The Journal of Systems & Software 195 (2023) 111533

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

The language mutation problem: Leveraging language product lines for mutation testing of interpreters[☆],[☆☆]

Walter Cazzola[*], Luca Favalli

Università degli Studi di Milano, Computer Science Department, Milan, Italy

ARTICLE INFO

Article history:
Received 18 February 2022
Received in revised form 26 August 2022
Accepted 30 September 2022
Available online 12 October 2022

MSC:
68N15
68N20
68M15

Keywords:
Language product lines
Mutation testing
Language testing

ABSTRACT

Compilers translate programs from a high level of abstraction into a low level representation that can be understood and executed by the computer; interpreters directly execute instructions from source code to convey their semantics. Undoubtedly, 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 work, we 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. We report results from a preliminary case study in which mutants of an ECMAScript interpreter are tested against the Sputnik conformance test suite for the ECMA-262 specification. The experimental data indicates that this approach can be used to create generally non-trivial mutants.

1. Introduction

Research context. 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 (Offutt and Untch, 2001).

Problem statement. 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 (Jia and Harman, 2011). All these problems still hold when the system under test (SUT) is the implementation of a programming language interpreter or compiler.[1] 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 (Kossatchev and Posypkin, 2005). While problem 3 draws the most attention in research (Peacock et al., 2021; van Hijfte and Oprescu, 2021; Kintis et al., 2018; Madeyski et al., 2014), in this work we want to focus on problem 2 to avoid the cost of 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 (Hariri et al., 2019). However, these approaches only partially solve problem 2 because, while they save

[☆] This work was partly supported by the MUR project “T-LADIES” (PRIN 2020TL3X8X).
[☆☆] Editor: Raffaela Mirandola.

[*] Corresponding author.
E-mail address: cazzola@di.unimi.it (W. Cazzola).

[1] Please note that this work focuses on interpreters rather than compilers.

https://doi.org/10.1016/j.jss.2022.111533
0164-1212/© 2022 Elsevier Inc. All rights reserved.

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.

Contribution. To cope with these issues we leverage language workbenches and properties that are specific to language implementations. Following the contribution from Leduc et al. (2020) on the *language extension problem*, we specify and tackle the *language mutation problem*: a language implementation together with its mutants can be seen as a language product line 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 language product line. According to this characterization, the mutation approach must respect the properties of *mutability in both dimensions*, *no modification or duplication*, *separate compilation* and *independent mutability*. In this approach, the base language is compiled once and then taken as a blackbox to which the mutation operators are applied at runtime. The result is a family of language mutants of the same base language. Therefore, the general contributions of this work are the characterization of the language mutation problem and the definition of a meta-model for its resolution.

Evaluation case study. We use the Neverlang (Vacchi and Cazzola, 2015) language workbench to define six mutation operators that can be applied at language feature level. We dub them *sourceless mutation operators* because they leverage previous work on Neverlang to adapt the parser (Cazzola and Vacchi, 2014) and the semantics (Cazzola et al., 2018) 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, given a base language, any sourceless mutation operator relies on the introspection and intercession capabilities provided by the Neverlang reflection API (Cazzola and Shaqiri, 2017b) to mutate either the language syntax, semantics or both. Each mutant is syntactically close to the base (correct) language implementation according to the *competent programmer hypothesis* (DeMillo et al., 1978). Then, we report the results of a preliminary evaluation case study 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. This evaluation has two non-general contributions: a concrete example of the applicability of the general resolution meta-model and the assessment of six language mutation operators for Neverlang.

Structure. The remainder of this paper is structured as follows. Section 2 presents the terminology. Section 3 provides an overview of our approach, whereas Section 4 shows how to apply this approach in practice. Section 5 showcases the evaluation case study. Finally in Sections 6 and 7 we give an overview of the related works and draw our conclusions.

2. Background

This section introduces the fundamentals and terminology required to understand the contribution that will be presented in Section 3. It first discusses the concepts of mutation testing and language product lines. Then it provides a brief overview of the *language extension problem* and of the *language feature* concept on which our solution is based.

2.1. Mutation testing

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 (1971) and other works from Hamlet (1977) and DeMillo et al. (1978) in the following years. The effectiveness of mutation testing depends on its capability of finding real faults (Geist et al., 1992). 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* (DeMillo et al., 1978) 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 (Offutt and Untch, 2001) 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.

2.2. Language product lines

Product lines are a staple in industrial production. Following the same ideas, *software product line* (SPL) engineering introduces the concepts of software variants and software families. A software family is a collection of related but different software variants that differ by the set of features they provide. SPLs are usually modeled in terms of their features following formalisms such as the *feature model* (FM), a concept firstly introduced as part of the FODA method (Kang et al., 1990). For this reason the development of SPLs is often referred to as feature-oriented programming. With the support of dedicated tools and environments such as FeatureIDE (Thüm et al., 2014; Meinicke et al., 2016, 2017), software engineers can cope with all the aspects of the development of a software family: domain analysis, domain implementation, requirement analysis, and product derivation. The idea of applying SPL concepts to the creation of families of language variants has gained popularity among researchers and practitioners (Ghosh, 2011; Kühn et al., 2014; Cazzola and Poletti, 2010), thus introducing *language product lines* (LPLs) (Vacchi et al., 2013; Kühn et al., 2015; Kühn and Cazzola, 2016).

2.3. Language workbenches

The LPL approach may prove useful to the creation of variants of a domain-specific language (Crane and Dingel, 2005; Tratt, 2008; Vacchi et al., 2014) and *dialects* of a general-purpose programming language (Cazzola and Olivares, 2016). LPL engineering 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* (Erdweg et al., 2015) embrace this philosophy to improve reusability and maintainability of language assets. The term *language workbench* was firstly introduced by Fowler (2005) to describe tools suited to the *language-oriented programming* paradigm (Ward, 1994), 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 (Fowler, 2005), 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 discussed in Section 2.2, 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 (Erdweg et al., 2012). The goal of a language workbench is to support all five forms of language composition through dedicated abstractions.

2.4. The language extension problem

To properly drive our research and to better express its constraints and challenges, we specified the problem introduced in Section 1 as an instance of the *language extension problem* (LEP). LEP was introduced by Leduc et al. (2020) as a paraphrase of the classic *expression problem* coined by Wadler (1998). 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 (Leduc et al., 2020). 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 the whole 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 (Leduc et al., 2020) to favor interesting design choices.

3. The language mutation problem

In this section, we introduce our approach towards the support of mutation testing for programming language implementations using LPLs. This section contains the following contributions:

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

The meta-model will later be applied in Sections 4 and 5 to prove its applicability using the Neverlang language workbench. The following subsections will outline our contributions, as well as the consequences, applicability and limitations of the resolution meta-model.

