing 2.1 to reduce CBM and the constraints introduced by* `AiDE 2`.

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 Sect. 4.2.4. 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 4.4 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 2.1 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. 4.3, 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 4.5. 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 4.5 whereas BackupSource behaves the same in both Listing 2.1 and Listing 4.5. 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 4.5 is more verbose and reduces the readability of the system as a whole, but—as discussed in Sect. 4.2.4—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 [72].

Increasing the number of semantic actions in a module negatively impacts their lack of cohesion: Fig. 4.8a 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 4.5. On the other hand, Fig. 4.8d 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 4.4 shows high CBM while Table 4.6 highlights high MI. This is achieved by the usage of endemic slices, as shown in Listing 2.1: 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 [184] 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 [72]—in which the configuration process focuses on one LPL and the products of the LPL are **`language`** 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 an 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 Sect. 4.4.3 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. This chapter discussed only the Neverlang language workbench and the metrics defined in Sect. 4.2.4 are based on Neverlang-specific 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 [170]. In Neverlang, the

[8] `http://diverse-project.github.io/k3/`

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 [217] 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 [221] 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.

4.4.7 Threats to Validity

Construct validity. 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, *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 [67] 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 [5, 6]. 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.

[9] `https://wiki.eclipse.org.Ecore`

Internal validity. 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. 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 [94], 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 $\mathbf{RQ_{4.2}}$ 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.

4.5 Summary of Chapter 4

This chapter presented a design methodology for LPLs. Four aspects of a design methodology were discussed: the *order in which decisions are made*, the *tools for system designers*, *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 $\mathbf{RQ_{4.1}}$

and $\mathbf{RQ_{4.2}}$ 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. Fine-grained language decompositions (through usage of better abstractions in their syntactic definitions) show better cohesion, coupling, complexity and maintainability results. Limiting the number of features in an LPL speeds up the continuous feedback loop required by the design methodology, since it improves the execution time and the accuracy of FM analysis tools. Moreover, it eases the configuration process of language variants. The PCA revealed that only three out of four considered properties are relevant during the design of Neverlang LPLs. Coupling is not a principal component in determining the variance of the results and CBM is subsumed by other metrics.

5 Mutation Testing based on Language Product Lines

The correctness of both compilers and interpreters is fundamental to reliably execute the semantics of any software developed by means of high-level languages. Testing is one of the most important methods to detect errors in any software, including compilers and interpreters. Among testing methods, mutation testing is an empirically effective technique often used to evaluate and improve the quality of test suites. However, mutation testing imposes severe demands in computing resources due to the large number of mutants that need to be generated, compiled and executed. In this chapter, we discuss the problem of performing mutation testing on language implementations and introduce a mutation approach for programming languages that mitigates this problem by leveraging the properties of language product lines, language workbenches and separate compilations. In this approach, the base language is taken as a black-box and mutated by means of mutation operators performed at language feature level to create a family of mutants of the base language. Each variant of the mutant family is created at runtime, without any access to the source code and without performing any additional compilation.

5.1 The Problem of Quality in Languages Test Suites

Mutation testing is a fault-based testing technique widely used in research for evaluating the quality of test suites. A mutation testing approach proceeds in three phases. First, it creates several modified version of a program, called *mutants*. Second, it runs the test suite against each mutant. A mutant is *killed* if the test suite detects a fault introduced by this mutant, otherwise it is said to have *survived*. Finally, the test suite is given a *mutation score* as the ratio of killed mutants over the total number of mutants. The actual mutants are created by means of *mutation operators*—*i.e.*, rules that are applied to a program to modify its behavior, for instance by changing an operator with another syntactically valid one or by deleting entire statements [164].

Despite its effectiveness, mutation testing is still struggling to become practical due to a few reasons:

1. the cost of executing a large amount of mutants against a test suite is substantial;
2. the mutation operators must replace program tokens with valid alternatives and the mutated program has to be recompiled every time;

3. a mutation testing approach must deal with the human oracle problem and the equivalent mutant problem [109].

Such problems still hold when the system under test (SUT) is the implementation of a programming language interpreter or compiler. One might even argue that testing interpreters is especially significant: the quality of an interpreter affects the correctness of any software developed by its means [125]. While problem 3 draws the most attention in research [171, 215, 118, 145], in this work we want to focus on problem 2 to avoid the cost or recompiling a language implementation for every new mutant. Usually, this is done by applying the mutation operators over an intermediate representation such as LLVM or Java bytecode [93]. However, these approaches only partially solve problem 2 because while they save the recompilation time, the intermediate representation must still be inspected to substitute tokens with valid alternatives. Moreover, handling intermediate representations—*e.g.*, through bytecode manipulation libraries—is usually harder that handling source code.

This chapter discusses a strategy to cope with these issues by leveraging language workbenches and properties that are specific to language implementations. Following the contribution from Leduc *et al.* [137] on the *language extension problem*, we specify and tackle the *language mutation problem*: a language implementation together with its mutants can be treated as an LPL whose products are a family of language mutants of the same base language. The mutation operators performed over the language implementation produce modular language extensions—*i.e.*, features of the LPL. In this approach, the base language is compiled once and then taken as a black-box to which the mutation operators are applied at runtime. The result is a family of language mutants of the same base language.

As a case study, we use the Neverlang [209] language workbench to define six mutation operators that can be applied at language feature level. We dub them *sourceless mutation operators* because they adapt the parser [40] and the semantics [32] of the language implementation without using any source code nor any intermediate representation. Neither the code of the base language nor the code of the mutated language feature are needed. Instead, the mutation approach relies on the introspection and intercession capabilities provided by the Neverlang reflection API [36] to mutate either the language syntax, semantics or both.

The contribution presented in this chapter is validated by answering the following research questions:

$RQ_{5.1}$ Which Neverlang sourceless mutation operators produce variants of the language mutant family that are reasonably hard to discover and kill?

$RQ_{5.2}$ Are mutation operators producing different language mutant variants? Can we obtain similar results by reducing the number of classes?

To answer these research questions we perform an empirical evaluation in which a family of mutants of an ECMAScript interpreter written in Neverlang are tested against the Sputnik conformance test suite for the ECMA-262 specification.

5.2 Mutation Testing Overview

Mutation testing is a fault-based testing technique that can be used to measure the adequacy of a test suite in terms of a *mutation adequacy score*. The origin of mutation testing can be traced back to 1971 in a student report by Lipton [140] and other works from DeMillo *et al.* [62] and Hamlet [91] in the following years. The effectiveness of mutation testing depends on its capability of finding real faults [85]. Since simulating all possible faults is unfeasible, mutation testing only focuses on reasonable faults—*i.e.*, those that are caused by variants of a program that are syntactically close to the correct program. This assumption is called the *competent programmer hypothesis* [62] because we assume that any competent programmer would merely deliver small faults, which can be corrected by a few syntactical changes. Given a set of *mutation operators* $F = \{f_1, \ldots, f_n\}$ and a program p, the traditional mutation testing process [164] generates a set of supposedly faulty programs $P = \{f_1(p), \ldots, f_n(p)\}$ called *mutants*. Next, a test set T is supplied to the system. If the result of running mutant $f_i(p)$ is different from the result of running p for any test case in T, then the mutant $f_i(p)$ is said to be *killed*; otherwise, it is said to have *survived*. The mutation adequacy score (or mutation score) is the ratio of the number of killed mutants over the total number of mutants. The goal of the mutation testing is to improve T until the mutation score is 1.

5.3 The Language Extension Problem

To properly drive our research and to better express its constraints and challenges, we specified the problem briefly introduced in Sect. 5.1 as an instance of the *language extension problem* (LEP). LEP was introduced by Leduc *et al.* [137] as a paraphrase of the classic *expression problem* coined by P. Wadler [222]. The goal of the LEP is to define a family of languages in which a new language can be added by adding new syntax or new semantics; the new semantics can be added over a new syntax or over an existing one [137]. According to the characterization provided by the authors, the LEP is subject to five different constraints that any candidate solution to the LEP should adhere to:

Extensibility in both dimensions. It should be possible to extend the syntax and adapt existing semantics accordingly. Furthermore, it should be possible to introduce new semantics on top of the existing syntax.

Strong static-type safety. All semantics should be defined for all syntax.

No modification or duplication. Existing language specifications and implementations should neither be modified nor duplicated.

Separate compilation. Compiling a new language should not encompass re-compiling the original syntax or semantics.

Independent extensibility. It should be possible to combine and use jointly language extensions independently developed.

Complying to all five constraints is extremely challenging and relaxing one or more of the constraints may be beneficial depending on the given context [137] to favor

interesting design choices.

5.4 The Language Mutation Problem

In this section, we introduce an approach towards the support of mutation testing for programming language implementations using LPLs. Such an approach encompasses:

1. a specification of the problem stated in Sect. 5.1, dubbed *language mutation problem*, as a derivation of the LEP;
2. a meta-model for the resolution of the *language mutation problem* based on language workbenches.

The meta-model will later be applied in Sect. 5.5 and Sect. 5.6 to showcase its applicability with regards to the Neverlang language workbench. This section outlines also outlines as the consequences, applicability and limitations of the resolution meta-model.

5.4.1 Problem Overview

The problem of language mutation can be seen as an instance of the *language extension problem* (LEP) which we will dub as *language mutation problem* (LMP). The LEP lifts the *expression problem* to the context of language engineering to provide a framework for reasoning on language extension and to compare different language extension approaches. Similarly, the LMP lifts the LEP to the context of mutation testing of language implementations. This approach aims to provide a framework for reasoning on language mutation, its challenges and for the comparison of different language mutation approaches.

According to the characterization of the LEP discussed in Sect. 5.3, the LMP concerns:

> The extension of a family of mutants of a base language through changes to the syntax and/or the semantics of one of its members via the application of mutation operators.

By extension, the constraints defined for the LEP are expressed in the LMP context as:

Mutability in both dimensions. It should be possible to mutate both the syntax and the semantics. It should be possible to mutate the semantics according to a new syntax. It should be possible to mutate the semantics of a un-mutated syntax.

No modification or duplication. Existing language specifications and implementations should neither be modified nor duplicated. Mutation operators are functions that produce mutated language features without changing nor duplicating the code of the original language feature.

Separate compilation. Creating a new mutant should not encompass re-compiling the syntax or semantics of the base language.

Independent mutability. It should be possible to use independent mutated language features jointly. Mutated language features are independent when they are the

result of the application of a mutation operator (either same or different) over two different language features.

In the context of the LMP, it is worthwhile to relax the *strong static-type safety* constraint of the LEP: a mutation approach for object-oriented systems needs to be able to make changes to types and data structure declarations [142, 141]. The test suite for a language interpreter should be able to detect any error in the type system and any missing semantics. Therefore, introducing errors in the type system when generating mutants may be beneficial to assess the mutation adequacy of a test suite.

To summarize, a language workbench can solve the LMP—*i.e.*, it can reach its goal—by satisfying each of the four constraints we introduced in this section: *mutability in both dimensions*, *no modification or duplication*, *separate compilation* and *independent mutability*.

5.4.2 Resolution Meta-model

In this section we introduce a meta-model for the resolution of the LMP by tackling each of the four constraints presented in Sect. 5.4.1. We discuss the actors of the software architecture, how these actors interact and the properties that are needed in order to solve the LMP.

Running example. To better drive the discussion, let us introduce a simple language

$$L = (f_1, f_2, f_3, f_4)$$

comprised of four language features: number (integer and floating point values), variable declaration, addition and (bounded) loop. Below, the Extended Backus-Naur Form (EBNF) grammar of this language with start symbol <*program*>.

```
   <program> ::= <statement>+
 <statement> ::= <assignment>
               | <loop>
<assignment> ::= identifier "=" <addition>
  <addition> ::= <term> "+" <addition>
               | <term>
      <term> ::= <number>
               | identifier
    <number> ::= digit+ ["." digit+]
      <loop> ::= "for" <addition> "{" <program> "}"
```

Listing 5.1 shows a program written using this language: the value of variable x is initially set to 5 and it is then decremented by 1 by iterating the for loop x times. Notice that the language does not support subtraction expressions. Several variants of L can be obtained by performing language extension and language restriction over the base language to obtain a family of language variants of L. Two examples are an extended variant with the subtraction language feature

```
1 x = 5
2 y = 0
3 for x {
4     x = y
5     y = y + 1
6 }
```

Listing 5.1: *Exemplary program written in language L.*

<addition> ::= *<term>* "+" *<addition>*
| *<term>* "-" *<addition>*
| *<term>*

and a restricted variant without loops

<statement> ::= *<assignment>*

Notice that Listing 5.1 is still a valid program for the former variant, but it is not for the latter.

Architecture. The LMP resolution meta-model is schematized in Fig. 5.1, which depicts both the software architecture and the mutation testing process by highlighting the interactions among the involved actors. The software architecture is split into three layers: the language implementation, the language workbench and the mutation testing framework.

First, let us focus on the language implementation (blue box in Fig. 5.1) because its modular structure drives the interaction among the three layers. Taking on the

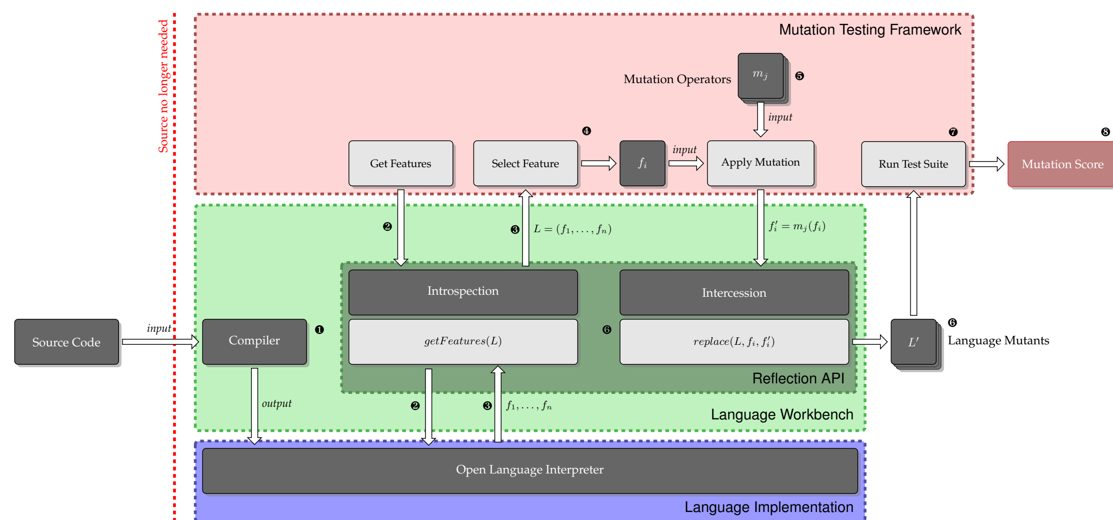


Figure 5.1: *Language mutation problem resolution meta-model, including the process, its actors and their interaction.*