3.1. Problem overview

The problem of language mutation can be seen as an instance of the *language extension problem* (LEP), hereby dubbed 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, this contribution lifts the LEP to the context of mutation testing of language implementations to hopefully 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 Section 2.4, 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 unmutated 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 (Ma et al., 2006, 2005). The test suite for a language interpreter should be able to detect any error in the type system or if the semantics for any syntax are missing. Therefore, introducing errors in the type system when generating mutants may be beneficial when the goal is to assess the mutation adequacy of a test suite.

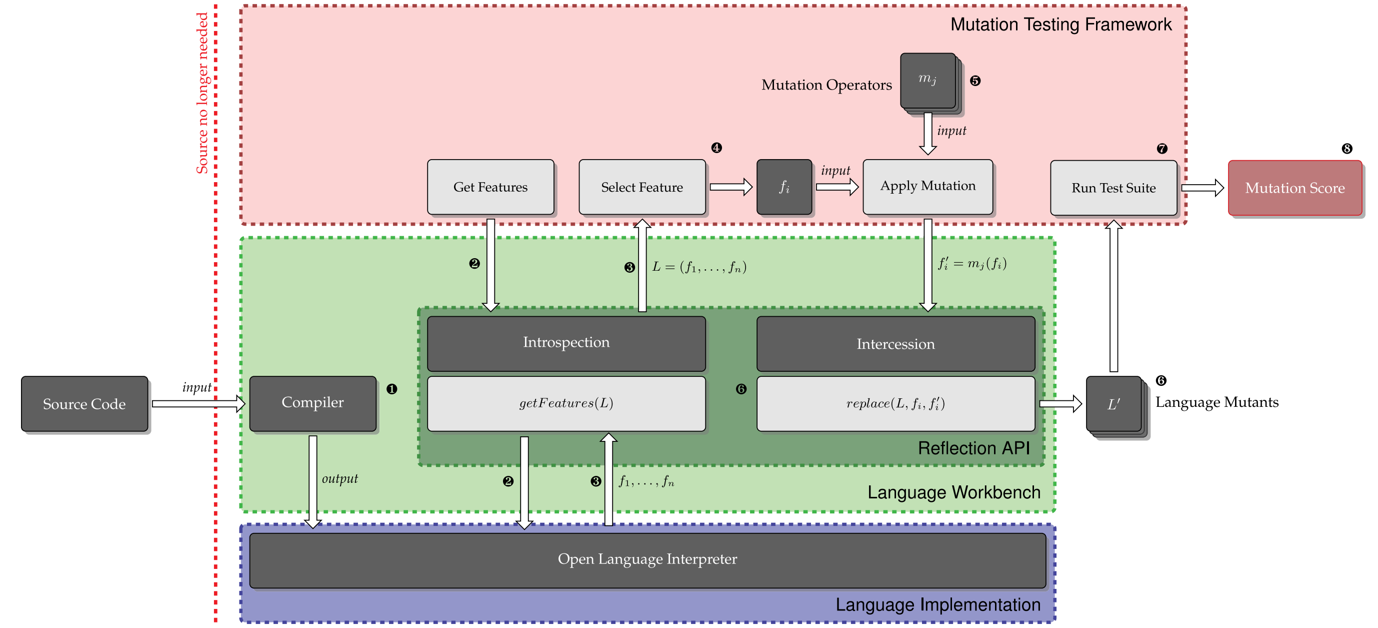


Fig. 1. Language mutation problem resolution meta-model, including the process, its actors and their interaction. (Figure colors are displayed only in the web version of this article.).

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*.

3.2. 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 Section 3.1. We discuss the actors of the software architecture, how these actors interact and the properties they should have 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 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

```
x = 5
y = 0
for x {
    x = y
    y = y + 1
}
```

Listing 1: Exemplary program written in language *L*.

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

and a restricted variant without loops

```
<statement>   ::=   <assignment>
```

Notice that Listing 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. 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. 1) because its modular structure drives the interaction among the three layers. Taking on the 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. 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, and addition between an integer and a float value promotes the integer to float. Finally, the evaluation phase runs each statement in the

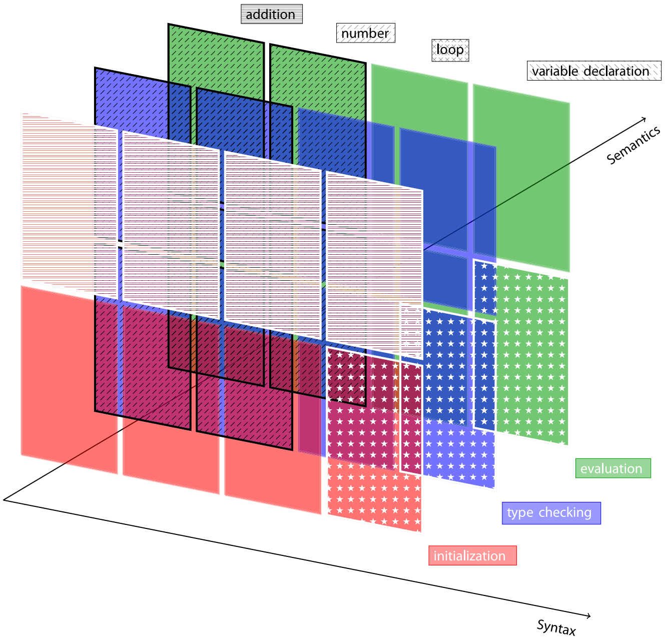


Fig. 2. Syntactic and semantic dimensions of a feature-oriented language implementation. Three different mutation operators are performed over the language implementation and highlighted with different patterns, depending on the dimensions they affect: over the semantic dimension case (white stars), over the syntactic dimension (white stripes) and over both (black stripes). (Figure colors are displayed only in the web version of this article.).

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. 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. 2 also shows exemplary mutation operators that can be performed over the language, as we will discuss later in this section. Finally, Fig. 1 shows that L is implemented as an *open language interpreter* (Cazzola and Shaqiri, 2017b) 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. 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. 1) and a *reflection API* (dark green box in Fig. 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 (Tanter, 2009).

Finally, the mutation testing framework (red in Fig. 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. 1.

First, the source code of the interpreter for L is given as an input to the language workbench compiler (Fig. 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. 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[2] 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. 1-❷). The request is forwarded to the language implementation which returns a collection of all the language features (Fig. 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. 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 $(m_1, \dots, m_n)$ and chooses one mutation operator m_j to apply over f_i to obtain a mutated feature f_i' (Fig. 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. 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 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 (Pfeiffer and Wortmann, 2021). Regardless of the chosen method, steps ❹ through ❻ are performed several times: once for each language mutant.