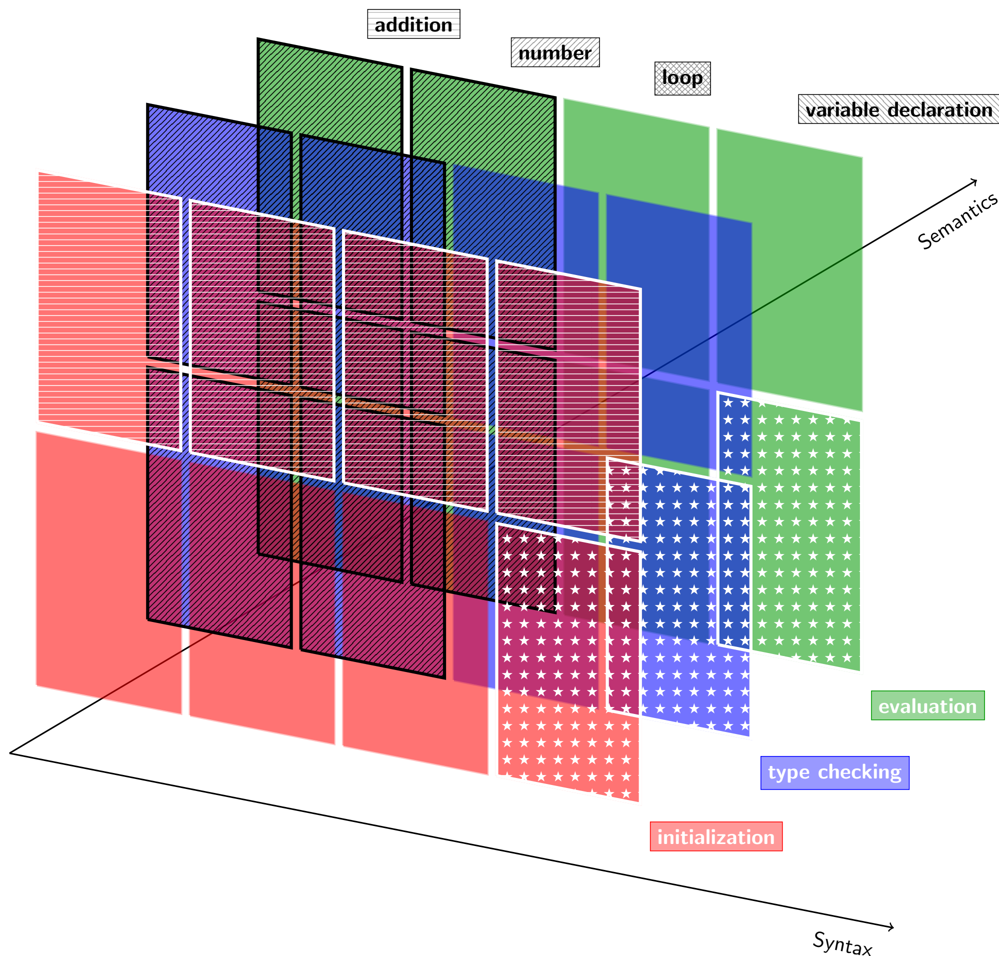


Figure 5.2: *Syntactic and semantic dimensions of a feature-oriented language implementation. Three different mutation operators are performed over the language implementation and highlighted with different patterns, depending on the dimensions they affect: over the semantic dimension case (white stars), over the syntactic dimension (white stripes) and over both (black stripes).*

previous running example, L is implemented in a modular way and it is comprised of four language features. The modular approach used to implement L is schematized in Fig. 5.2. Each language feature is made of a syntax and three semantic phases—each being a traversal of the program's abstract syntax tree (AST): initialization, type checking and evaluation. The initialization semantic phase reads the terminal tokens to establish their types and their values. The type checking phase uses these pieces of information to check the validity of the program with regards to its types and performs any conversion. For instance, an addition between an integer and a float value promotes the integer to float. Finally, the evaluation phase runs each statement in the script. Each color—*i.e.*, each element along the Semantics axis—represents one of the aforementioned evaluation phases. Each element along the Syntax axis represents a syntactic asset—*i.e.*, addition, number, loop and variable declaration; each syntactic asset

can contain more than one grammar production, as represented by the rectangles in Fig. 5.2. The intersection between the two dimensions represents a language feature, comprised of a syntactic asset and its semantics. Implementing the language interpreter according to this abstraction will be useful to solve the LMP. In fact, Fig. 5.2 also shows exemplary mutation operators that can be performed over the language, as we will discuss later in this section. Finally, Fig. 5.1 shows that L is implemented as an *open language interpreter* [36] so that its structure can be reasoned about and modified to affect its behavior. In the context of mutation testing, an open implementation can be leveraged to change the language behavior by switching language features of the base language with mutated language features without encompassing re-compilation of the source code. In other words, an open language interpreter is compliant to the *separate compilation* constraint of the LMP.

The remaining two layers of the resolution meta-model are more straightforward. The language workbench (light green box in Fig. 5.1) is an abstraction over the language implementation that provides primitives to interact with any language developed by its means. This includes a compiler that translates the language source code into an executable language interpreter, a runtime environment (not shown in Fig. 5.1) and a *reflection API* (dark green box in Fig. 5.1) capable of gaining access to hidden aspects of the implementation. To perform language mutation, the reflection API must support two reflection mechanisms: *introspection—i.e.*, the ability to reason about otherwise implicit aspects of the implementation—and *intercession—i.e.*, the ability to act upon otherwise implicit aspects of the implementation [202].

Finally, the mutation testing framework (red in Fig. 5.1) is the most external layer and has no initial knowledge of the base language (the SUT); the mutation testing framework interacts with the language workbench's reflection API to gain knowledge of the SUT and to perform mutation operations. The mutation testing framework also handles the execution of the test suite.

Process. Let $L = (f_1, f_2, f_3, f_4)$ be our running example language and T the test suite for the verification of L. Now, we will overview how the three layers interact to evaluate the mutation score of T according to the meta-model shown in Fig. 5.1.

First, the source code of the interpreter for L is given as an input to the language workbench compiler (Fig. 5.1-❶). The workbench compiler outputs an executable open language interpreter that will be the SUT, as well as the bottom layer of the meta-model. As shown by the red dashed line in Fig. 5.1, the source code is no longer needed throughout the rest of the process: this is fundamental for the meta-model to be able to achieve better scalability by avoiding recompilation.

Next, the mutation testing framework starts the feature selection process: at first the mutation testing framework has no knowledge of the language implementation, other than an identifier[1] provided by the user. To do so, the mutation testing framework interacts with the language workbench layer's reflection API to perform an introspection over the language implementation (Fig 5.1-❷). The request is forwarded to the language

[1]The identifier for a language implementation can be, for instance, the class name when working in Java.

implementation which returns a collection of all the language features (Fig. 5.1-❸). In our example, the introspection mechanism will return (f_1, f_2, f_3, f_4)—*i.e.*, the four features L is comprised of: addition, number, loop and variable declaration respectively. Steps ❶ through ❸ are performed only once throughout the whole process regardless the number of generated mutants.

Next, the mutation testing framework selects one feature f_i among (f_1, f_2, f_3, f_4) as the subject of the mutation (Fig. 5.1-❹). The selection can be performed at random or by means of heuristics. For example, let us assume that $f_i = f_3$—*i.e.*, the loop feature—was selected in this step. The mutation testing framework contains several mutation operators $(m_1, \ldots, m_n)$ and chooses one mutation operator m_j to apply over f_i to obtain a mutated feature f_i' (Fig.5.1-❺). For example, let us take $f_3' = m_j(f_3)$ where f_3' is the loop feature deprived of its semantics. The properties of the mutation operators will be discussed in detail later in this section. The mutation operation can be performed directly through intercession over f_i or by generating and compiling the new mutated feature separately. This step is critical and must be handled in a way that does not violate neither the *no modification or duplication* nor the *separate compilation* constraints. The mutation testing framework uses the intercession capabilities of the language workbench's reflection API to generate a new mutated language L' in which f_i was substituted by f_i' (Fig. 5.1-❻). In our example, $L' = (f_1, f_2, f_3', f_4)$ is an interpreter that accepts the same programs L does, but in which loops have no semantics. For instance, running Listing 5.1 on the interpreter for L' raises no parsing error, but yields a final value of `x = 5` instead of `x = 4`, since the body of the loop is never executed. If T is mutation adequate, L' will be killed when executed against T. As an alternative to intercession, step ❻ can leverage loose coupling between mutated features and the base language using the *black-box aggregation* technique [176]. Regardless of the chosen method, steps ❹ through ❻ are performed several times: once for each language mutant that must be generated.

Finally, the mutation testing framework runs T against all the language mutants generated in the preceding steps (Fig. 5.1-❼) and outputs the mutation score of T (Fig. 5.1-❽).

Mutation Operators. To be compliant to the *mutability in both dimensions* constraint it must be possible to mutate the syntax and the semantics separately. This allows to properly express the variability of the mutants. Fig. 5.2 shows a modularization approach compliant to this constraint and its interaction with mutation operators. As discussed earlier, the language implementation shown in Fig. 5.2 was decomposed over the two dimensions of syntax and semantics; in this case, the implementation is comprised of four syntactic constructs and three semantic phases. To solve the LMP, the mutation testing framework must provide at least three categories of mutation operators:

- along the syntactic dimension;
- along the semantic dimension;
- along both dimensions.

A language mutation operator is a function that takes a language feature as input and returns a new language feature as output. The output is obtained by mutating the original feature along one of the dimensions or both, depending on the category of the mutation operator. Fig. 5.2 shows three different mutation operators that were performed over three different dimensions of the base language. A mutation operator over the *semantic dimension* (white stars) mutated all the semantics of the variable declaration syntax. For instance a mutant may change the type of the variable before it is assigned or increment its value. A mutation operator over the *syntactic dimension* (white stripes) mutated the initialization semantic phase for all four syntactic constructs. An example would be changing the grammar so that <*addition*> and <*assignment*> are merged into the same nonterminal. This would affect the behavior of all features, for instance by allowing nested assignments (`x = y = 5`) and assignments inside loop bounds (`for x = 5 {...}`), both of which are not accepted by the base language *L*. A mutation operator over *both dimensions* (black stripes) mutated the type checking and evaluation semantic phases of the addition and number syntaxes—for instance a mutation operator may change the type of numeric literals from integer to float and vice versa or even increment their value whenever they are used in an addition.

The meta-model does not require a specific number of mutation operators, as long as there is at least one for each of the three categories. For instance, as we will discuss in Sect. 5.5, we used six mutation operators in our application of the meta-model.

The mutation operators should be designed to be compliant to the *independent mutability* constraint: performing a mutation operation must not prevent the application of further mutations to produce high-order mutants. Moreover, it should be possible to perform two mutation operations over two language features. Then, it should be possible to apply the two mutated language features over the base language either jointly or independently. The language workbench must also provide API to generate, compile and load additional mutated language features on demand. Alternatively, the language workbench must provide API to mutate the already loaded language implementation without generating additional source code.

Summary. The actors involved in the meta-model must address all four constraints of the LMP:

- the *mutability in both dimensions* constraint is addressed by a modular implementation of the language interpreter and by a mutation testing framework that provides mutation operators capable of targeting the syntactic and semantic dimensions of such an implementation separately;
- the *no modification or duplication* constraint is addressed by the language workbench that does not change the source code directly and that instead performs mutation operations at runtime through introspection and intercession over an open language implementation;
- the *separate compilation* constraint is addressed by the language workbench, that generates mutated language features at runtime to compile them separately;

- the *independent mutability* constraint is addressed by the mutation testing framework that defines the mutation operators ad applies them one at a time and separately over the base language.

5.4.3 Consequences and Limitations

In this section, we discuss any implications that the LMP resolution meta-model has over the design of interpreters and on the test suite execution. We will also discuss the applicability and limitations of the meta-model with regards to the capabilities of the language workbench.

Families of language mutants. According to the goal of the LMP, by modeling the mutation testing approach in a way that is compliant to the meta-model discussed in Sect. 5.4.2, a base language implementation is mutated to generate a family of language mutants. In the context of language workbenches, a language mutant family can be modeled as an LPL which variability is expressed in terms of its features. Such an LPL contains two types of features:

1. the base features of the SUT (such as $f_1, \ldots, f_n$ in Fig. 5.1);
2. all the results of performing a mutation operation over a base feature (such as $fi' = m_j(f_i)$ in Fig. 5.1).

The products of the LPL are the base language and its mutants. If the number of members of the language family is finite then we say the language family is *closed* [137]. Otherwise, the language family is *open* [137]. Given an LPL with n features, the number of members of the family is at most 2^n. Therefore, if n is finite then the language family is closed. In other words, if the number of possible mutated features is finite, then the family of first-order language mutants[2] is closed, otherwise it is open. Families of high-order language mutants[3] are always open. The rest of this chapter will mainly focus on closed language mutant families, using only first-order operators.

Syntactic mutation operators. Among the three categories of mutation operators the resolution meta-model is concerned with, mutation operators over performed over the syntactic dimension have interesting consequences. In fact, performing mutation operations over the syntactic dimension usually changes the language grammar. Changing the grammar of a language may render some test cases obsolete or cause the grammar itself to become ambiguous. While this is generally against the goal we discussed in Sect. 5.1, it may still be beneficial to generate some language mutants that can potentially cause parsing errors. If running a test suite over a language mutant raises a parsing error, then it means that there exists at least one test case that can capture the inconsistency in the language grammar, causing the mutant to be killed. Otherwise, no

[2] A first-order language mutant is obtained can be obtained by performing a single mutation operation over a base language feature.

[3] A high-order language mutant is obtained by performing any number of mutation operations in succession.

parsing error is risen and the test suite may not be mutation adequate. For example, a mutation operation may change a keyword—such as repeat instead of for in our running example language *L*. Similarly, renaming a nonterminal in the grammar may render the <*loop*> nonterminal unreachable. In both cases, if the for keyword was never used in any case of the test suite, then the test suite is not capable of killing the mutant.

Another possible effect of changing the grammar is increasing the family of programs that the language accepts. We discussed such an example in Sect. 5.4.2: a syntax like `x = y = 5` is not accepted by *L*, but it is accepted by a mutant of *L* in which <*addition*> and <*assignment*> are merged into the same nonterminal. A mutation adequate test suite for *L* should be able to detect this inconsistency by killing the mutant.

The separate compilation constraint. Leduc *et al.* [137] relaxed the *separate compilation* constraint in the context of the LEP in favor of non-functional properties such as performance and readability. Instead, the *separate compilation* constraint cannot be relaxed in the context of the LMP: in that case, all the mutants should be generated and compiled in advance to avoid the cost of recompilation. This is not feasible for any open mutant family. Depending on the size of variability space, this may not be feasible for closed mutant families too.

Applicability and Limitations. While the LMP can be solved by instances of the meta-model proposed in Sect. 5.4.2, this approach has some limitations that should be considered. First, the language workbench capabilities required by this approach are very strict. To the best of our knowledge, only the Neverlang language workbench supports both separate compilation of language artifacts and runtime adaptation with the intercession API. Therefore, while the meta-model is general, its applicability may be limited unless a considerable implementation effort is made to extend the capabilities of the used language workbench. Otherwise, a different solution to the LMP that is compliant to the language workbench's capabilities must be found.

Another limitation of the resolution meta-model is its generality: the mutation testing framework has no initial knowledge of the language implementation and it cannot make any general assumptions, neither on its syntax nor on its semantics. The only requirement is for the language to be implemented in a modular fashion. Instead, all the knowledge of the language is gained at runtime through introspection. Therefore, the mutation operators may be hard to design in a way that is relevant to any SUT and they will usually not be able to target specific parts of the language. A solution to this issue would require switching to an hybrid approach in which such information is provided by the language implementation itself by either declaring sensible parts of the implementation or even the mutation operators directly. Then, the mutation testing framework would access and use this knowledge through reflection. Such an hybrid approach will be part of a future work.

Similarly, the meta-model leverages the modularity of the language implementation to generate the language mutants. While the concept in itself is general, the modularization approach differs wildly depending on the language workbench, therefore it is

impossible to define a set of mutation operators that is valid for all language workbenches. Instead, the meta-model is limited to the definition of the three *categories* of mutation operators discussed earlier—syntactic, semantic and both—since the concepts of syntax and semantics are shared by all language workbenches. The instantiation of these categories into actual mutation operators will depend on the application scenario and on the language workbench of choice.

5.5 Solving the Language Mutation Problem in Practice

In this section, we show the application of the conceptual meta-model proposed in Sect. 5.4.2 to a concrete use case. This section is not meant to restrict the applicability of the meta-model to a specific technological framework. There exist several language workbenches with a different approach to modularization and instantiating the meta-model to each existing language workbench is beyond the scope of this dissertation. However, this section shows the applicability of the approach in for a concrete language workbench. Moreover, we provide some hints on how to generalize the application of the meta-model to other language workbenches.

5.5.1 Mutation Testing in Neverlang

In this section, we discuss how the modularization approach chosen by Neverlang is compliant to the *no modification or duplication*, *separate compilation* and *independent mutability* constraints of the LMP due to its composition mechanism and workbench capabilities.

Notice that the output of the mutation testing process presented in Sect. 5.4.2 is an LPL of language mutants and that Neverlang supports LPL engineering thanks to AiDE, as discussed Chapter 4. In the context of the LMP, mutated language features are handled by the AiDE algorithm to produce the FM of language mutant family, whereas the AiDE composer generates variants of the language mutant family. Moreover, AiDE tracks all unresolved dependencies and guides the language deployer throughout the configuration mechanism.

While the LPL capabilities provided by AiDE are useful to support the generation and deployment of language families, they do not directly address any of the constraints of the LMP. Instead, in the following paragraphs we discuss each constraint and their relation to the workbench capabilities of Neverlang. The *mutability in both dimensions* constraint is closely related to the chosen mutation operators and will be discussed in the following section.

No modification or duplication. Neverlang slices provide mechanisms to adapt existing and initially incompatible language assets to drive their composition into a language feature without modifying the original code. For instance, the **`reference syntax`** of a module can be adapted to an incompatible semantic asset using the **`mapping`** keyword [209] which remaps the nonterminal references in a semantic action to a different

nonterminal. Similarly, slices can attach and detach semantic roles from a language feature using the **`with role`** keyword, as we discussed in Sect. 4.4.6. In both cases, the semantics of a language feature are adapted to a different context without accessing any of the original code and using glue code only—that of the slice compilation unit. Neverlang languages can adapt language features too: the rename mechanism adapts incompatible grammar fragments by renaming nonterminals in their productions. These composition mechanisms are intended to improve the reusability of language artifacts in contexts that differ from what they were originally designed for, however they can be leveraged to mutate the syntax and the semantics of a base language without modifying the original code and thus they address the *no modification or duplication* constraint of the LMP.