Finally, the mutation testing framework runs T against all the language mutants generated in the preceding steps (Fig. 1-❼) and outputs the mutation score of T (Fig. 1-❽).

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

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. 2 shows a modularization approach compliant to this constraint and its interaction with mutation operators. As discussed earlier, the language implementation shown in Fig. 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 operation 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. 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 Section 4, 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.

3.3. 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 according to the meta-model discussed in this section, a base language implementation is used to define a family of language mutants. In the context of language workbenches, a language mutant family can be modeled as a LPL which variability is expressed in terms of its features. Such a LPL contains two types of features:

- the base features of the SUT (such as $f_1, \ldots, f_n$ in Fig. 1);
- all the results of performing a mutation operation over a base feature (such as $fi' = m_j(f_i)$ in Fig. 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* (Leduc et al., 2020). Otherwise, the language family is *open* (Leduc et al., 2020). Given a 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[3] is closed, otherwise it is open. Families of high-order language mutants[4] are always open. In the following sections, we focus on closed first-order language mutant families.

Syntactic mutation operators. 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 the grammar itself may even become ambiguous. While this is generally against the goal we discussed in Section 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 and therefore the mutant is killed. Otherwise, no 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 Section 3.2: 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.

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

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

Table 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.

Approach	Type	Benefits	Limitations	Compatibility
LMP meta-model	Mutant execution	Avoids recompilation cost	Specific to language interpreters and requires paradigm shift	–
Greedy algorithms (Jeffrey and Gupta, 2005; Jatana et al., 2016)	Test suite reduction	Improves recurring test suite execution	All mutants must be generated, compiled and linked	✓
Test prioritization (Zhang et al., 2012, 2013)	Test suite reduction	Improves recurring test suite execution	All mutants must be generated, compiled and linked	✓
Compiler integration (Just et al., 2011)	Mutant execution	Nice trade-off between source code and bytecode	All mutants must be generated at the same time	✗
Sufficient operators (Offutt et al., 1996; Barbosa et al., 2001; Siami Namin et al., 2008)	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 (Zhang et al., 2019)	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 (Polo et al., 2009)	Mutant execution	Number of mutants is almost halved	All mutants must be generated at the same time	✓
Evolutionary algorithms (Domínguez-Jiménez et al., 2011)	Mutant execution	Generation of effective and hard to kill mutants	The entire test suite must be executed against all the mutants	✓
Weak mutation testing (Howden, 1982)	Mutant execution	Much less test suite execution is required	Less effective than strong mutation testing and not viable for critical applications	✓

The separate compilation constraint. Leduc et al. (2020) 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 Section 3.2, 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.

3.4. Comparison with the state-of-the-art

Mutation testing is intuitively expensive because it requires to run many tests against many mutants. Thus, we can distinguish between two types of approaches that try to reduce the mutation testing cost: approaches that save on the cost of running the test suite and the (more common) approaches that save on the cost of generating or executing mutants. Our meta-model 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 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. Minimizing a test suite has been shown to be NP-hard (Garey and Johnson, 1979), although there are approaches that try to save on test suite execution with greedy algorithms (Jeffrey and Gupta, 2005; Jatana et al., 2016) and test prioritization techniques (Zhang et al., 2012, 2013). 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 (Just et al., 2011) 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 (Offutt et al., 1996; Barbosa et al., 2001; Siami Namin et al., 2008) 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 (Basile et al., 2020, 2022). 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 (Polo Usaola and Reales Mateo, 2010). This can be improved upon using sufficient operators. However, the suitability of mutation operators might depend on the programming language (Polo Usaola and Reales Mateo, 2010) and therefore a new set of sufficient operators must be defined for each language.

Predictive mutation testing (Zhang et al., 2019) 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 features 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 (Polo et al., 2009), 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.

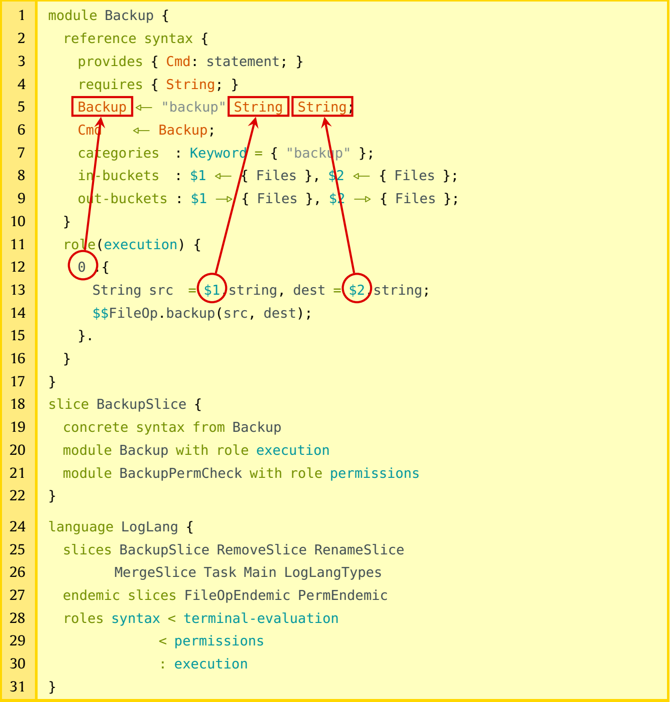
Evolutionary mutation testing (Domínguez-Jiménez et al., 2011) is a technique based on genetic algorithms that measures the usefulness of a mutant according to a fitness function (the execution matrix) to produce less but more effective mutants. The algorithm favors equivalent and difficult to kill mutants 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 (Howden, 1982) 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''), yet empirical evidence shows that it can be used as an effective alternative of non-critical applications (Offutt and Lee, 1991). 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 Polo et al. (2009), 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 Polo et al. (2009) 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 (Domínguez-Jiménez et al., 2011) also uses a fitness function based on a matrix that is very similar to the one we used in this work (Section 5). Finally, weak mutation testing can be combined with the LMP resolution meta-model thanks to the language workbenches capabilities of automatically generating a debugger for any mutant (Kühn et al., 2019). 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 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.