Separate compilation. The Neverlang compiler translates Neverlang modules and other language artifacts into Java code that can then be compiled by the stock Java compiler. However, given a Neverlang module, its reference syntax and each semantic action declared in the module is translated into a different Java class [209] and has no references to the other elements in the same module or in other modules. This approach allows for Neverlang modules to be compiled only once and then to be referenced by the glue code in slices and languages. Whenever a new language feature or a new language are generated, only the binaries of the composed syntax and semantics are needed, so no recompilation of modules or existing slices is encompassed. According to the *no modification or duplication* constraint, each mutated feature is implemented as a new slice and therefore creating a new mutant does not encompass re-compiling the syntax nor the semantics of the base language.

Independent mutability. Each Neverlang slice is an independent artifact that embodies a language feature. As it was shown in Listing 2.1 (lines 24-29) and later discussed as a mechanism to create variants of the same LPL, the **`language`** construct handles the composition among all the language features to generate a language implementation. Instead, each slice is unaware of which languages it will be used in. Therefore, using two independently mutated features jointly can be achieved by creating a new language unit in which the two base features are substituted by the two mutated features. Notice that the evaluation in Sect. 5.6 focuses on first-order mutants and therefore there will be no instances in which two mutated features will be used jointly in the same language mutant.

5.5.2 Neverlang Mutation Operators

In this section, we instantiate each of the three mutation operator categories discussed in Sect. 5.4.2. The mutation operators are designed to satisfy the *mutability in both dimensions* constraint by performing mutations either over the syntactic dimension, the semantic dimension or both. The mutation operators are summarized in Table 5.1.

Operator class	Neverlang unit	Category
Rename	Language	Syntax
Attribute Mapping	Slice	Semantics
Mapping	Slice	Semantics
Duplicate Role	Slice	Semantics
Remove Role	Slice	Semantics
Remove Slice	Language	Syntax+Semantics

Table 5.1: *Mutation operators in Neverlang. For each operator we report the compilation unit that is leveraged to perform the mutation without access to source code and which of the three categories it pertains to.*

Overview. As discussed in Sect. 5.4.2, the mutation operators are functions over language features. In Neverlang, the focus is on slices: the mutation operators in this section perform at slice level without modifying the composed modules. Other language workbenches have different composition mechanisms and will require different mutation operators. We exemplify six mutation operators, each belonging to one of the three categories introduced before: along the dimensions of syntax, semantics or both. All mutation operators can be applied independently at runtime, without any duplication nor additional compilation thanks to the language workbench capabilities discussed in Sect. 5.5.1.

For each mutation operator, we exemplify a mutation operation: while the source code is never modified or even accessed, each example shows the change that the operation would make if the mutation was performed at source-level. The goal is to show that these mutation operators adhere to the *competent programmer hypothesis*: each fault always affects only one line of code with small mistakes that a competent programmer could reasonably make. We highlight in green any portion of code that would be added and in red any portion of code that would be removed. For each mutation operator we indicate between parenthesis its pertaining category: syntactic dimension (syntax), semantic dimension (semantics) or both (syntax+semantics). Moreover, we show that the family of first-order mutants generated by a mutation operator is always closed by providing an upper bound to the number of possible operations that can be performed with a single mutation operator.

Rename (syntax). This mutation operator takes a nonterminal of the language grammar and renames it into any other nonterminal of the same grammar. A mutant generated by a mutation operation of this type can cause several different changes to the grammar, such as changing the priority among operators or their associativity, as well as the type of recursion or the type of tokens accepted by a grammar fragment. A grammar can even become ambiguous, which effects were discussed in Sect. 5.4.3. For instance, in a standard *term-factor* grammar [2] renaming the <*factor*> nonterminal into the <*term*> nonterminal causes addition and multiplication to have the same priority.

To generate a mutation operation for the **Rename** operator in a base language with n nonterminals, it means to choose a source nonterminal among the n available, then

to choose a target nonterminal among the $n-1$ remaining ones. Therefore the upper bound in the number of possible first-order mutants of this type is $n(n-1)$.

A **Rename** mutation operation would be implemented by using only glue code in Neverlang as follows. In the example, the grammar of the Expressions language was mutated on line 4 by adding a **rename**: all occurrences of the Factor nonterminal were renamed to Term. The rest of the language implementation is unchanged.

```
1 language Expressions {
2   slices Addition Multiplication
3   roles syntax < evaluation
4   rename { Factor -> Term; }
5 }
```

Attribute Mapping (semantics). This mutation operator takes an attribute of the attribute grammar and maps it to a different attribute in the context of a fragment of the semantics. This can cause several faulty mutants in which the invalid state is caused because a required attribute is missing or replaced by a different one. For instance, mapping the `value` attribute to the `name` attribute in a fragment dedicated to the evaluation of variables may cause the syntax-directed evaluation to forward the name of the variable instead of its value, and eventually to hinder type inference.

To generate a mutation operation for the **Attribute Mapping** operator in a base language, it means to choose an element of the EBNF grammar and then a pair of non-equal attributes of the attribute grammar. The elements of the EBNF grammar are the list of all terminal and nonterminal symbols appearing in all the productions of the grammar. If the EBNF grammar element must be chosen among m elements and the pair of attributes among n elements, the upper bound in the number of possible first-order mutants of this type is $mn(n-1)$.

The attribute mapping would be implemented by using only glue code in Neverlang as follows. In the example, the Variables slice is obtained by composing the VarSyntax syntactic asset and the evaluation role of the VarSemantics semantic asset, however the latter was mutated by remapping the value attribute for the nonterminal in position `$1` to name by adding the mapping on line 4.

```
1 slice Variables {
2   concrete syntax from VarSyntax
3   module VarSemantics with role evaluation
4     mapping attributes { $1.value => name }
5 }
```

Mapping (semantics). This mutation operator makes a fragment of the semantics reference a different nonterminal. This can cause all kinds of unpredictable behaviors, such as swapping a dividend with a divisor in the context of a division.

To generate a mutation operation for the **Mapping** operator in a base language with n slices, it means to choose one of the n slices and then to perform a permutation of the nonterminals present in its grammar fragment. Therefore, the upper bound in the number of possible first-order mutants of this type is $\sum_{i=1}^{n}(m_i!)$, where m_i is the number of nonterminals present in the grammar fragment of the i-th slice.

Mapping would be implemented by using only glue code in Neverlang as follows. In the example, the Division slice is obtained by composing the DivSyntax syntactic asset and the evaluation role of the DivSemantics semantic asset, however the latter was mutated by performing a permutation over the references to the nonterminals in the grammar. In this case, the nonterminal in position $1 was replaced with the nonterminal in position $2 and vice-versa by adding the mapping on line 4.

```
1 slice Division {
2   concrete syntax from DivSyntax
3   module DivSemantics with role evaluation
4     mapping { 1 ⇒ 2, 2 ⇒ 1 }
5 }
```

Duplicate Role (semantics). This mutation operator takes a language feature and duplicates (part of) its semantics—which are called roles in Neverlang—so that they are executed twice. Since in attribute grammars the semantics are stateful and depend on the abstract syntax tree visit order [2], this mutant may cause unpredictable behaviors. For instance, freeing the same pointer twice in C is a reasonable mistake, but one that can cause crashes and heap corruption.

To generate a mutation operation for the **Duplicate Role** operator in a base language with n slices, it means to choose one of the n slices and then to choose which of its roles must be duplicated. Therefore, the upper bound in the number of possible first-order mutants of this type is $\sum_{i=1}^{n} r_i$, where r_i is the number of roles present in the i-th slice.

Role duplication would be implemented using only glue code in Neverlang as follows. In the example, the Free slice is obtained by composing the FreeSyntax syntactic asset with the type-checking and compile roles of the FreeSemantics semantic asset, however the compile role was duplicated, as shown on line 4.

```
1 slice Free {
2   concrete syntax from FreeSyntax
3   module FreeSemantics with role
4     type-checking  compile  compile
5 }
```

Remove Role (semantics). This mutation operator takes a language feature and removes (part of) its semantics. Removing a role can cause some of the grammar attributes not to be properly inherited or synthesized or missing entire code fragments.

Taking on the same example as before, not freeing a memory fragment allocated on the heap is a very common mistake.

To generate a mutation operation for the **Remove Role** operator in a base language with n slices, it means to choose one of the n slices and then to choose which of its roles must be removed. Therefore, the upper bound in the number of possible first-order mutants of this type is $\sum_{i=1}^{n} r_i$, where n is the number of slices in the language and r_i is the number of roles present in the i-th slice.

Role removal would be implemented by using only glue code in Neverlang as follows. In the example, the Free slice is obtained by composing the FreeSyntax syntactic asset with the type-checking role of the FreeSemantics semantic asset. Instead, the compile role was removed, as shown by the red box on line 4.

```
1 slice Free {
2   concrete syntax from FreeSyntax
3   module FreeSemantics with role
4     type-checking compile
5 }
```

Remove Slice (syntax+semantics). This mutation operator removes both the syntax and the semantics of a language feature from the base language. This should usually result in a parsing error for every source program that contains that language feature. A fault of this type may seem more prominent then any of the others we introduced so far. It should never remain unnoticed and the competent programmer should never make such a mistake in the first place. However, this is not always the case in real-world situations and some faults can be very subtle. Small mistakes in the grammar definition of the language are enough to render entire portions of the grammar unreachable. Moreover, failing to properly test more obscure language features and less used operators—such as the shift operators in Java—is not uncommon: if the test suite is not varied enough then the removed slice might never be tested and the corresponding mutant might not be *killed*.

To generate a mutation operation for the **Remove Slice** operator in a base language with n slices, it means to choose one of the n slices to be removed. Therefore, the upper bound in the number of possible first-order mutants of this type is n.

Slice removal would be implemented by using only glue code in Neverlang as follows. In the example, the base Expressions language is made of four language features. However, the language was mutated by removing the RightShift slice from the language as shown on line 5.

Language workbench	Syntax	Semantics	Syntax+Semantics
Spoofax	SDF3 grammar specification	Rules and strategies	Strategies pattern matching
MPS	Editor	Behavior	Concept extension + overriding
Melange	Ecore metamodel	Kermeta aspects	Renaming + aspect extension
MontiCore	Syntax tree node extension	Attribute injection	Associations
Rascal	Abstract data type adapters	Function wrappers	Pattern-based dispatch mechanism

Table 5.2: *Possible targets of a mutation operator class in several language workbenches.*

```
1 language Expressions {
2   slices
3     Addition
4     Multiplication
5     RightShift
6     LeftShift
7   roles syntax < evaluation
8 }
```

Operators in other workbenches. Each language workbench has a different approach to modularization and different workbench capabilities. Therefore, the mutation operators discussed in this section cannot be used by different language workbenches as they are, since some of the concepts are not shared among workbenches. However, the general meta-model should be applicable as long as the workbench supports the separate compilation of its artifacts. In fact, the only difference among two instances of the meta-model applied over two language workbenches should be the chosen mutation operators whereas the same mutation operator *categories* should always be applicable: despite their differences, all language workbenches have their definition of syntactic and semantic artifacts that can be the target of a mutation operation. Therefore, each language workbench should leverage its own peculiarities to implement the meta-model presented in Sect. 5.4.2 and to satisfy the four constraints of the LMP presented in Sect. 5.4.1. This can be achieved by defining different mutation operators that target syntactic artifacts, semantic artifacts or both. Table 5.2 hints at some well-known language workbenches and the artifacts that could be targeted by the mutation operators. However, to the best of our knowledge, no language workbench was previously used to perform mutation testing over language implementations. This list is not meant to be exhaustive and other language workbenches could provide a different solution to the LMP.

5.6 Case Study: ECMAScript Conformance Test Suite

In this section, we assess the Neverlang implementation of the meta-model outlined in Sect. 5.4.2 and detailed in Sect. 5.5, with the goal of answering $\mathbf{RQ}_{5.1}$ and $\mathbf{RQ}_{5.2}$. The SUT will be a family of mutants of a Neverlang implementation of the ECMAScript interpreter obtained by applying the mutation operators introduced in Sect. 5.5.2 on

the language features of the base language. Then, we discuss any threats to the validity of this evaluation and overview the lessons we learned by doing this experiment. This section contributes to this dissertation by showing the applicability of the approach in a concrete scenario and by assessing a set of mutation operators that are compliant to the LMP resolution meta-model and that can be used to evaluate the mutation adequacy of the test suites for Neverlang-based language interpreters.

To answer $\mathbf{RQ_{5.1}}$ we will determine if the mutation operators applied at language feature level are viable: if killing a mutant is trivial, then the corresponding mutation operator might not be significant for the quality assessment of a test suite in a real scenario. The triviality will be measured in terms of the mutation score of the test suite and in terms of the probability with which each test is capable of killing different variants of the mutant family. To answer $\mathbf{RQ_{5.2}}$ we will determine how varied the mutation operators are—*i.e.*, if different mutation operations produce different faulty language variants and therefore they are killed by different tests. Given the set of tests that killed each variant of the family, the similarity between two variants can be measured in terms of the Jaccard similarity between the two sets. A similar assumption was made by Shin *et al.* when they introduced the *distinguishing mutation adequacy criterion* [195] based on the idea that mutants can be distinguished from each other by the set of tests that kills them.

5.6.1 Setup

Hardware setup. All experiments were run on a 64 bits Arch Linux machine with an Intel Core i7-1065G7 3.9GHz processor and a 16 GB RAM. Please note that the hardware setup does not effect this evaluation. However, re-compiling the ECMAScript interpreter with Neverlang takes about 7 seconds on average on this machine. Thus, avoiding to re-compile all the mutants by applying the mutation operators at runtime saves about 7 seconds on each run, for a total of about 2 hours across all 1000 mutants. Different hardware may yield different gains.

Software setup. All the experiments were run using a custom mutation testing framework based on Neverlang 2.2.0 that handles the generation and application of the sourceless mutation operators over the base ECMAScript implementation. The experiment execution was automated using scripts written in GNU bash 5.1.16 and Python 3.10.2.

Data Setup. The base language implementation on which the mutations are performed is a Neverlang implementation of the ECMAScript interpreter [34]—which we did not modify in any way. The data for the evaluation were obtained from the Sputnik ECMA-262 specification conformance test suite[4] used for testing the conformance of the V8 Javascript engine used in Google Chrome. The test suite contains 5538 tests. Since the Neverlang implementation of ECMAScript does not provide any Javascript standard

[4]https://code.google.com/archive/p/sputniktests/

library function, we filtered out all tests that were not compliant with the base language, for a total of 2137 remaining tests.

Experimental Setup. For this evaluation, we generated a family of 1000 different first-order language mutants of ECMAScript by applying a random instance of one of the six mutation operator classes over the base language implementation. The mutants were generated in 30 batches, each with a different random seed. For each mutant, we ran the 2137 selected tests from the Sputnik test suite and stored the result. We also kept track of the class of each mutant using additional meta-data[5] . Finally, we loaded the results into a 2137×1000 matrix, in which position (i, j) was set to 1 if the i-th test killed the j-th mutant and 0 otherwise. All the results reported in Sect. 5.6.2 are obtained by analyzing this matrix.

5.6.2 Results

Let us introduce a notation abuse we will use throughout this section for brevity and better readability: whenever we use the term *mutant class*, we actually refer to the class of first-order mutants generated by the corresponding mutation operators. For instance, the **Remove Slice** mutant class is the class of first-order language mutants that were obtained performing a **Remove Slice** mutation operation over the base ECMAScript implementation. Notice that when we state that two mutants are similar or redundant, we mean that they are likely to be killed by the same tests unless stated otherwise.

RQ$_{5.1}$. A good mutation operator class should produce mutants that find a trade-off in the number of tests capable of killing it. If the number of tests that kill the mutant is too high, then the mutant could be trivial and therefore worthless to evaluate a test suite. Conversely, if no test can kill the mutant then maybe the mutant did not introduce any fault at all, as we introduced in Sect. 5.1 with the mutant equivalent problem. Testing the mutation operators introduced in Sect. 5.5.2 against the Sputnik conformance test suite, we expect the test suite to be able to kill all generated mutants.

Fig. 5.3 summarizes the results of the evaluation. The first row depicts the number of survived mutants with respect to the number of tests in the test suite. The second row depicts the mutation adequacy score of the test suite with respect to the number of tests in the test suite. The left column shows the results divided by mutant class, whereas the right column contains the overall results. We expect the well-known Sputnik test suite to be mutation adequate. In general, a test suite is considered to be *mutation adequate* if the mutation score is 1; this requirement is relaxed when stubborn mutants are present [197]. The results meets our expectations and the test suite scores a mutation adequacy score of 1 when considering all the mutants and all the tests of the test suite. However, in this study we are not interested in the evaluation of the test suite, which we assume to be reliable. Instead, we evaluate the mutation operators: the mutation score is used as an indicator of how much instances of a mutant class

[5] The dataset containing all the results is available at `https://doi.org/10.5281/zenodo.7024829`.

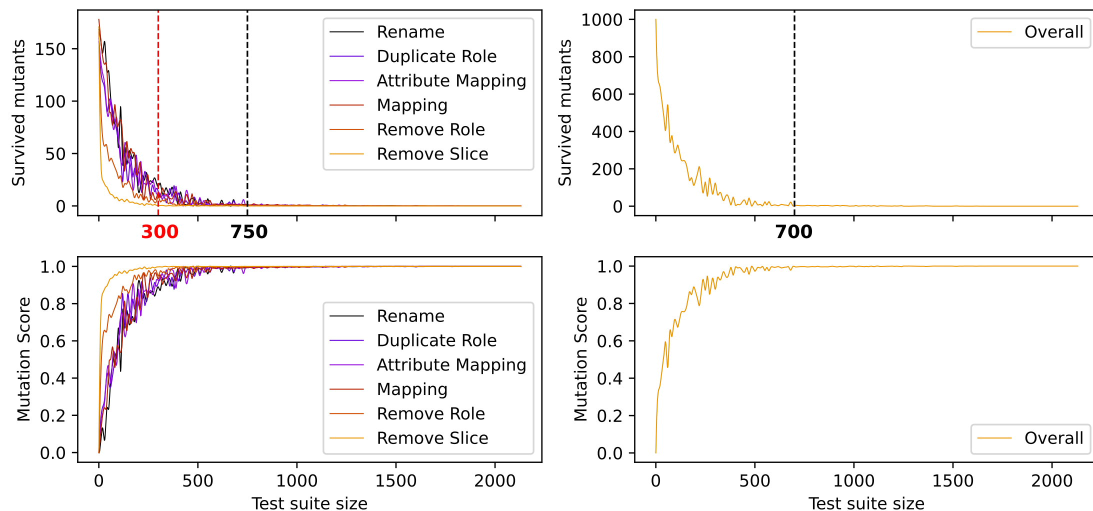


Figure 5.3: *Number of survived mutants and mutation score for each of the considered mutation operators.*

are hard to kill. The higher the mutation score, the easier the instances of a class are to kill; a lower mutation score is desirable because it means that the mutant is harder to kill. Therefore, Fig. 5.3 also shows the effect that reducing test suite size[6] has over the number of survived mutants and the corresponding mutation score. All classes show similar results: the test suite is capable of consistently killing most of the mutants and to achieve a mutation score of 1 when the number of tests is greater then 750—as shown by the black dashed line in Fig. 5.3. The only exception is the **Remove Slice** class, that can be killed consistently by a smaller test suite—with less than 300 tests, as shown by the red dashed line in Fig. 5.3. When considering all classes, most mutants are killed by a test suite of more than 700 tests overall and the mutation score is consistently 1 at over 1500 tests.

Fig. 5.4 and Fig. 5.5 focus on different aspects of this evaluation. Fig. 5.4 highlights the average probability with which each individual test was capable of discovering and killing a mutant of each class. The brighter the color, the higher the probability. There are tests that scored a very high probability on all mutant classes, as shown by the clear vertical stripes in Fig. 5.4; these tests fall under DeMillo *et al.*'s definition of *coupling effect*: «test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors» [62]. Fig. 5.4 visually confirms the results from Fig. 5.3: the **Remove Slice** mutants are the easiest to kill and **Rename** are the hardest to kill, with the other classes ranging in between. In Fig. 5.5, each box does not only represent the median probability of killing a mutant of each class, but also the locality, spread and skewness of the results. Again, **Remove Slice** has the highest median (0.44) and **Rename** has the lowest one (0.00). The outliers for each class match with the tests with high killing

[6] For each test suite size on the x axis, the corresponding y value was calculated over 30 random runs.

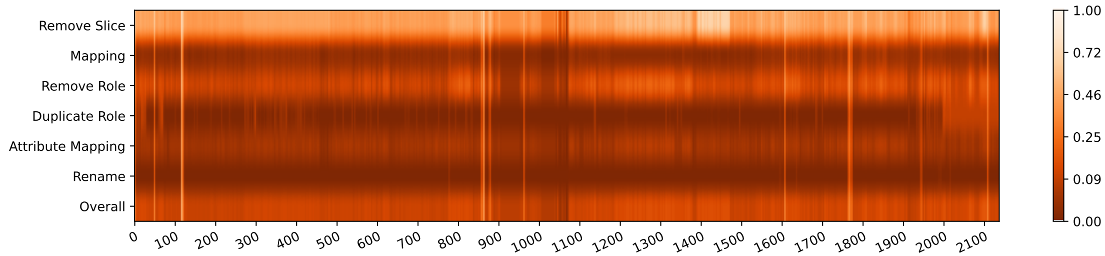


Figure 5.4: *Probability with which each selected test kills a mutant of each class.*

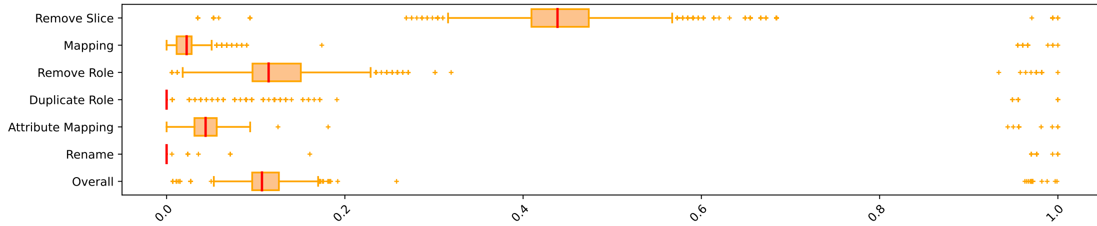


Figure 5.5: *Box plot representing the locality, spread and skewness of the results for each of the six Neverlang sourceless language mutation operators.*

probability from Fig. 5.4. Otherwise, an higher median probability is always matched to higher quartile coefficient of dispersion for all classes. We do not report all these results in a table to improve readability by avoiding redundancy: all these pieces of information were directly obtained from Fig. 5.5 and add very little to this evaluation. Please refer to the companion dataset[9] for the results of all tests.

RQ$_{5.2}$. Recall that each column of the 2137×1000 matrix we introduced in Sect. 5.6.1 is a binary array. In this context, each mutant can be seen as a set M whose indicator function is the corresponding column of the matrix:

$$\mathbb{1}_M(t) = \begin{cases} 1 & \text{if } t \text{ killed } M \\ 0 & \text{otherwise.} \end{cases}$$

By extension, given the test set T:

$$M = \{t \in T \mid t \text{ killed } M\} .$$

Now, given the sets for two mutants M_1 and M_2 defined as above, the Jaccard similarity coefficient (also known as Jaccard index) between the two sets is calculated as:

$$J(M_1, M_2) = \frac{|M_1 \cap M_2|}{|M_1 \cup M_2|} .$$

Fig. 5.6 graphically shows the Jaccard similarity coefficient between each pair of mutants, divided by mutation operator class. The brighter the color, the higher the coefficient

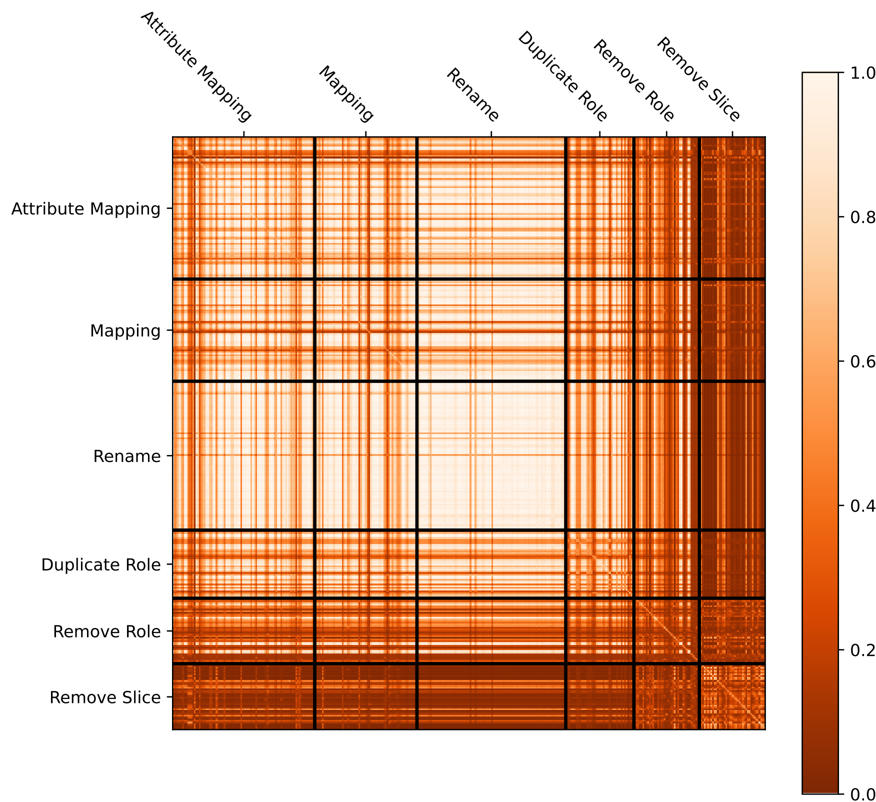


Figure 5.6: *Average Jaccard similarity coefficient for all mutation operator classes.*

and therefore the corresponding mutants obtained similar results—*i.e.*, they were killed by a similar set of tests. Notice that of course the matrix is symmetric and all the values on the diagonal are equal to 1 by construction. Fig. 5.6 reports the mutation operator classes on the two axes. The **Remove Slice** and **Remove Role** are the classes that show the lowest similarity coefficient, both within the class and with instances of other classes. The other classes have higher similarity with each other, but overall Fig. 5.6 shows that the similarity between mutants of different classes is usually low despite the presence of some clusters. Instead, Fig. 5.7 isolates each of the six classes by highlighting in large detail the sub-matrices along the diagonal of Fig. 5.6 with the same color scale. Each sub-matrix represents the similarity between mutants of the same class. In this case the results show that the similarity coefficient within the **Rename** and **Mapping** classes is high and that the different mutation operator instances from these classes might be redundant with each other since they are killed by the same tests.

5.6.3 Discussion

$RQ_{5.1}$. A viable mutation testing approach must produce mutants that can be tested without being killed: if killing a mutant is too easy, then that mutant is not useful to

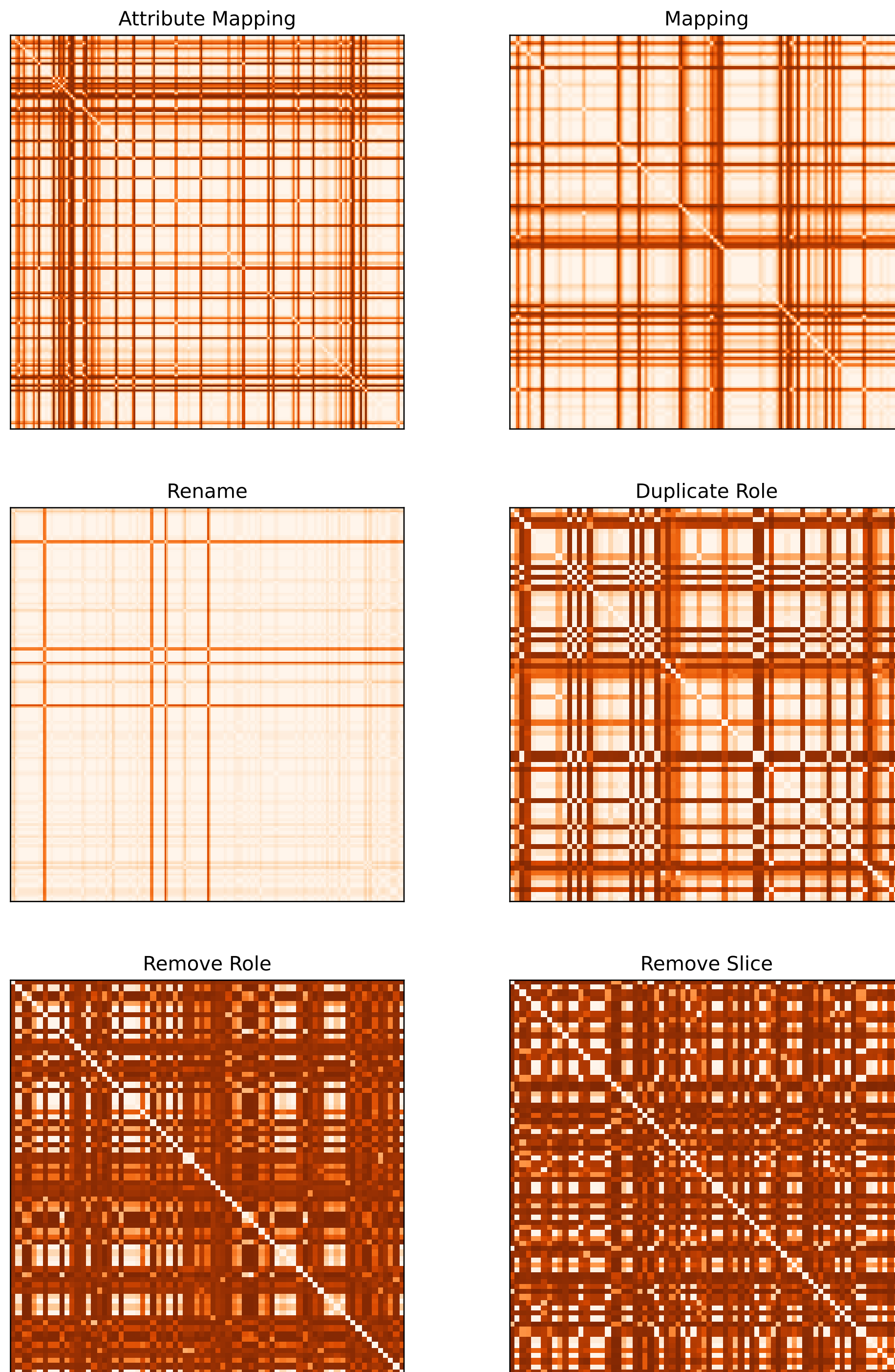


Figure 5.7: *Average Jaccard similarity coefficient for each mutation operator class.*

improve the quality of the test suite and artificially skews the mutation score towards 1. Fig. 5.4 shows that each test can kill most variants of the mutant family with a probability below 10%—with the exception of those from the **Remove Slice** class. The mutation operators could be refined in a future work to produce variants with subtler faults, but this would require a different approach in which some initial knowledge over the base language is available, a requirement that is not general enough to be able to solve the LMP. The current approach already shows that non-trivial mutants can be created using this general meta-model, without compiling source code, without an intermediate representation and without initial knowledge over the SUT. Based on our evaluation and according to Fig. 5.3, consistently killing a language mutant requires a test suite of about 700 or more tests. The box below shows a concise answer to **RQ$_{5.1}$**, although it should be noted that this evaluation addresses **RQ$_{5.1}$** for one specific case study and the results may differ in a different scenario.

> **RQ$_{5.1}$ Which Neverlang sourceless mutation operators produce variants of the language mutant family that are reasonably hard to discover and kill?**
>
> **Rename**, **Duplicate Role**, **Attribute Mapping**, **Mapping** and **Remove Role** mutation operators produce mutants that are reasonably hard to discover and kill. Most **Remove Slice** mutants can be easily killed and should be refined or substituted with different operators.

RQ$_{5.2}$. The **Attribute Mapping**, **Duplicate Role**, **Remove Role** and **Remove Slice** mutation operators produce mutants that are discovered by set of tests with very low similarity. The similarity within the **Rename** and **Mapping** mutant classes is very high according to Fig. 5.7. Therefore two different mutation operator instances of the same class are very likely to be redundant and hard to distinguish [195]. However, the similarity between a **Rename** mutant and mutants of any other class is low. The same goes for the **Mapping** class. To summarize, the classes of mutation operators we introduced in this work do not form a partition: mutants whose similarity with other mutants of the same class is low, usually also have low similarity with mutants in other classes and vice versa. The box below shows a concise answer to **RQ$_{5.2}$**, although it should be noted that this evaluation addresses **RQ$_{5.2}$** for one specific case study and the results may differ in a different scenario

> **RQ$_{5.2}$ Are mutation operator classes from different categories producing different mutants? Can we obtain similar results by reducing the number of classes?**
>
> The similarity between different classes is low as shown in Fig. 5.6: different classes are usually killed by different tests, Therefore we cannot obtain similar results by reducing the number of classes. However, the similarity within each class could be reduced considerably, as shown in Fig. 5.7.