4. Language mutation in practice

In this section, we show the application of the conceptual meta-model proposed in Section 3.2 to a concrete use case. This section is not meant to restrict the applicability of the meta-model to a specific technological framework and it should instead be used as a complement to our contribution. This section also includes an introduction to Neverlang and its language workbench capabilities: this background information can be used as a reference for the evaluation case study presented in Section 5. There exist several language workbenches with a different approach to modularization. Instantiating the meta-model to each existing language workbench is beyond the scope of this work, however the end of this section provides some hints on how to generalize the application of the meta-model to other language workbenches.



```
 1  module Backup {
 2    reference syntax {
 3      provides { Cmd: statement; }
 4      requires { String; }
 5      Backup ← "backup" String String;
 6      Cmd    ← Backup;
 7      categories  : Keyword = { "backup" };
 8      in-buckets  : $1 ← { Files }, $2 ← { Files };
 9      out-buckets : $1 → { Files }, $2 → { Files };
10    }
11    role(execution) {
12      0.{
13        String src  = $1.string, dest = $2.string;
14        $$FileOp.backup(src, dest);
15      }.
16    }
17  }
18  slice BackupSlice {
19    concrete syntax from Backup
20    module Backup with role execution
21    module BackupPermCheck with role permissions
22  }
24  language LogLang {
25    slices BackupSlice RemoveSlice RenameSlice
26          MergeSlice Task Main LogLangTypes
27    endemic slices FileOpEndemic PermEndemic
28    roles syntax < terminal-evaluation
29                 < permissions
30                 : execution
31  }
```

Listing 2: Syntax and semantics for the backup task.

4.1. The Neverlang language workbench

In this section, we introduce the basic concepts of the Neverlang language workbench and its syntax.

Overview. Neverlang (Cazzola, 2012; Cazzola and Vacchi, 2013; Cazzola and Shaqiri, 2017a) is a language workbench for the development of programming languages compilers, interpreters and their ecosystem in a modular way. Neverlang embraces the feature-oriented programming paradigm and LPL engineering since the entire development cycle is based on the *language feature* concept (Cazzola and Favalli, 2022). Language features are implemented in compilation units called *slices*. Slices implement language features by performing composition between several units called *modules*. The composition mechanism is *syntax-driven*: the language grammar is used for selecting insertion points where semantic actions are plugged in. The semantics are implemented through inherited and synthesized attributes using the *syntax directed translation* technique (Aho et al., 2006).

Neverlang syntax. Listing 2 showcases the basics of Neverlang's syntax through the implementation of the Backup language feature of the LogLang LPL. LogLang (Cazzola and Poletti, 2010) is a simple DSL that describes tasks for a log rotating tool similar to the Unix logrotate utility with a modular Neverlang implementation. The Backup module declares a reference syntax for the backup task (lines 2–10). The reference syntax of a module also piggybacks (Kühn et al., 2019) meta-data for the deployment of basic IDE services, such as syntax highlighting (the categories on line 7) and code-completion (the input and output buckets on lines 8–9). Semantic actions (lines 12–15) are attached to nonterminals of the productions using the syntax-driven mechanism. Nonterminals are numbered according to their position in the reference syntax: numbering starts with 0 and grows from the top left to the bottom right.[5] 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. Then, the head of the second production (Cmd) will be $3 and the righ-hand nonterminal (Backup) will be $4. Grammar attributes are accessed by name using a dot notation (as done in line 13) over the nonterminals which are indexed as discussed above. The **slice** compilation unit embodies the language feature concept and drives the composition between syntactic and semantic assets while adapting possibly incompatible assets. In this example, 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 24–31) indicates which slices should be composed to generate the language interpreter (lines 25–26). Therefore, composition in Neverlang is twofold: 1. between modules, which yields slices, and 2. between slices, which yields a language implementation. Composition is also supported through bundles (not shown in Listing 2) that behave just as languages but they can be embedded in other languages. As a result of the composition mechanism, the grammar fragments from each concrete syntax of every slice 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; in this case the goal is to match a nonterminal provided by another production in the grammar. Semantic actions are executed with regards to the program's abstract syntax tree and roles are executed according to the **roles** clause (line 28) of the **language** descriptor. In the example, permission is executed after parsing and terminal-evaluation. The **language** clause can also declare **endemic slices**—*i.e.*, semantic assets with no syntax that are shared across multiple compilation phases (line 27). Please see Vacchi and Cazzola (2015) for a complete overview of Neverlang.

4.2. Neverlang workbench capabilities

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.

Overview. The output of the mutation testing process presented in Section 3.2 is a LPL of language mutants. Neverlang supports LPL engineering thanks to AiDE (Kühn et al., 2015; Cazzola and Favalli, 2022). 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) to determine the language features and their dependencies and synthesizes the corresponding FM for a given language family (Vacchi and Cazzola, 2015) in a bottom-up fashion using the algorithm introduced in Vacchi et al. (2013) and refined in Favalli et al. (2020). 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 (Cazzola and Favalli, 2022) throughout the configuration mechanism. AiDE is currently integrated with FeatureIDE[6] and the Gradle build tool[7] to ease the generation of Java artifacts and the deployment of a language variant implementation.

While the LPL capabilities are useful to support the generation and deployment of language families, they do not directly

[5] 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.

[6] https://featureide.github.io/.

[7] https://gradle.org/.

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 (Vacchi and Cazzola, 2015) 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 shown on line 21 of Listing 2. 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 discussed in Section 4.1 adapts incompatible grammar fragments by renaming nonterminals in their productions. These composition mechanisms were 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 (Vacchi and Cazzola, 2015) 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, lines 24–31, 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 Section 5 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.

4.3. Mutation operators

In this section, we instantiate each of the three mutation operator categories discussed in Section 2.3. 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.

Table 2
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.

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

Overview. As discussed in Section 3.2, the mutation operators are functions over language features. In Neverlang the focus will be 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 Section 4.2.

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. The same operators are summarized in Table 2.

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 Section 3.3. For instance, in a standard *term-factor* grammar (Aho et al., 2006) 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 ith 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 (Aho et al., 2006), 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 ith 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 ith 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.

Table 3
Possible targets of a mutation operator class in several language workbenches.