5.6.4 Lessons Learned

By performing this evaluation, we learned that using a language workbench that is fully compliant with the LEP eases the implementation of a mutation approach that solves the LMP. In the context of Neverlang, we could implement six different mutation operator classes using pre-existing API and then generate 1000 random mutants to create a LPL of language mutants. The evaluation shows that non-trivial mutation operators can be defined and assessed without changes to the original language workbench, by leveraging existing composition mechanisms between language features. This duality between language features and language mutation is made apparent by the definition of the LMP as a derivation of the LEP: language extension and language mutation are similar problems that can be tackled in a similar way. In this study, we learned that designers should be concerned with the LMP during the early stages of development to better drive the development of language workbenches. Due to the relation between LMP and LEP, the same tools that are used to perform language mutation can also be used to accommodate the composition between language features and therefore to produce standard LPLs. This should in turn improve the reusability of existing language features due to flexible composition mechanisms—*i.e.*, through well-designed custom mutations over the original feature. In fact, the main limitation of the application of the resolution meta-model to Neverlang and of the overall evaluation is the problem of granularity: most mutation testing approaches from literature work at fine granularity (statement level). Instead, Neverlang composition mechanisms work at a coarse granularity: that of language feature. Foreseeing the LMP during the design of Neverlang would have allowed for the definition of more fine-grained mutation operators. With this work, we learned that language workbenches should strive to achieve the best of both worlds: both the LMP and the LEP should be solved using composition mechanisms that work at feature level but that allow to tweak the semantics at a fine granularity level. Otherwise, it may be still beneficial to combine this approach with traditional mutation testing: our approach can be used first to save on the recompilation time. Then, once the test suite is mutation adequate against sourceless mutation operators, fine-grained traditional mutation operators can be used to introduce subtler faults.

5.6.5 Threats to Validity

Internal Validity. To evaluate the mutation testing approach, we had to develop a custom mutation testing framework for Neverlang: this may affect the results of the evaluation. To stem this validity issue, our framework does not introduce any additional API and it is only used to automate the generation of random mutants. This is done with the Random class from the standard Java library. We also kept track of all the seeds we used in each batch of experiments to ensure they could be replicated. The seeds themselves were chosen randomly using the `$RANDOM` bash function. To avoid any further internal validity issues we used a pre-existing ECMAScript implementation that we did not modify in any way and the well-known Sputnik test suite. We did

not inspect the source code of the test suite; instead the selection was performed automatically by filtering any tests that were not compliant with the base language. In Fig. 5.3, the selection of a subset of the test suite was also performed at random: we performed 30 random runs for each test suite size and measured the average adequacy score. Selecting only test suite subsets may have reduced the coverage of the test suite, however, this work does not want to evaluate the test suite but rather test the applicability of the approach. We think that the evaluation of a very complete and mutation adequate test suite would yield way less information with regards to used mutation operator classes. Rather, evaluating an incomplete test suite by means of the mutation score is more representative of a real-world application of mutation testing in which the evaluated test suite is the result of an ongoing development process.

External Validity. We implemented a mutation testing approach based on Neverlang and using a specific set of mutation operator classes. Then, we evaluated this specific implementation. The evaluation we performed may therefore not be applicable to other research settings. To stem this threat to validity we specified the LMP without reliance to any Neverlang-specific concepts. Instead, the LMP is an instance of the LEP, which was a pre-established general problem defined by a third party. Moreover, all four constraints of the LMP are general: they use concepts that are applicable to all programming language implementations, such as syntax, semantics, source code and compilation. Despite not being general to all programming language implementations, the goal of the LMP is general to all LPL approaches since it only relies on the concept of language family. Similarly, we outlined a resolution meta-model (Sect. 5.4.2) that is based on the same concepts and not limited to Neverlang. The only limitation is that the mutation operators are Neverlang-specific and cannot be used in other language workbenches due to their differences in the modularization approach. Nonetheless the implementation presented in Sect. 5.5 and the evaluation of Sect. 5.6 prove the applicability of the meta-model to a real language workbench and show the mutation testing process of a real language implementation in such a workbench. We also detailed our evaluation process so that it can replicated for the evaluation of different mutation operators in other language workbenches. Finally, we addressed the general applicability in Table 5.2, which reports the concepts from other language workbenches that could be used as a target for the mutation operators. Yet, the main threat to the external applicability of our contribution is that, to the best of our knowledge, no other language workbench fully supports separate compilation and runtime adaptability. Similarly, not all language workbenches support LPLs explicitly, which is a primary concern when considering that the goal of the LMP is to create an LPL of language mutants. Therefore, most language workbenches may not be able to satisfy the *separate compilation* and *independent mutability* constraints to solve the LMP. However, the importance of these aspects was already discussed by Leduc *et al.* [137] in a context more general than mutation testing. Therefore, any language workbench that wants to solve the LEP must already satisfy these constraints regardless they also want to solve the LMP or not. Moreover, we advised alternative solutions, such as generating

the entire family of mutants in advance, a solution that is applicable—although not advised—to closed mutant families.

5.7 Summary of Chapter 5

Mutation testing in the context of language implementations lacks the proper focus and it is usually dedicated to the generation of test suites, rather than to their evaluation. In this chapter, we specified this problem as a derivation of the *language extension problem* [137], dubbed *language mutation problem*. Then, we proposed a solution based on language product lines and language workbenches that satisfies four different constraints of the *language mutation problem*, using Neverlang as a running example. Finally, we performed an empirical evaluation to answer $\mathbf{RQ}_{5.1}$ and $\mathbf{RQ}_{5.2}$ and to demonstrate the applicability of the approach. The results show that a set of non-trivial mutation operators can be defined using existing technologies, although with limitations related to the used language workbench and its capabilities with regards to introspection, intercession and separate compilation.

6 Related Work

SPLs and LPLs are an increasingly popular trend among researchers and practitioners due to their promises of portability and mass customization. The SPL community has devoted huge efforts towards the development of feature-oriented tools, techniques and methodologies for SPLs and LPLs. The goal is achieving a significant improvement in the quality of the products, avoiding the *clone-and-own* approach in favor of reuse of carefully crafted software assets. Whether this result is actually achieved is debatable: high levels of quality are the result of the application of a proper design methodology and must be assessed through effective evaluation strategies. The main goal of this dissertation is to outline the dimensions that concern the quality of SPLs and LPLs in particular, including their construction and their evaluation. This work is founded on the contributions brought by researchers in this field, as it was discussed in Chapter 2. However, there are further contributions that, albeit not being directly referenced in this dissertation, provide a related contribution, either because they use similar techniques or because they try to solve similar problems. In this chapter, we provide a brief and possibly non-exhaustive introduction to such works.

6.1 Design Patterns for the Definition of SPLs

Chapter 3 of this dissertation deals with the complexity of creating and managing SPLs, as well as with the problem of compatibility among different approaches.

A huge variety of approaches to support the definition of SPLs have been proposed by researchers [17, 158, 124]. However, all these approaches are based on preprocessors and, to the best of our knowledge, there has been little research on design patterns specifically suited to variability modeling and the implementation of SPLs. In Sect 3.3, we discussed VMJ as a purely architectural pattern-based solution to describe variability of SPLs through Java modules and delta-oriented programming [192]. In this approach, each feature is implemented through decorators [82] over the base implementation. Seidl *et al.* [191] presented a generative SPL development method using variability-aware versions of the observer, strategy, template method and composite [82] patterns and introduced the Family Role Model as a notation to capture constraints on the variable application. Shatnawi and Cunningham [193] addressed the difficulty of specifying and maintaining feature models due to the SPLE tools requiring specific knowledge and skills and they proposed to encode FMs using JSON. Their contribution shares with the devise pattern the design choice of using mainstream technologies

rather then external composers to develop SPLs. On a similar note, Chimalakonda and Lee [47] discussed the inconsistency and incompatibility of tools and methods in SPLs and the need for the introduction of standards in their development. They argue that the diversified range of tools and methods is one of the primary hindrances to the adoption of SPLs in the industry, since artifacts developed with one suite are not compatible nor reusable with other ones.

6.2 Language Product Lines Design Methodology

Chapter 5 of this dissertation deals with several quality aspects in the design of LPLs, including the IDE support for their development and their evaluation.

LPL engineering with IDE support. Most recent language workbenches [71]—such as, Spoofax [221], MPS [217], MontiCore [126], LISA [97] and Melange [61]—provide *tools for system designers* by addressing the problem of IDE support for modular DSLs. Some of these approaches generate an IDE using templates, thus neglecting feature modularity and the specific characteristics of the DSL under development. EMF-based tools [199] such as EMFText [95] support modular language implementation and IDE generation for DSLs. EMFText is similar to Neverlang since it uses attribute grammars to share IDE implementations through languages, however it does not explicitly consider language variability for LPL development. 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.* [28] 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 including syntax highlighting, code-completion and parse error recovery, but also the creation of language configurations. Liebig *et al.* [139] used Spoofax alongside FeatureHouse for the representation and composition of language features. Mendez-Acuña *et al.* presented several contributions to the topic of reuse in language workbenches: in [108] the authors address the problems of programming languages evolution and maintenance, as well as their verification and validation; in [152] 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 [53], used by language designers to build and compose DSLs and by domain designers to coordinate their models. MPS offers full IDE support and customizable abstract syntax tree manipulation. Most notably, mbeddr [218] is a project built on top of MPS, presented as a set of integrated and extensible languages based on C for embedded software engineering with an IDE and support for SPL development. MPS arguably represents the most fully realized version of a development environment with support for the development of language families. However, 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. Moreover, to the best of our

knowledge, there is no contribution for a dedicated bottom-up LPL engineering process supported by a development environment and covering all involved roles and all the evolution phases of an incremental LPL, including creation, development of language artifacts, configuration and deployment. Other works focus instead on one or a few of said aspects and embrace LPL engineering in a top-down fashion.

Specification techniques. Thüm *et al.* [205] 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 postponed to future work. On the same topic, CBS [160] provides an extensible library of reusable language specification components based on fundamental programming constructs (funcons).

Design flaw detection. 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 [4] and MoSo-PoLiTe [166] 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.* [175] apply automated generation of test products using the t-wise SPL test adequacy criteria. Several works [11, 99, 212, 228] evaluate SPLs using structural or service utilization metrics and Aldekoa *et al.* [5, 6] 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*.

6.3 Mutation Testing of Language Implementations

Chapter 5 of this dissertation deals with several different topics, including adaptable languages, compiler testing and mutation testing.

Adaptable Languages. Cazzola *et al.* [32] used micro-languages to evolve an interpreter at runtime through a micro-dynamic adaptations (μDA) domain-specific language. μDA adaptations are similar to the mutation operators introduced in Chapter 5. Kollár and Forgáč [122] presented an adaptive approach to both program and language modification to support dynamic evolution. More recently, Jouneaux *et al.* [110] proposed the concepts of *self-adaptable languages* and the *L-MODA* conceptual reference framework

that abstract the design, execution and feedback loop of self-adaptable systems. Yet, to the best of our knowledge adaptable languages were never used to avoid the cost of recompilation in any mutation testing approaches.

Mutation Testing. Literature proposed mutation approaches that perform at low levels of abstraction. In these approaches, mutations are usually applied at compiler intermediate representation level. The goal is usually to provide multi-language tools, as opposed to source level (language-specific) mutation approaches. The LLVM [135] framework is usually the target of the mutation. Some examples are SRCIROR [92], Mull [63] and the contributions from Sousa and Sen [198] and from Papadakis *et al.* [168]. Similarly, JAVALANCHE [190] and PIT [51] manipulate Java bytecode directly to avoid the cost of recompilation. Mutation testing in the context of SPLs often relies on model-based mutation operators [133, 96] rather than using the composer to create mutated products as in our approach. To the best of our knowledge, none of these approaches perform the mutation operators directly at runtime. Sect. 6.3.1 discusses the state-of-the art of mutation testing in more detail, including a comparison between the LMP resolution meta-model and other existing approaches.

Language Testing. Chen *et al.* [42] performed a survey of the field of compiler testing. According to their classification, our approach falls under the *non-semantics-preserving mutation* approaches category. The authors identified five different approaches in this category. Nagai *et al.* [161], tested the validity of C compilers using randomly generated programs under the assumption that longer expressions are more likely to induce undefined behavior. Chen *et al.* [44] used Markov Chain Monte Carlo sampling to select mutations with higher chance of triggering compiler bugs. Holler *et al.* [101] replace random nonterminals in test cases with expansions of the same nonterminals according to the language grammar. Garoche *et al.* [84] take a complete test suite as input and mutate it to produce more failure-inducing programs. Groce *et al.* [89] propose a similar approach, but with the added goal of achieving some desired property of the mutated test suite, such as reducing its size while keeping the same coverage. It should be noted that these approaches perform mutations over the test suite rather than on the language implementation and are therefore suited to different use cases, such as generating a test suite, improving coverage and detecting more faults. Instead, the LMP resolution meta-model modifies the language implementation directly and is therefore more suited to test suite quality assessment. Moreover, none of these contributions involves LPLs to the best of our knowledge. Please refer to the aforementioned survey for a full overview on the topic of language testing approaches, including those from different categories.

6.3.1 Comparing the LMP with other Mutation Testing Techniques

Mutation testing is intuitively expensive because it requires to run many tests against many mutants. Thus, we can distinguish between two kinds of approaches that try to

Approach	Type	Benefits	Limitations	Compatibility
LMP meta-model	Mutant execution	Avoids recompilation cost	Specific to language interpreters and requires paradigm shift	–
Greedy algorithms [106, 105]	Test suite reduction	Improves recurring test suite execution	All mutants must be generated, compiled and linked	✓
Test prioritization [227, 226]	Test suite reduction	Improves recurring test suite execution	All mutants must be generated, compiled and linked	✓
Compiler integration [111]	Mutant execution	Nice trade-off between source code and bytecode	All mutants must be generated at the same time	✗
Sufficient operators [162, 12, 196]	Mutant execution	Prevents the generation of redundant and equivalent mutants	Sufficient operators must be defined for each language on a case-by-case basis	✓
Predictive mutation testing [225]	Mutant execution	Predicts mutation testing results without executing mutants	The effectiveness depends on the quality and the size of the training set	✓
Second order mutants [178]	Mutant execution	Number of mutants is almost halved	All mutants must be generated at the same time	✓
Evolutionary algorithms [66]	Mutant execution	Generation of effective and hard to kill mutants	The entire test suite must be executed against all the mutants	✓
Weak mutation testing [103]	Mutant execution	Much less test suite execution is required	Less effective than strong mutation testing and not viable for critical applications	✓

Table 6.1: *Comparison among mutation testing approaches. For each approach, the table summarizes its benefits and limitations, as well as its capability to be used jointly with the LMP resolution meta-model.*

reduce the mutation testing cost:

1. approaches that save on the cost of running the test suite;
2. approaches that save on the cost of generating or executing mutants.

The meta-model discussed in Chapter 5 falls under the second category. In this section, we discuss several approaches from each of the two categories and compare them with our LMP resolution meta-model. This comparison is summarized in Table 6.1.

Saving on test suite execution. For each mutant that cannot be killed, the entire test suite is executed. For each mutant that can be killed, several tests are potentially executed before running one that actually kills the mutant. The problem of minimizing a test suite has been shown to be NP-hard [83], although there are approaches that try to save on test suite execution with greedy algorithms [106, 105] and test prioritization techniques [227, 226]. Compared to the LMP resolution meta-model, these techniques have the drawback that performing a test selection still requires generating, compiling, and linking all the mutants to execute them against the reduced test suite. However, they are beneficial in the long run, since the reduced test suite can be used to perform recurring builds.

Saving on mutant execution. Approaches that try to reduce the cost of generating, compiling and executing several mutants are more numerous and diverse.

Compiler integrated mutation testing frameworks such as Major [111] directly modify the program's abstract syntax tree to avoid modification of the source code or of an intermediate representation and to allow for specific optimizations. This trade-off gives access to more semantic information while preventing the mutation of desugared code. However, since the mutation is integrated in the compiler, all of the mutants have to be generated at once to avoid the cost of recompilation.

Several contributions investigate selective mutation using sufficient mutation operators [162, 12, 196] to avoid the generation of redundant mutants and equivalent mutants. Equivalent and subsumed mutants can also be prevented using refinement relations over the model of two different mutants [13, 14]. This prevents wasting resources and skewing the mutant adequacy score. In fact, research has shown that—even for a random selection of the mutants—a 100 percent mutation adequacy for 10 percent of the mutants is nearly adequate for a full mutation analysis [179]. This can be improved upon using sufficient operators. However, the suitability of mutation operators might depend on the programming language [179] and therefore a new set of sufficient operators must be defined for each language.

Predictive mutation testing [225] reduces the effectiveness but improves the efficiency of mutation testing by not executing the mutants at all. Instead, it uses machine learning techniques to predict the mutation adequacy score: the first versions of a program and its mutants are used as a training set, then the classification model is used to predict whether any new mutant is killed or survived based on the same feature used to train the model. The main limitation to this approach is its dependency to the training set: a bigger training set can improve the effectiveness of this approach but it might reduce its efficiency. Conversely, a smaller training set may cause excessive effectiveness loss.

Following a different approach [178], two sets of first-order mutants can be combined to produce a set of second-order mutants to reduce the number of equivalent mutants, which is usually relatively high for first-order mutants. Although the application of mutation operators is not necessarily commutative, the same approach can be used to almost halve the number of mutants in a set depending on the algorithm. However, this technique requires that all of the first-order mutants are generated in advance and

also the execution of an additional algorithm to produce the second-order mutants.

Evolutionary mutation testing [66] is a technique based on genetic algorithms that measures the usefulness of a mutant according to a fitness function (the execution matrix) to produce less but more effective mutants. The algorithm favors equivalent and difficult to kill mutants and penalizes set of mutants that are killed by the same tests. The problem with this approach is that the entire test suite must be executed on all of the mutants on each generation to measure the fitness function even if a mutant has already been killed.

Weak mutation testing [103] is a well-known technique to reduce the cost of mutation testing. Using weak mutation testing, much less program execution is required, since the result can be determined prior to the test completing its execution; instead, a mutant can be killed as soon as it causes an internal state that differs from the internal state of the base program. This technique is usually considered to be less effective than the traditional strong mutation testing (hence the term "weak"), although empirical evidence shows that it can be used as an effective alternative of non-critical applications [163]. Moreover, it requires additional infrastructure to be able to inspect the execution state at any time.

Compatibility with the LMP. Despite their differences, each of the approaches presented in this section work at different levels of abstractions and can often be used jointly. Some works, such as [178], already discussed the compatibility between their approach and other techniques in literature. In fact, the LMP resolution meta-model is compatible with most of the presented approaches: using them in conjunction would allow to leverage the strengths of both techniques to further improve on the cost reduction. Since the meta-model does not manipulate the test suite, it is compatible with any test suite reduction technique: the main limitation of the latter is that all mutants must be generated and compiled even for recurring execution of the test suite. This limitation could be avoided combining test reduction techniques with the meta-model. Any application of the LMP resolution meta-model would benefit by the definition of sufficient operators to be used in the process depending on the language workbench. Predictive mutation testing has no requirements over the underlying architecture and could be used to predict the execution results of mutants generated using the LMP resolution meta-model. However, it should be noted that the latter requires a shift in the programming paradigm and therefore the predictive mutation testing technique may not be applicable due to lack of a suitable training set. All the algorithms used to generate second-order mutants discussed in [178] can be used over mutants generated by our meta-model; a combination of the two approaches would even be more beneficial since it prevents the necessity of generating all the mutants in advance. Evolutionary algorithms could be combined with the LMP resolution meta-model to produce hard to kill mutants; moreover, Domínguez-Jiménez *et al.*'s work [66] also uses a fitness function based on a matrix that is very similar to the one we used to evaluate the LMP (Sect. 5.6). Finally, weak mutation testing can be combined with the LMP resolution meta-model thanks to the language workbenches capabilities of automatically generat-

ing a debugger for any mutant [131]. The debugger can then be used to inspect the internal execution state: if it differs from that of the base language then the test suite execution is terminated. According to our comparison and as reported in Table 6.1, the only approach that is not compatible with the LMP resolution meta-model is the integration of mutation operators inside the compiler, because in the meta-model the compiler is executed only once at the beginning of the process, whereas mutants are generated later at runtime. However, it should be noted that the meta-model is based on the capabilities of the language workbenches, that can be used to create language interpreters and compilers with integrated mutation support.

7 Conclusions

Language-oriented programming is a paradigm that can help bringing the development activity to a wider audience, by involving domain experts through tailored and more powerful abstractions to improve productivity and software quality. However, language-oriented programming was ultimately deemed unsuccessful due to the innate complexity of the language development activity. While modular development and modern language workbenches can smooth the creation and maintenance of DSLs, we argue that further improvement can be achieved by considering all the dimensions of variability modeling.

In this dissertation, we discussed the dimensions we deem to be the most relevant by presenting a design technique for SPLs in general and then focusing on LPLs in particular to outline all the elements of a design methodology for language decompositions and language families. A primary concern when designing LPLs is asserting the desired properties of all software assets. The assessment must consider both the language features, whose design errors can be detected through the measurement of specific metrics, and the test suites, whose adequacy can be evaluated using mutation testing techniques. This cycle of continuous development and quality assessment constitutes an iterative engineering process suited for the development of families of DSLs in an agile context. Such an approach is enabled by meeting several requirements: i) high degree of communication between the actors involved in the process (developer, deployer and user), ii) low overlap in the artifacts that must be accessed concurrently through separate compilation, iii) adequate tools for system designers and iv) a faster mutation testing process that does not encompass re-compilation. In this dissertation, we expressed and addressed these dimensions by means of our research questions **RQ**$_{4.1}$, **RQ**$_{4.2}$, **RQ**$_{5.1}$ and **RQ**$_{5.2}$. By answering **RQ**$_{4.1}$, we defined the properties of a well-defined language decomposition—*i.e.*, low complexity and high cohesion and maintainability. By answering **RQ**$_{4.2}$, we defined metrics to detect inconsistencies between the desired properties of the language decomposition and the actual implementation of language components. By answering **RQ**$_{5.1}$ and **RQ**$_{5.2}$, we defined and assessed mutation operators that can be used to evaluate the quality of the test suite the language implementations are tested against. These results combined offer tools and techniques to evaluate all phases of the development of an LPL, from conception to implementation, testing and deployment.

Although this dissertation was limited to an overview of these dimensions, this list is not exhaustive and several more dimensions should be considered in future works. The devise pattern presented in Chapter 3 lacks support for multi-dimensional

feature models and an empirical testing involving dynamic SPLs with model-based dynamic software updating. The design methodology presented in Chapter 4 still lacks dedicated specification techniques—*i.e.*, point 4 of the design methodology according to Parnas. The future work in this regard will involve the integration of theorem proving techniques in the Neverlang LPLs, as well as a tool-supported language configuration process that takes the usability of language features into account when suggesting valid completions to the language deployer. The usability can be evaluated directly by the language user and then used to update the FM based on this feedback, thus increasing the amount of communication between the actors. We also plan on providing further support for the evaluation of software artifacts during each iteration of the engineering process by considering the well-definedness of the attribute grammar [113] as a desirable property of the language decomposition. Moreover, the LPL development environment should be improved with regards to usability and performance and it should shift from being Eclipse-based to a platform-agnostic approach, for instance by applying the language server protocol and the language server index format[1].

Another dimension that can affect the quality of an LPL is that of the *granularity* of language composition mechanisms. Current taxonomies focus on composition at language level [70], but languages are too coarse grained to support language evolution with a satisfying degree of reuse. Neverlang supports composition between modules to yield slices and between slices to yield languages, but such composition mechanisms lack a proper formal classification and can still be considered too coarse grained. Based on a more refined taxonomy, the design methodology can be reviewed with a new configuration process that takes all levels of granularity into account, so that the variability space of the language family allows for even more valid variants.

The sentiment that drives all these future works and that was shared by this entire dissertation is the need to support an ever-growing degree of flexibility and, eventually, to satisfy the innate diversity of human needs while maintaining high levels of reuse, productivity and software quality.

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

Bibliography

[1] Hadil Abukwaik, Andreas Burger, Berima Kweku Andam, and Thorsten Berger. Semi-Automated Feature Traceability with Embedded Annotations. In Foutse Khomh and David Lo, editors, *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*, pages 529–533, Madrid, Spain, September 2018. IEEE.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, USA, second edition, 2006.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, 1986.

[4] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malter Lochau, and Gunter Saake. IncLing: Efficient Product-Line Testing Using Incremental Pairwise Sampling. In Ina Schaefer, editor, *Proceedings of the 15th International Conference on Generative Programming and Component Engineering (GPCE'16)*, pages 144–155, Amsterdam, The Netherlands, October-November 2016. ACM.

[5] Gentzane Aldekoa, Salvador Trujillo, Goiuria Sagardui Mendieta, and Oscar Díaz. Experience Measuring Maintainability in Software Product Lines. In José Cristóbal Riquelme Santos and Pere Botella, editors, *Proceedings of the XI Jornadas de Ingenieria del Software y Bases de Datos (JISBD'06)*, pages 173–182, Barcelona, Spain, October 2006.

[6] Gentzane Aldekoa, Salvador Trujillo, Goiuria Sagardui, and Oscar Díaz. Quantifying Maintainability in Feature Oriented Product Lines. In Christos Tjiortjis and Andreas Winter, editors, *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08)*, pages 243–247, Athens, Greece, April 2008. IEEE.

[7] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-oriented software product lines*. Springer, 2016.

[8] Sven Apel, Don Batory, Christian Kästner, and Gunter Saale. *Feature-Oriented Software Product Lines*. Springer, April 2013.

[9] Sven Apel, Christian Kästner, and Christian Lengauer. Language-Independent, Automated Software Composition. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 221–231, Vancouver, BC, Canada, May 2009. IEEE.

[10] Cyrille Artho, Klaus Havelund, Rahul Kumar, and Yoriyuki Yamagata. Domain-Specific Languages with Scala. In Michael Butler, Sylvain Conchon, and Fatiha Zaïdi, editors, *17th International Conference on Formal Engineering Methods (ICFM'15)*,

Lecture Notes in Computer Science 9407, pages 1–16, Paris, France, November 2015. Springer.

[11] Ebrahim Bagheri and Dragan Gasevic. Assessing the Maintainability of Software Product Line Feature Models Using Strutural Metrics. *Software Quality Journal*, 19(3):576–612, January 2011.

[12] Ellen Francine Barbosa, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. Toward the Determination of Sufficient Mutant Operators for C. *Journal of Software: Testing, Verification and Reliability*, 11(2):113–136, May 2001.

[13] Davide Basile, Maurice H. ter Beek, Maxime Cordy, and Axel Legay. Tackling the Equivalent Mutant Problem in Real-Time Systems: The 12 Commandments of Model-Based Mutation Testing. In Philippe Collet and Sarah Nadi, editors, *Proceedings of the 24th International Software Product Line Conference (SPLC'20)*, pages 252–262, Montréal, Canada, October 2020. ACM.

[14] Davide Basile, Maurice H. ter Beek, Sami Lazreg, Maxime Cordy, and Axel Legay. Static Detection of Equivalent Mutants in Real-Time Model-Based Mutation Testing. *Empirical Software Engineering*, 27, 2022.

[15] D. Batory. *Automated Software Design Volume 1*. Primedia eLaunch LLC, 2020.

[16] Don Batory, Rich Cardone, and Yoannis Smaragdakis. Object-Oriented Frameworks and Product Lines. In P. Donohoe, editor, *Software Product Lines*, SECS 576, pages 227–247. Springer, 2000.

[17] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004.

[18] Maurice H. ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini. Static Analysis of Featured Transition Systems. In Laurence Duchien and Thomas Thüm, editors, *Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC'19)*, pages 39–51, Paris, France, September 2019. ACM.

[19] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–636, September 2010.

[20] Jon Bentley. Programming Pearls: Little Languages. *Communications of the ACM*, 29(8):711–721, August 1986.

[21] Francesco Bertolotti, Walter Cazzola, and Luca Favalli. Features, Believe It or Not! A Design Pattern for First-Class Citizen Features on Stock JVM. In Jane Cleland-Huang and Wesley K. G. Assunção, editors, *Proceedings of the 26th International Software Product Line Conference (SPLC'22)*, pages 32–42, Graz, Austria, September 2022. ACM.

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

[23] Carla I. M. Bezerra, Rossana M. C. Andrade, and José Maria Monteiro. 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, pages 282–297, Miami, FL, USA, January 2015. Springer.

[24] Gilad Bracha and William Cook. Mixin-Based Inheritance. In Akinori Yonezawa, editor, *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP'90)*, pages 303–311, Ottawa, Canada, October 1990. ACM.

[25] Lionel Briand, Sandro Morasca, and Victor R. Basili. Defining and Validating High-Level Design Metrics. Technical Report CS-TR 3301, University of Maryland, Baltimore, MD, USA, 1994.

[26] Lionel C. Briand, John Daly, Victor Porter, and Jürgen Wüst. A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems. In *Proceedings of the 5th International Symposium on Software Metrics (METRICS'98)*, pages 246–257, Bethesda, MD, USA, March 1998. IEEE.

[27] Lionel C. Briand, John Daly, and Jürgen Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering*, 3(1):65–117, March 1998.

[28] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 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)*, pages 75–82, Madrid, Spain, February 2018. ACM.

[29] David N. Card, Victor E. Church, and William W. Agresti. An Empirical Study of Software Design Practices. *IEEE Transactions on Software Engineering*, 12(2):264–271, February 1986.

[30] David N. Card, Gerald T. Page, and Frank E. McGarry. Criteria for Software Modularization. In Meir M. Lehman, Horst Hünke, and Barry Boehm, editors, *Proceedings of the 8th International Conference on Software Engineering (ICSE'85)*, pages 372–377, London, United Kingdom, August 1985. IEEE.

[31] Walter Cazzola. Domain-Specific Languages in Few Steps: The Neverlang Approach. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Proceedings of the 11th International Conference on Software Composition (SC'12)*, Lecture Notes in Computer Science 7306, pages 162–177, Prague, Czech Republic, 31st of May-1st of June 2012. Springer.

[32] Walter Cazzola, Ruzanna Chitchyan, Awais Rashid, and Albert Shaqiri. μ-DSU: A Micro-Language Based Approach to Dynamic Software Updating. *Computer Languages, Systems & Structures*, 51:71–89, January 2018.

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

[34] Walter Cazzola and Diego Mathias Olivares. Gradually Learning Programming Supported by a Growable Programming Language. *IEEE Transactions on Emerging Topics in Computing*, 4(3):404–415, September 2016. Special Issue on Emerging Trends in Education.

[35] Walter Cazzola and Davide Poletti. DSL Evolution through Composition. In *Proceedings of the 7th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'10)*, Maribor, Slovenia, on 23rd of June 2010. ACM.

[36] Walter Cazzola and Albert Shaqiri. Open Programming Language Interpreters. *The Art, Science, and Engineering of Programming Journal*, 1(2):5–1–5–34, April 2017.

[37] Walter Cazzola and Ivan Speziale. Sectional Domain Specific Languages. In *Proceedings of the 4th Domain Specific Aspect-Oriented Languages (DSAL'09)*, pages 11–14, Charlottesville, Virginia, USA, on 3rd of March 2009. ACM.

[38] Walter Cazzola and Edoardo Vacchi. DEXTER and Neverlang: A Union Towards Dynamicity. In Eric Jul, Ian Rogers, and Olivier Zendra, editors, *Proceedings of the 7th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'12)*, Beijing, China, 11th of June 2012. ACM.

[39] Walter Cazzola and Edoardo Vacchi. Neverlang 2: Componentised Language Development for the JVM. In Walter Binder, Eric Bodden, and Welf Löwe, editors, *Proceedings of the 12th International Conference on Software Composition (SC'13)*, Lecture Notes in Computer Science 8088, pages 17–32, Budapest, Hungary, 19th of June 2013. Springer.

[40] Walter Cazzola and Edoardo Vacchi. On the Incremental Growth and Shrinkage of LR Goto-Graphs. *Acta Informatica*, 51(7):419–447, October 2014.

[41] Celia Chen, Reen Alfayez, Kamonphop Srisopha, Barry Boehm, and Lin Shi. 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)*, pages 377–378, Buenos Aires, Argentina, May 2017. IEEE.

[42] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A Survey of Compiler Testing. *ACM Computing Surveys*, 53(1), January 2021.

[43] Ting Chen, Simon Kornblith, and Mohammed Norouzi. A Simple Framework for Contrastive Learning of Visual Representations. In Emtiyaz Daumé III and Po-ling Loh, editors, *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*, pages 1597–1607, Vienna, Austria, July 2020. PMLR.

[44] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-Directed Differential Testing of JVM Implementations. In Emery D. Berger, editor, *Proceedings of the 37th International Conference on Programming Language Design adn Implementation (PLDI'16)*, pages 85–99, Santa Barbara, CA, USA, June 2016. ACM.