Language workbench	Syntax	Semantics	Syntax + Semantics
Spoofax (Wachsmuth et al., 2014)	SDF3 grammar specification	Rules and strategies	Strategies pattern matching
MPS (Völter and Pech, 2012)	Editor	Behavior	Concept extension + overriding
Melange (Degueule et al., 2015)	Ecore metamodel	Kermeta aspects	Renaming + aspect extension
MontiCore (Krahn et al., 2010)	Syntax tree node extension	Attribute injection	Associations
Rascal (Klint et al., 2009)	Abstract data type adapters	Function wrappers	Pattern-based dispatch mechanism

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.

```
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 Section 3.2 and to satisfy the four constraints of the LMP presented in Section 3.1. This can be achieved by defining different mutation operators that target syntactic artifacts, semantic artifacts or both. Table 3 hints at some well-known language workbenches and the artifacts that could be targetted by the mutation operators. This list is not meant to be exhaustive and other language workbenches could provide a different solution to the LMP.

5. Evaluation case study

Overview. In this section, we assess the Neverlang implementation of the meta-model outlined in Section 3.2 and detailed in Section 4. The SUT will be a family of mutants of a Neverlang implementation of the ECMAScript interpreter obtained by applying the mutation operators introduced in Section 4.3 on the language features of the base language. First, we frame the scope of this evaluation by introducing the research questions that we are trying to answer, then we review how the experiment was setup and report the results. Finally, we answer the research questions, discuss any threats to the validity of this evaluation and overview the lessons we learned in this work. This section contains the following contributions:

1. it shows the applicability of the approach in a concrete scenario and
2. it assesses 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.

Research questions. This evaluation is validated by answering the following research questions:

RQ$_1$. Which Neverlang sourceless mutation operators produce variants of the language mutant family that are reasonably hard to discover and kill?

RQ$_2$. Are mutation operators producing different language mutant variants? Can we obtain similar results by reducing the number of classes?

To answer **RQ$_1$** 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 **RQ$_2$** 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* (Shin et al., 2018) based on the idea that mutants can be distinguished from each other by the set of tests that kills them.

5.1. Setup

Hardware setup. All experiments were run on a 64 bits Arch Linux machine with an Intel Core i7-1065G7 3.9 GHz 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 s on average on this machine. Thus, avoiding to re-compile all the mutants by applying the mutation operators at runtime saves about 7 s on each run, for a total of about 2 h 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 (Cazzola and Olivares, 2016)—which we did not modify in any way. The data for the evaluation were obtained from the Sputnik ECMA-262 specification conformance test suite[8] 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 library function, we filtered out all tests that were not compliant with the base language, for a total of 2137 remaining tests.

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

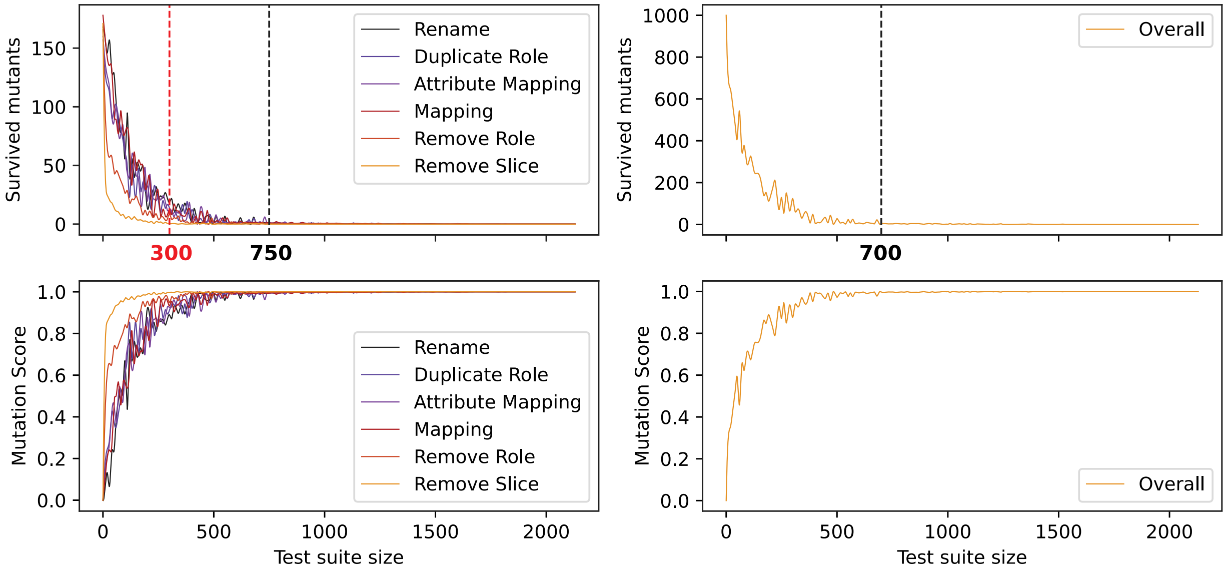


Fig. 3. Number of survived mutants and mutation score for each of the considered mutation operators.

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.[9] Finally, we loaded the results into a 2137 × 1000 matrix, in which position (i, j) was set to 1 if the ith test killed the jth mutant and 0 otherwise. All the results reported in Section 5.2 are obtained by analyzing this matrix.

5.2. 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_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 Section 1 with the mutant equivalent problem. Testing the mutation operators introduced in Section 4.3 against the Sputnik conformance test suite, we expect the test suite to be able to kill all generated mutants.

Fig. 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 (Smith and Williams, 2009). 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 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. 3 also shows the effect that reducing test suite size[10] 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. 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. 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.

Figs. 4 and 5 focus on different aspects of this evaluation. Fig. 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. 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» (DeMillo et al., 1978). Fig. 4 visually confirms the results from Fig. 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, each box does not only represent the median

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

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

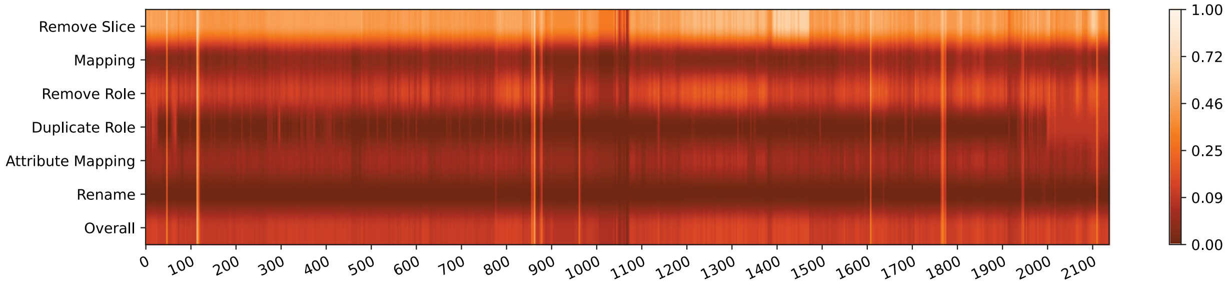


Fig. 4. Probability with which each selected test kills a mutant of each class. (Figure colors are displayed only in the web version of this article.).

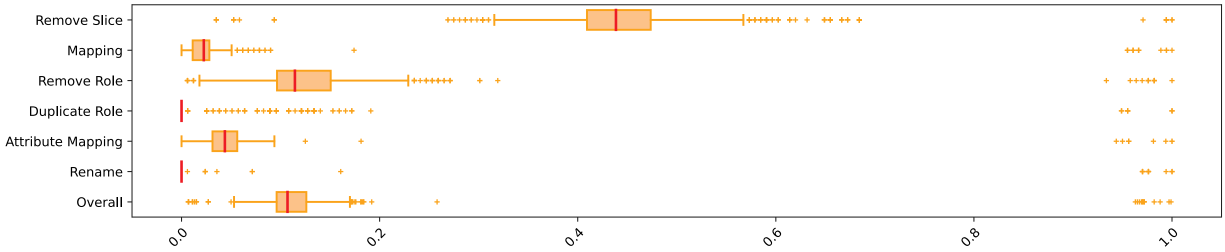


Fig. 5. Box plot representing the locality, spread and skewness of the results for each of the six Neverlang sourceless language mutation operators. (Figure colors are displayed only in the web version of this article.).

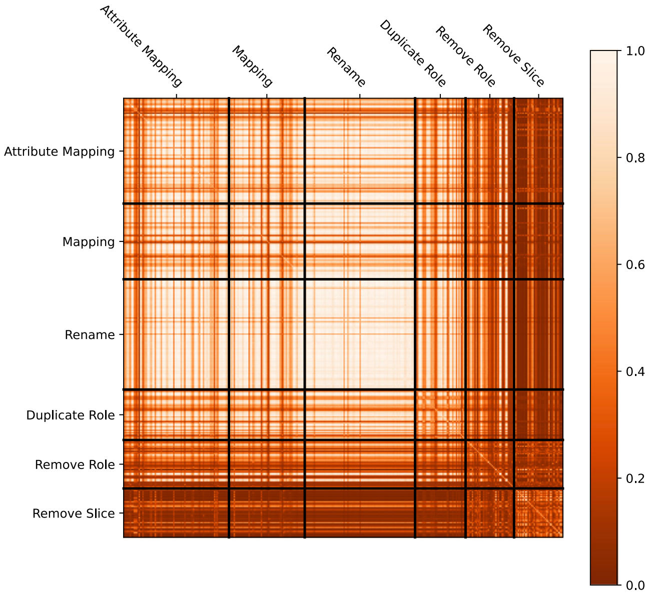


Fig. 6. Average Jaccard similarity coefficient for all mutation operator classes. (Figure colors are displayed only in the web version of this article.).

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 probability from Fig. 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 and add very little to this evaluation. Please refer to the companion dataset[9] for the results of all tests.

RQ$_2$. Recall that each column of the 2137 × 1000 matrix we introduced in Section 5.1 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:

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

We cannot tabulate the results due to space constraints so please refer to the companion dataset[9] to replicate the results. Instead, Fig. 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 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. 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. 6 shows that the similarity between mutants of different classes is usually low despite the presence of some clusters. Instead, Fig. 7 isolates each of the six classes by highlighting in large detail the sub-matrices along the diagonal of Fig. 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.

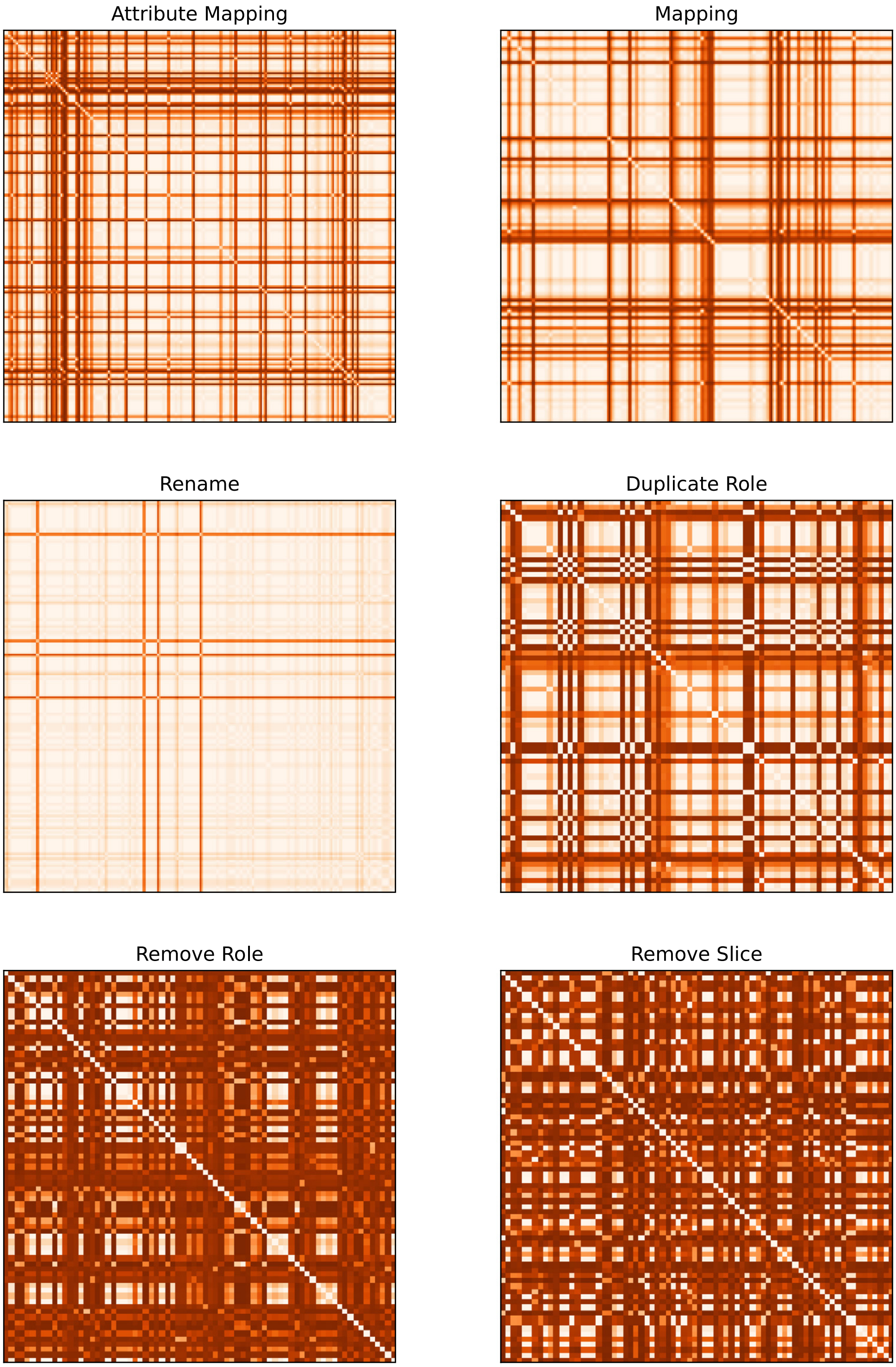


Fig. 7. Average Jaccard similarity coefficient for each mutation operator class.

5.3. Discussion

RQ$_1$. 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 improve the quality of the test suite and artificially skews the mutation score towards 1. Fig. 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. 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$_1$.

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$_2$. 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. 7. Therefore two different mutation operator instances of the same class are very likely to be redundant and hard to distinguish (Shin et al., 2018). 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$_2$.

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. 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. 7.

5.4. Threats to validity

In this section, we discuss any internal and external threats that could affect the validity of this evaluation. Following Wohlin et al. (2003), the internal validity is «the degree to which conclusions can be drawn about the causal effect of the treatments on the outcomes», whereas the external validity is «the degree to which the results of the research can be generalized to the population under study and other research setting».

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. 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 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. In this work, 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 and 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 (Section 3.2) 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 Section 4 and the evaluation of Section 5 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 be replicated for the evaluation of different mutation operators in other language workbenches. Finally, we addressed the general applicability in Table 3, 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 a 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. (2020) 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 hinted at alternative solutions, such as generating the entire family of mutants in advance, a solution that is applicable—although not advised—to closed mutant families.

5.5. Learned lessons

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.

6. Related work

Overview. This work deals with several different domains, including adaptive languages, compiler testing and mutation testing. In this section, we briefly discuss these topics. We do not discuss the topic of language workbenches because—to the best of our knowledge—there are no other contributions that leverage language workbenches to perform the mutation testing of language implementations.

Adaptable languages. Cazzola et al. (2018) 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 this work. Kollár and Forgáč (2010) presented an adaptive approach to both program and language modification to support dynamic evolution. More recently, Jouneaux et al. (2021) 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. There are several works that try to reduce the mutation testing cost, some of which we detailed in Section 3.4. Literature also 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 (Lattner and Adve, 2004) framework is usually the target of the mutation. Some examples are SRCIROR (Hariri and Shi, 2018), Mull (Denisov and Pankevich, 2018) and the contributions from Sousa and Sen (2012) and from Papadakis et al. (2018). Similarly, JAVALANCHE (Schuler and Zeller, 2010) and PIT (Coles et al., 2016) manipulate Java bytecode directly to avoid the cost of recompilation. Mutation testing in the context of SPLs often relies on model-based mutation operators (Lackner and Schmidt, 2014; Henard et al., 2014) 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. However, all the techniques reviewed in this paragraph are orthogonal to ours and could be used in conjunction to achieve the best of both worlds. Such an approach using both sourceless mutation operators and traditional mutation operators will be part of a future work.