[45] Shyam R. Chidamber and Chris F. Kemerer. Towards a Metrics Suite for Object-Oriented Design. In Andreas Paepcke, editor, *Proceedings of the 6th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'91)*, pages 197–211, Phoenix, AZ, USA, October 1991. ACM.

[46] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[47] Sridhar Chimalakonda and Lee Dan Hyung. On the Evolution of Software and Systems Product Line Standards. *ACM SIGSOFT Software Engineering Notes*, 41(3):27–30, May 2016.

[48] Michele Chinosi and Alberto Trombetta. BPMN: An Introduction to the Standard. *Computer Standards and Interfaces*, 34(1):124–134, January 2012.

[49] Don Coleman. Assessing Maintenaibility. In *Proceedings of the HP Software Engineering Productivity Conference (SEPC'92)*, pages 525–532, Palo Alto, CA, USA, 1992. HP.

[50] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using Metrics to Evaluate Software System Maintainability. *IEEE Computer*, 27(8):44–49, August 1994.

[51] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: A Practical Mutation Testing Tool for Java. In Abhik Roychoudhury, editor, *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*, pages 449–452, Saarbrücken, Germany, July 2016. ACM.

[52] Adrian Colyer, Awais Rashid, and Gordon Blair. On the Separation of Concerns in Program Families. Technical Report 107, Lancaster University, Lancaster, United Kingdom, 2004.

[53] Benoît Combemale, Olivier Barais, and Andreas Wortmann. Language Engineering with the GEMOC Studio. In *Proceedings of the International Conference on Software Architecture Workshop (ICSAW'17)*, pages 189–191, Gothenburg, Sweden, April 2017. IEEE.

[54] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. ProVeLines: A Product Line of Verifiers for Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference (SPLC'13 Workshops)*, pages 141–146, Tokyo, Japan, August 2013. ACM.

[55] Michelle L. Crane and Juergen Dingel. UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal. In Lionel Briand and Clay Williams, editors, *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05)*, Lecture Notes in Computer Science 3713, pages 97–112, Montego Bay, Jamaica, 2005. Springer.

[56] Daniel Cruz, Eduardo Figueiredo, and Jabier Martinez. A Literature Review and Comparison of Three Feature Location Techniques Using ArgoUML-SPL. In Guilles Perrouin and Danny Weyns, editors, *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'19)*, pages 16:1–16:10, Leuven, Belgium, February 2019. ACM.

[57] Krysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In Sven Apel and Stefania Gnesi, editors, *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS'12)*, pages 173–182, Leipzig, Germany, January 2012. ACM.

[58] Ferruccio Damiani and Michael Lienhardt. Refactoring Delta-Oriented Product Lines to Achieve Monotonicity. In Julia Rubin and Thomas Thüm, editors, *Proceedings 7th International Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE'16)*, pages 2–16, Eindhoven, The Netherlands, April 2016.

[59] José Braga de Vasconcelos, Chris Kimble, Paulo Carreteiro, and Álvaro Rocha. The Application of Knowledge Management to Software Evolution. *International Journal of Information Management*, 37(1):1499–1506, February 2017.

[60] Thomas Degueule. Interoperability and Composition of DSLs with Melange. ACM Student Research Competition Grand Finals, June 2016.

[61] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: a Meta-Language for Modular and Reusable Development of DSLs. In Davide Di Ruscio and Markus Völter, editors, *Proceedings of the 8th International Conference on Software Language Engineering (SLE'15)*, pages 25–36, Pittsburgh, PA, USA, October 2015. ACM.

[62] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[63] Alex Denisov and Stanislav Pankevich. Mull It Over: Mutation Testing Based on LLVM. In Marinos Kintis, Nan Li, and José Miguel Rojas, editors, *Proceedings of the 13th International Workshop on Mutation Analysis (MUTATION'18)*, pages 25–31, Västerås, April 2018. IEEE.

[64] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[65] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denis Poshyvanyk. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software: Evolution and Process*, 25(1):53–95, January 2013.

[66] Juan José. Domínguez-Jiménez, Antonia Estero-Botaro, Antonio García-Domínguez, and Immaculada Medina-Bulo. Evolutionary Mutation Testing. *Information and Software Technology*, 53(10):1108–1123, October 2011.

[67] Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. Cyclomatic Complexity. *IEEE Software*, 33(6):27–29, November-December 2016.

[68] Torbjörn Ekman and Görel Hedin. The JastAdd System — Modular Extensible Compiler Construction. *Science of Computer Programming*, 69(1-3):14–26, December 2007.

[69] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. Metrics for Analyzing Variability and Its Implementation in Software Product Lines: A Systematic Literature Review. *Information and Software Technology*, 106:1–30, February 2019.

[70] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language Composition Untangled. In Anthony M. Sloane and Suzana Andova, editors, *Proceedings of the 12th Workshop on Language Description, Tools, and Applications (LDTA'12)*, Tallinn, Estonia, March 2012. ACM.

[71] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Alex Kelly, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. *Computer Languages, Systems and Structures*, 44:24–47, December 2015.

[72] Luca Favalli, Thomas Kühn, and Walter Cazzola. Neverlang and FeatureIDE Just Married: Integrated Language Product Line Development Environment. In Philippe Collet and Sarah Nadi, editors, *Proceedings of the 24th International Software Product Line Conference (SPLC'20)*, pages 285–295, Montréal, Canada, 19th-23rd of October 2020. ACM.

[73] Matthias Felleisen, Robert B. Findler, Matthew Flatt, Shiram Krishnamurthi, Eli Barzilay, and Sam McCarthy, Jay amd Tobin-Hochstadt. A Programmable Programming Language. *Communications of the ACM*, 61(3):62–71, March 2018.

[74] Norman E. Fenton. *Software Metrics: a Rigorous Approach*. London: Chapman & Hall, 1991.

[75] Fernández-Sáez, Ana M., Michel R. V. Chaudron, and Marcela Genero. An Industrial Case Study on the Use of UML in Software Maintenance and Its Perceived Benefits and Hurdles. *Empirical Software Engineering*, 23(6):3281–3345, March 2018.

[76] David Flanagan. *Java in a Nutshell*. O'Reilly Media, 2005.

[77] Matthew Flatt. Creating Languages in Racket. *ACM Queue*, 9(11):1–15, November 2011.

[78] Jerry A Fodor. *The language of thought*, volume 5. Harvard university press, 1975.

[79] Martin Fowler. Fluent Interface. Martin Fowler's Blog, May 2005.

[80] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? Martin Fowler's Blog, May 2005.

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

[82] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, Ma, USA, 1995.

[83] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman & Co., January 1979.

[84] Pierre-Loïc Garoche, Folk Howar, Temesghen Kahsai, and Xavier Thirioux. Testing-Based Compiler Validation for Synchronous Languages. In Julia M. Badger and Kristin Yvonne Rozier, editors, *Proceedings of the 6th International Symposium on NASA Formal Methods (NFM'14)*, Lectures Notes in Computer Science 8430), pages 246–251, Houston, TX, USA, April 2014. Springer.

[85] Robert Geist, A. Jefferson Offutt, and Frederick C. Harris, Jr. Estimation adn Enhancement of Real-Time Software Reliability through Mutation Analysis. *IEEE Transactions on Computers*, 41(5):550–558, May 1992.

[86] Debasish Ghosh. DSL for the Uninitiated. *ACM Queue Magazine*, 9(6):1–11, June 2011.

[87] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[88] Harald Görtz. A Concrete Grammar Specification Language for JastAdd. Master thesis, Department of Computer Science, Lund University, Lund, Sweden, June 2014.

[89] Alex Groce, Mahammad Amin Alipour, Chaqiang Zhang, Yang Chen, and John Regehr. Cause Reduction: Delta Debugging, Even without Bugs. *Journal of Software: Testing, Verification and Reliability*, 26(1):40–68, January 2016.

[90] Maurice H. Halstead. *Elements of Software Science*. Operating and Programming Systems. Elsevier, May 1977.

[91] Richard G. Hamlet. Testing Programs with the Aid of a Computer. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[92] Farah Hariri and August Shi. SRCIROR: A Toolset for Mutation Testing of C Source Code and LLVM Intermediate Representation. In Christian Kästner and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*, pages 860–863, Montpellier, France, September 2018. ACM.

[93] Farah Hariri, August Shi, Vimuth Fernando, Suleman Mahmood, and Darko Marinov. Comparing Mutation Testing at the Levels of Source Code and Compiler Intermediate Representation. In Myra Cohen and Atif Memon, editors, *Proceedings of the 12th Conference on Software Testing, Validation and Verification (ICST'19)*, pages 114–124, Xi'an, China, April 2019. IEEE.

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

[95] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Model-Based Language Engineering with EMFText. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'11)*, Lecture Notes in Computer Science 7680, pages 322–345, Braga, Portugal, July 2011. Springer.

[96] Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutation-Based Generation of Software Product Line Test Configurations. In Claire Le Goues and Shin Yoo, editors, *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE'14)*, Lecture Notes in Computer Science 8636, pages 92–106, Fortaleza, Brasil, August 2014. Springer.

[97] Pedro Rangel Henriques, Maria João Varanda Pereira, Marjan Mernik, Mitja Lenič, Jeff Gray, and Hui Wu. Automatic Generation of Language-Based Tools Using the LISA System. *IEE Proceedings — Software*, 152(2):54–69, April 2005.

[98] Sallie Henry and Calvin Selig. Predicting Source-Code Complexity at the Design Stage. *IEEE Software*, 7(2):36–44, March 1990.

[99] Jin Sun Her, Ji Hyeok Kim, Sung Yul Oh, Sang Hun amd Rhew, and Soo Dong Kim. A Framework for Evaluating Reausability of Core Asset in Product Line Engineering. *Information and Software Technology*, 49(7):740–760, July 2007.

[100] Martin Hitz and Behzad Montazeri. Measuring Coupling and Cohesion in Object-Oriented Systems. In *Proceedings of International Symposium on Applied Corporate Computing*, Monterrey, Mexico, October 1995.

[101] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with Code Fragments. In Tadayoshi Kohno, editor, *Proceedings of the 21st USENIX Security Symposium (USENIX'12)*, pages 445–458, Bellevue, WA, USAj, August 2012. USENIX Association.

[102] José Miguel Horcas Aguilera, Mónica Pinto, and Lidia Fuentes. Software Product Line Engineering: A Practical Experience. In Laurence Duchien and Thomas Thüm, editors, *Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC'19)*, pages 164–176, Paris, France, September 2019. ACM.

[103] William E. Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[104] ISO/IEC/IEEE International Standard. Systems and Software Engineering—Vocabulary. Standard 24765-2017, August 2017.

[105] Nishtha Jatana, Suri, Bharti, Prateek Kumar, and Bimlesh Wadhwa. Test Suite Reduction by Mutation Testing Mapped to Set Cover Problem. In *Proceedings of the 2nd International Conference on Information and Communication Technology for Competiive Strategies (ICTCS'16)*, pages 1–6, Udaipur, India, March 2016. ACM.

[106] Dennis Jeffrey and Neelam Gupta. Test Suite Reduction with Selective Redundancy. In Tibor Gyimothy and Vaclav Rajlich, editors, *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 549–558, Budapest, Hungary, September 2005. IEEE.

[107] Jean-Marc Jézéquel, Benoît Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. Mashup of Metalanguages and Its Implementation in the kermeta Language Workbench. *Software and Systems Modeling*, 14(2):905–920, May 2015.

[108] Jean-Marc Jézéquel, David Méndez-Acuña, Thomas Degueule, Benoît Combemale, and Olivier Barais. When Systems Engineering Meets Software Language Engineering. In Frédéric Boulanger, Daniel Krob, Gérard Morel, and Jean-Claude Rousse, editors, *Proceedings of the 5th International Conference on Complex Systems Design & Management (CSDM'14)* , pages 1–13, Paris, France, November 2014. Springer.

[109] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, September 2011.

[110] Gwendal Jouneaux, Olivier Barais, Benoît Combemale, and Gunter Mussbacher. Towards Self-Adaptable Languages. In Wolfgang De Meuter, editor, *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (ONWARD!'21)*, pages 97–113, Chicago, IL, USA, October 2021. ACM.

[111] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler. In Corina Pasareanu and John Hosking, editors, *Proceedings of the 26th International Conference on Automated Software Engineering (ASE'11)*, pages 612–615, Lawrence, KS, USA, November 2011. IEEE.

[112] Henry F. Kaiser. The Application of Electronic Computers to Factor Analysis. *Educational and Psychological Measurement*, 20(1):141–151, April 1960.

[113] Ted Kaminski and Eric Van Wyk. Modular Well-Definedness Analysis for Attribute Grammars. In Krzysztof Czarnecki and Görel Hedin, editors, *Proceedings of the 5th International Conference on Software Language Engineering (SLE'13)*, Lecture Notes in Computer Science 7745, pages 352–371, Dresden, Germany, September 2013. Springer.

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

[115] Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In Martin Rinard, Kevin J. Sullivan, and Daniel H. Steinberg, editors, *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*, pages 444–463, Reno, Nevada, USA, October 2010. ACM.

[116] Anthony Kenny. The homunculus fallacy. In *Investigating psychology*, pages 169–179. Routledge, 2016.

[117] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. Reducing Configurations to Monitor in a Software Product Line. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, and Insup Lee, editors, *Proceedings of the First International Conference on Runtime Verification (RV'10)*, Lecture Notes in Computer Science 6418, pages 285–299, St. Julians, Malta, November 2010. Springer.

[118] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. Detecting Trivial Mutant Equivalences via Compiler Optimisations. *IEEE Transactions on Software Engineering*, 44(4):308–333, April 2018.

[119] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Meta-Programming with Rascal. In João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering III (GTTSE'09)*, Lecture Notes in Computer Science 6491, pages 222–289, Braga, Portugal, July 2009. Springer.

[120] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In Andrew Walenstein and Sibylle Schupp, editors, *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*, pages 168–177, Edmonton, Canada, September 2009. IEEE.

[121] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[122] Ján Kollár and Michal Forgáč. Combined Approach to Program and Language Evolution. *Computing and Informatics*, 29(6):1103–1116, 2010.

[123] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain Specific Languages: A Systematic Mapping Study. *Information and Software Technology*, 71:77–91, March 2016.

[124] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Damiani Ferruccio. DeltaJ 1.5: Delta-Oriented Programming for Java 1.5. In Bruce Childers, editor, *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools (PPJ'14)*, pages 63–74, Cracow, Poland, September 2014. ACM.

[125] Alexander S. Kossatchev and Mikhail A. Posypkin. Survey of Compiler Testing Methods. *Programming and Computer Software*, 31(1):10–19, January 2005.

[126] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, September 2010.

[127] Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. Comparing Algorithms for Efficient Feature-Model Slicing. In Rick Rabiser and Bing Xie, editors, *Proceedings of the 20th International Systems and Software Product-Line Conference (SPLC'16)*, pages 60–64, Beijing, China, September 2016. ACM.

[128] Charles W. Krueger. Easing the Transition to Software Mass Customization. In F. van der Linden, editor, *Proceedings of the 1st International Workshop on Software*

Product-Family Engineering (PFE'01), Lecture Notes in Computer Science 2290, pages 282–293, Bilbao, Spain, October 2001. Springer.

[129] Thomas Kühn and Walter Cazzola. Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines. In Rick Rabiser and Bing Xie, editors, *Proceedings of the 20th International Software Product Line Conference (SPLC'16)*, pages 50–59, Beijing, China, 19th-23rd of September 2016. ACM.

[130] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. Choosy and Picky: Configuration of Language Product Lines. In Goetz Botterweck and Jules White, editors, *Proceedings of the 19th International Software Product Line Conference (SPLC'15)*, pages 71–80, Nashville, TN, USA, 20th-24th of July 2015. ACM.

[131] Thomas Kühn, Walter Cazzola, Nicola Pirritano Giampietro, and Massimiliano Poggi. Piggyback IDE Support for Language Product Lines. In Thomas Thüm and Laurence Duchien, editors, *Proceedings of the 23rd International Software Product Line Conference (SPLC'19)*, pages 131–142, Paris, France, 9th-13th of September 2019. ACM.

[132] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A Metamodel Family for Role-Based Modeling and Programming Languages. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jürgen Vinju, editors, *Proceedings of the 7th International Conference Software Language Engineering (SLE'14)*, Lecture Notes in Computer Science 8706, pages 141–160, Västerås, Sweden, September 2014. Springer.

[133] Hartmut Lackner and Martin Schmidt. Towards the Assessment of Software Product Line Tests: A Mutation System for Variable Systems. In Patrick Heymans and Julia Rubin, editors, *Proceedings of 18th International Software Product Line Conference (SPLC'14)*, pages 62–69, Florence, Italy, September 2014. ACM.

[134] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, April 2006.