Language testing. Chen et al. (2021) recently 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. (2014), 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. (2016) used Markov Chain Monte Carlo sampling to select mutations with higher chance of triggering compiler bugs. Holler et al. (2012) replace random nonterminals in test cases with expansions of the same nonterminals according to the language grammar. Garoche et al. (2014) take a complete test suite as input and mutate it to produce more failure-inducing programs. Groce et al. (2016) 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, our approach modifies the language implementation directly and is therefore more suited to test suite quality assessment. Moreover, none of these contributions involves language product lines 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.

7. Conclusions

Testing the implementation of a programming language interpreter is of critical importance because its quality affects the quality of all software developed my means of that language. A popular technique used to evaluate the quality of test suites in research is mutation testing. However, 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 work we specified this problem as a derivation of the *Language Extension Problem* (Leduc et al., 2020), 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 evaluation 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. This aspect could be improved upon by extending Neverlang with more fine-grained composition mechanisms in a future work. This should reduce the redundancy and the triviality of some of the mutation operators. Future works will also include an hybrid (non-general) approach in which some knowledge over the base language is known in advance to target more sensible parts of the grammar or changing its semantics. Moreover, we will investigate whether the same approach translates well to the mutation testing of a software developed by means of the programming language, rather than the language implementation itself. Finally, we will extend the AiDE (Cazzola and Favalli, 2022) LPL development environment with an algorithm to produce the feature model of the mutant family and use its properties to tackle the mutant equivalent problem.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

There is a link to a Zenodo record in the paper.

References

Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D., 2006. Compilers: Principles, Techniques, and Tools, second ed. Addison-Wesley, Boston, MA, USA.

Barbosa, E.F., Maldonado, J.C., Rizzo Vincenzi, A.M., 2001. Toward the determination of sufficient mutant operators for c. J. Softw.: Test. Verif. Reliab. 11 (2), 113–136.

Basile, D., ter Beek, M.H., Lazreg, S., Cordy, M., Legay, A., 2022. Static detection of equivalent mutants in real-time model-based mutation testing. Empir. Softw. Eng. 27.

Basile, D., ter Beek, M.H., Cordy, M., Legay, A., 2020. Tackling the equivalent mutant problem in real-time systems: The 12 commandments of model-based mutation testing. In: Collet, P., Nadi, S. (Eds.), Proceedings of the 24th International Software Product Line Conference (SPLC'20). ACM, Montréal, Canada, pp. 252–262.

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

Cazzola, W., Chitchyan, R., Rashid, A., Shaqiri, A., 2018. μ-DSU: A micro-language based approach to dynamic software updating. Comput. Lang. Syst. Struct. 51, 71–89. http://dx.doi.org/10.1016/j.cl.2017.07.003.

Cazzola, W., Favalli, L., 2022. Towards a recipe for language decomposition: Quality assessment of language product lines. Empir. Softw. Eng. 27 (4), http://dx.doi.org/10.1145/3514232.

Cazzola, W., Olivares, D.M., 2016. Gradually learning programming supported by a growable programming language. IEEE Trans. Emerg. Top. Comput. 4 (3), 404–415. http://dx.doi.org/10.1109/TETC.2015.2446192, special Issue on Emerging Trends in Education.

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

Cazzola, W., Shaqiri, A., 2017a. Context-aware software variability through adaptable interpreters. IEEE Softw. 34 (6), 83–88. http://dx.doi.org/10.1109/MS.2017.4121222, special Issue on Context Variability Modeling.

Cazzola, W., Shaqiri, A., 2017b. Open programming language interpreters. Art Sci. Eng. Program. J. 1 (2), 5–1–5–34. http://dx.doi.org/10.22152/programming-journal.org/2017/1/5.

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

Cazzola, W., Vacchi, E., 2014. On the incremental growth and shrinkage of LR goto-graphs. Acta Inform. 51 (7), 419–447. http://dx.doi.org/10.1007/s00236-014-0201-2.

Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., Zhang, L., 2021. A survey of compiler testing. ACM Comput. Surv. 53 (1).

Chen, Y., Su, T., Sun, C., Su, Z., Zhao, J., 2016. Coverage-directed differential testing of JVM implementations. In: Berger, E.D. (Ed.), Proceedings of the 37th International Conference on Programming Language Design Adn Implementation (PLDI'16). ACM, Santa Barbara, CA, USA, pp. 85–99.

Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A., 2016. PIT: A practical mutation testing tool for java. In: Roychoudhury, A. (Ed.), Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16). ACM, Saarbrücken, Germany, pp. 449–452.