[135] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Michael D. Smith, editor, *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–86, San José, CA, USA, March 2004.

[136] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

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

[138] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software. In Jeff Kramer, Judith Bishop, Prem Devanbu, and Sebastian Uchitel, editors, *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*, pages 105–114, Cape Town, South Africa, May 2010. IEEE.

[139] Jörg Liebig, Rolf Daniel, and Sven Apel. Feature-Oriented Language Families: A Case Study. In Philippe Collet and Klaus Schmid, editors, *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'13)*, Pisa, Italy, January 2013. ACM.

[140] Richard Lipton. Fault Diagnosis of Computer Programs. Student report, Carnegie Mellon University, 1971.

[141] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: An Automated Class Mutation System. *Software Testing, Verification & Reliability*, 15(2):97–133, June 2005.

[142] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: An Automated Class Mutation System. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 827–830, Shanghai, China, May 2006. ACM. Demo Paper.

[143] Laurens van der Maaten and Geoffrey Hinton. Visualizing Data Using t-SNE. *Journal of Machine Learning Research*, 9(86):2579–2605, November 2008.

[144] Allen Macro and John Buxton. *The Craft of Software Engineering*. Addison-Wesley, July 1987.

[145] Lech Madeyski, Yojciech Orzeszyna, Richard Torkar, and Mariusz Józala. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering*, 40(1):23–44, January 2014.

[146] Stefan Mann and Georg Rock. Control Variant-Rich Models by Variability Measures. In Krzysztof Czarnecki and Ulrich W. Eisenecker, editors, *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS'11)*, pages 29–38, Namur, Belgium, January 2011. ACM.

[147] Jabier Martinez, Nicolas Ordoñez, Xhevahire Tërnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. Feature Location Benchmark with ArgoUML SPL. In Thorsten Berger and Paulo Borba, editors, *Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC'18)*, pages 257–263, Gothenburg, Sweden, September 2018. ACM.

[148] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[149] Leland McInnes, John Healy, and James Melville. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *arXiv e-prints*, arXiv:1802.03426, February 2018.

[150] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017.

[151] Jens Meinicke, Thomas Thüm, Reimar Schröter, Sebastian Krieter, Fabian Benduhn, Gunter Saake, and Thomas Leich. FeatureIDE: Taming the Preprocessor Wilderness. In *Procedings of the 38th International Conference on Software Engineering Companion (ICSE'16-Companion)*, pages 629–632, Austin, TX, USA, May 2016. IEEE.

[152] David Méndez-Acuña, Jos´ A. Galindo, Benoît Combemale, Arnaud Blouin, and Benoît Baudry. Puzzle: A Tool for Analyzing and Extracting Specification Clones in DSLs. In Georgia M. Kapitsaki and Eduardo Santana de Almeida, editors, *Proceedings of the International Conference on Software Reuse (ICSR'16)*, Lecture Notes in Computer Science 9679, pages 393–396, Limassol, Cyprus, June 2016. Springer.

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

[154] Marjan Mernik. An Object-Oriented Approach to Language Compositions for Software Language Engineering. *Journal of Systems and Software*, 86(9):2451–2464, September 2013.

[155] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain Specific Languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[156] Marjan Mernik, Mitja Lenic, Enis Avdičauševic, and Viljem Zumer. Multiple attribute grammar inheritance. 1999.

[157] Marjan Mernik, Mitja Lenič, Enis Avdičauševič, and Viljem Žumer. LISA: An Interactive Environment for Programming Language Development. In Nigel R. Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*, Lecture Notes in Computer Science 2304, pages 1–4, Grenoble, France, April 2002. Springer.

[158] Mira Mezini and Klaus Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In Matthew B. Dwyer, editor, *Proceedings of the 12th international Symposium on Foundations of Software Engineering (FSE'04)*, pages 127–136, New Port Beach, CA, USA, October 2004. ACM.

[159] Alex Morgan. Representations gone mental. *Synthese*, 191(2):213–244, 2014.

[160] Peter D. Mosses. Software Meta-Language Engineering and CBS. *Journal of Computer Languages*, 50:39–48, February 2019.

[161] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions. *Information and Media Technologies*, 9(4):456–465, December 2014.

[162] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An Experimental Determination of Sufficient Mutant Operators. *Transaction on Software Engineering and Methodology*, 5(2):99–118, April 1996.

[163] A. Jefferson Offutt and Stephen D. Lee. How Strong Is Weak Mutation? In William E. Howden, editor, *Proceedings of the 4th Symposium on Testing, Analysis and Verification (TAV'91)*, pages 200–213, Victoria, BC, Canada, October 1991. ACM.

[164] A. Jefferson Offutt and Roland H. Untch. *Mutation 2000: Uniting the Orthogonal*, pages 34–44. Advances in Database Systems 24. Springer, May 2001.

[165] Aäron van den Oord, Yazhe Li, and Oriol Vinyals. Representation Learning with Contrastive Predictive Coding. *arXiv e-prints*, arXiv:1807.03748:1–13, July 2018.

[166] Sebastian Oster, Ivan Zorcic, Florian Markert, and Malte Lochau. MoSo-PoLiTe—Tool Support for Pairwise and Model-Based Software Product Line Testing. In Krzysztof Czarnecki and Ulrich W. Eisenecker, editors, *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS'11)*, pages 79–82, Namur, Belgium, January 2011. ACM.

[167] Stephen Palmer. Fundamental aspects of cognitive representation. 1978.

[168] Mike Papadakis, Thierry Titcheu Chekam, and Yves Le Traon. Mutant Quality Indicators. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW'18)*, pages 32–39, Västerås, Sweden, April 2018. IEEE.

[169] David L. Parnas. Information Distribution Aspects of Design Methodology. *Information Processing*, 71:339–344, 1971.

[170] David L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[171] Samuel Peacock, Lin Deng, Josh Dehlinger, and Suranjan Chakraborty. Automatic Equivalent Mutants Classification Using Abstract Syntax Tree Neural Networks. In *Proceedings of the IEEE 14th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'21)*, pages 13–18. IEEE, April 2021.

[172] Karl Pearson. Notes on Regression and Inheritance in the Case of Two Parents. *Proceedings of the Royal Society of London*, 58:240–242, June 1895.

[173] Juliana Alves Pereira, Sebastian Krieter, Jens Meinicke, Reimar Schröter, Gunter Saake, and Thomas Leich. FeatureIDE: Scalable Product Configuration of Variable Systems. In Georgia M. Kapitsaki and Eduardo Santana de Almeida, editors, *Proceedings of the 15th International Conference on Software Reuse (ICSR'16)*, Lecture Notes in Computer Science 9679, pages 397–401, Limassol, Cyprus, June 2016. Springer.

[174] Mikhail Perepletchikov, Caspar Ryan, Keith Frampton, and Zahir Tari. Coupling Metrics for Predicting Maintainability in Service-Oriented Designs. In *Proceedings of the 18th Australian Software Engineering Conference (ASWEC'07)*, pages 329–340, Melbourne, Australia, April 2007. IEEE.

[175] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journal*, 20(3):605–643, 2012.

[176] Jérôme Pfeiffer and Andreas Wortmann. Towards the Black-Box Aggregation of Language Components. In Federico Ciccozzi, Thomas Degueule, Romina Eramo, and Sébastien Gérard, editors, *Proceedings of the 3rd International Workshop on Modelling Language Engineering (MLE'21)*, pages 576–585, Fukuoka, Japan, October 2021. IEEE.

[177] Klaus Pohl, Klaus Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[178] Macario Polo, Mario Piattini, and Ignacio Garcia-Rodriguez. Decreasing the Cost of Mutation Testing with Second-Order Mutants. *Journal of Software: Testing, Verification and Reliability*, 19(2):111–131, June 209.

[179] Macario Polo Usaola and Pedro Reales Mateo. Mutation Testing Cost Reduction Techniques: A Survey. *IEEE Software*, 27(3):80–86, May-June 2010.

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

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

[182] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, sixth edition, 2005.

[183] Abdul Razzaq, Asanka Wasala, Chris Exton, and Jim Buckley. The State of Empirical Evaluation in Static Feature Location. *Transaction on Software Engineering and Methodology*, 28(1):1–58, January 2019.

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

[185] Julia Rubin and Marsha Chechik. A Survey of Feature Location Techniques. In Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and Jorn Bettin, editors, *Domain Engineering: Product Lines, Languages and Conceptual Models*, pages 29–58. Springer, 2013.

[186] Dan Saffer. *Designing for Interaction: Creating Innovative Applications and Devices*. New Riders, 2010.

[187] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-Oriented Programming of Software Product Lines. In Jan Bosch and Jaejoon Lee, editors, *Proceedings of the 14th International Software Product Line Conference (SPLC'10)*, Lecture Notes on Computer Science 6287, pages 77–91, Jeju Island, South Korea, September 2010. Springer.

[188] Klaus Schmid. A Comprehensive Product Line Scoping Approach and Its Validation. In Jeff Magee and Michal Young, editors, *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 593–603, Orlando FL, Florida, May 2002. ACM.

[189] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In David Forsyth, Ivan Laptev, Deva Ramanan, and Aude Oliva, editors, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'18)*, pages 815–823, Salt Lake City, UT, USA, June 2015. IEEE.

[190] David Schuler and Andreas Zeller. (Un-)Covering Equivalent Mutants. In Ana Rosa Cavalli and Sudipto Ghosh, editors, *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10)*, pages 45–54, Paris, France, April 2010. IEEE.

[191] Christoph Seidl, Sven Schuster, and Ina Schaefer. Generative Software Product Line Development Using Variability-Aware Design Patterns. *Computer Languages, Systems and Structures*, 48:89–111, June 2017.

[192] Maya Retno Ayu Setyautami and Reiner Hähnle. An Architectural Pattern to Realize Multi Software Product Lines in Java. In Paul Grünbacher, Christoph Seidl, Deepak Dhungana, and Helena Łovasz-Bukvova, editors, *Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS'21)*, pages 9:1–9:9, Krems, Austria, February 2021. ACM.

[193] Hazim Shatnawi and H. Conrad Cunningham. Encoding Feature Models Using Mainstream JSON Technologies. In Eric Gamess, editor, *Proceedings of the 2021 ACM Southeast Conferance (ACM-SE'21)*, pages 146–153, USA, April 2021. ACM.

[194] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. Variability Model of the Linux Kernel. In David Benavides, Don S. Batory, and Paul Grünbacher, editors, *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10)*, pages 45–51, Linz, Austria, January 2010. ACM.

[195] Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. A Theoretical and Empirical Study of Diversity-Aware Mutation Adequacy Criterion. *IEEE Transactions on Software Engineering*, 44(10):914–931, October 2018.

[196] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. Sufficient Mutation Operators for Measuring Test Effectiveness. In Matthew B. Dwyer and Volker Gruhn, editors, *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 351–360, Leipzig, Germany, May 2008. IEEE.

[197] Ben H. Smith and Laurie Williams. Should Software Testers Use Mutation Analysis to Augment a Test Set? *Journal of Systems and Software*, 82(11):1819–1832, November 2009.

[198] Marcelo Sousa and Alper Sen. Generation of TLM Testbenches Using Mutation Testing. In Naehyuck Chang and Franco Fummi, editors, *Proceedings of the eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'12)*, pages 323–332, Tampere, Finland, 2012. ACM.

[199] Dave Steinberg, Dave Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, December 2008.

[200] Chico Sundermann, Michael Nieke, Paul M. Bittner, Tobias Heß, and Thomas Thüm. Applications of #SAT Solvers on Feature Models. In Paul Grünbacher, Christoph Seidl, Deepak Dhungana, and Helena Łovasz-Bukvova, editors, *Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS'21)*, pages 1–10, Krems, Austria, February 2021. ACM.

[201] Richard E. Sweet. The Mesa Programming Environment. *ACM Sigplan Notice*, 20(7):216–229, July 1985.

[202] Éric Tanter. Reflection and Open Implementation. Technical Report TR-DCC-20091123-013, DCC, University of Chile, November 2009.

[203] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 79(1):70–85, January 2014.

[204] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In Ina Schaefer, Isabel John, and Klaus

Schmid, editors, *Proceedings of the 15th International Systems and Software Product-Line Conference (SPLC'11)*, pages 191–200, Münich, Germany, August 2011. ACM.

[205] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, and Sven Apel. Proof Composition for Deductive Verification of Software Product Lines. In *Proceedings of the International Conference on Software Testing Verification and Validation Workshop (ICSTW'11)*, pages 270–277, Berlin, Germany, March 2011. IEEE.

[206] Ilya Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Dosovitskiy. MLP-Mixer: An all-MLP Architecture for Vision. In Marc'Aurelio Ranzato, A. Beygelzimer, Y. Dauphin, P. S. Lian, and J. Wortman Vaughan, editors, *Processing of the 35th Conference on Neural Information Processing Systems (NeurIPS'21)*, volume 34, pages 24261–24272, Virtual, December 2021. Curran Associates, Inc.

[207] Laurence Tratt. Domain Specific Language Implementation Via Compile-Time Meta-Programming. *ACM Transaction on Programming Languages and Systems*, 30(6):31:1–31:40, October 2008.

[208] Douglas A. Troy and Stuart H. Zweben. Measuring the Quality of Structured Designs. *Journal of Systems and Software*, 2(2):113–120, June 1981.

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

[210] Edoardo Vacchi, Walter Cazzola, Benoît Combemale, and Mathieu Acher. Automating Variability Model Inference for Component-Based Language Implementations. In Patrick Heymans and Julia Rubin, editors, *Proceedings of the 18th International Software Product Line Conference (SPLC'14)*, pages 167–176, Florence, Italy, 15th-19th of September 2014. ACM.

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

[212] André van der Hock, Ebru Dincel, and Nenad Medvidović. 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)*, pages 298–308, Sidney, Australia, 2003. IEEE.

[213] Frank J Van der Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media, 2007.

[214] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*, pages 45–54, 2001.

[215] Lars van Hijfte and Ana Oprescu. MutantBench: an Equivalent Mutant Problem Comparison Framework. In *Proceedings of the IEEE 14th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'21)*, pages 7–12. IEEE, April 2021.

[216] Eelco Visser, Guido Wachsmuth, Andrew Tolmach, Pierre Neron, Vlad Vergu, Augusto Passalaqua, and Gabriël Konat. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!'14)*, pages 95–111, Portland, OR, USA, October 2014. ACM.

[217] Markus Völter and Vaclav Pech. Language Modularity with the MPS Language Workbench. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pages 1449–1450, Zürich, Switzerland, June 2012. IEEE.

[218] Markus Völter, Daniel Ratiu, Bernhard Schätz, and Bernd Kolb. 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)*, pages 121–140, Tucson, AZ, USA, October 2012. ACM.

[219] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards User-Friendly Projectional Editors. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Proceedings of the 7th International Conference on Software Language Engineering (SLE'14)*, Lecture Notes in Computer Science Volume 8706, pages 41–61, Västerås, Sweden, September 2014. Springer.

[220] Vladimir Vujović, Mirjana Maksimović, and Branko Perišić. Sirius: A Rapid Development of DSM Graphical Editor. In Tamás Haidegger and Levente Kovács, editors, *Proceedings of the IEEE 18th International Conference on Intelligent Engineering Systems (INES'14)*, Budapest, Hungary, July 2014. IEEE.

[221] Guido H. Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. Language Design with the Spoofax Language Workbench. *IEEE Software*, 31(5):35–43, September/October 2014.

[222] Philip Wadler. The Expression Problem. Java Genericity Mailing List, November 1998.

[223] Martin P. Ward. Language Oriented Programming. *Software—Concept and Tools*, 15(4):147–161, October 1994.

[224] Sik-Sang Yau, James S. Collofello, and T. MacGregor. Ripple Effect Analysis of Software Maintenance. In *Proceedings of the 2nd International Computer Software and Applications Conference (COMPSAC'78)*, pages 60–65, Chicago, IL, USA, November 1978. IEEE.

[225] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. Predictive Mutation Testing. *IEEE Transactions on Software Engineering*, 45(9):898–918, September 2019.

[226] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. Faster Mutation Testing Inspired by Test Prioritization and Reduction. In Mark Harman, editor, *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'13)*, pages 235–245, Lugano, Switzerland, July 2013. ACM.

[227] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. Regression Mutation Testing. In Zhendong Su, editor, *Proceedings of International Symposium on Software Testing and Analysis (ISSTA'12)*, pages 331–341, Minneapolis, MN, USA, July 2012. ACM.

[228] Tao Zhang, Lei Deng, Jian Wu, Qiaoming Zhou, and Chunyan Ma. Some Metrics for Accessing Quality of Product Line Architecture. In *Proceedings of the International Conference on Computer Science and Software Engineering (ICCSSE'08)*, pages 500–503, Wuhan, China, 2008.