Crane, M.L., Dingel, J., 2005. UML vs. Classical vs. Rhapsody statecharts: Not all models are created equal. In: L. Briand, C. Williams (Ed.), Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05). In: Lecture Notes in Computer Science, vol. 3713, Springer, Montego Bay, Jamaica, pp. 97–112.

Degueule, T., Combemale, B., Blouin, A., Barais, O., Jézéquel, J.-M., 2015. Melange: a meta-language for modular and reusable development of DSLs. In: Ruscio, D. Di, Völter, M. (Eds.), Proceedings of the 8th International Conference on Software Language Engineering (SLE'15). ACM, Pittsburgh, PA, USA, pp. 25–36.

DeMillo, R.A., Lipton, R.J., Sayward, F.G., 1978. Hints on test data selection: Help for the practicing programmer. IEEE Comput. 11 (4), 34–41.

Denisov, A., Pankevich, S., 2018. Mull it over: Mutation testing based on LLVM. In: Kintis, M., Li, N., Rojas, J.M. (Eds.), Proceedings of the 13th International Workshop on Mutation Analysis (MUTATION'18). IEEE, Västerås, pp. 25–31.

Domínguez-Jiménez, J.J., Estero-Botaro, A., García-Domínguez, A., Medina-Bulo, I., 2011. Evolutionary mutation testing. Inf. Softw. Technol. 53 (10), 1108–1123.

Erdweg, S., Giarrusso, P.G., Rendel, T., 2012. Language composition untangled. In: Sloane, A.M., Andova, S. (Eds.), Proceedings of the 12th Workshop on Language Description, Tools, and Applications (LDTA'12). ACM, Tallinn, Estonia.

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

Favalli, L., Kühn, T., Cazzola, W., 2020. Neverlang and FeatureIDE just married: Integrated language product line development environment. In: Collet, P., Nadi, S. (Eds.), Proceedings of the 24th International Software Product Line Conference (SPLC'20). ACM, Montréal, Canada, pp. 285–295.

Fowler, M., 2005. Language workbenches: The killer-app for domain specific languages? Martin Fowler's Blog URL http://www.martinfowler.com/articles/languageWorkbench.html.

Garey, M.R., Johnson, D.S., 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman & Co..

Garoche, P.-L., Howar, F., Kahsai, T., Thirioux, X., 2014. Testing-based compiler validation for synchronous languages. In: Badger, J.M., Rozier, K.Y. (Eds.), Proceedings of the 6th International Symposium on NASA Formal Methods (NFM'14). In: Lectures Notes in Computer Science, vol. 8430, Springer, Houston, TX, USA, pp. 246–251.

Geist, R., Offutt, A.J., Harris, Jr., F.C., 1992. Estimation adn enhancement of real-time software reliability through mutation analysis. IEEE Trans. Comput. 41 (5), 550–558.

Ghosh, D., 2011. DSL for the uninitiated. ACM Queue Mag. 9 (6), 1–11.

Groce, A., Alipour, M.A., Zhang, C., Chen, Y., Regehr, J., 2016. Cause reduction: Delta debugging, even without bugs. J. Softw.: Test. Verif. Reliab. 26 (1), 40–68.

Hamlet, R.G., 1977. Testing programs with the aid of a computer. IEEE Trans. Softw. Eng. 3 (4), 279–290.

Hariri, F., Shi, A., 2018. SRCIROR: A toolset for mutation testing of c source code and LLVM intermediate representation. In: Kästner, C., Fraser, G. (Eds.), Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18). ACM, Montpellier, France, pp. 860–863.

Hariri, F., Shi, A., Fernando, V., Mahmood, S., Marinov, D., 2019. Comparing mutation testing at the levels of source code and compiler intermediate representation. In: Cohen, M., Memon, A. (Eds.), Proceedings of the 12th Conference on Software Testing, Validation and Verification (ICST'19). IEEE, Xi'an, China, pp. 114–124.

Henard, C., Papadakis, M., Le Traon, Y., 2014. Mutation-based generation of software product line test configurations. In: Goues, C. Le, Yoo, S. (Eds.), Proceedings of the International Symposium on Search Based Software Engineering (SSBSE'14). In: Lecture Notes in Computer Science, vol. 8636, Springer, Fortaleza, Brasil, pp. 92–106.

Holler, C., Herzig, K., Zeller, A., 2012. Fuzzing with code fragments. In: Kohno, T. (Ed.), Proceedings of the 21st USENIX Security Symposium (USENIX'12). USENIX Association, Bellevue, WA, USAj, pp. 445–458.

Howden, W.E., 1982. Weak mutation testing and completeness of test sets. IEEE Trans. Softw. Eng. 8 (4), 371–379.

Jatana, N., Suri, Bharti, Kumar, P., Wadhwa, B., 2016. 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). ACM, Udaipur, India, pp. 1–6.

Jeffrey, D., Gupta, N., 2005. Test suite reduction with selective redundancy. In: Gyimothy, T., Rajlich, V. (Eds.), Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05). IEEE, Budapest, Hungary, pp. 549–558.

Jia, Y., Harman, M., 2011. An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 37 (5), 649–678.

Jouneaux, G., Barais, O., Combemale, B., Mussbacher, G., 2021. Towards self-adaptable languages. In: Meuter, W. De (Ed.), Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (ONWARD!'21). ACM, Chicago, IL, USA, pp. 97–113.

Just, R., Schweiggert, F., Kapfhammer, G.M., 2011. MAJOR: An efficient and extensible tool for mutation analysis in a java compiler. In: Pasareanu, C., Hosking, J. (Eds.), Proceedings of the 26th International Conference on Automated Software Engineering (ASE'11). IEEE, Lawrence, KS, USA, pp. 612–615.

Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

Kintis, M., Papadakis, M., Jia, Y., Malevris, N., Le Traon, Y., Harman, M., 2018. Detecting trivial mutant equivalences via compiler optimisations. IEEE Trans. Softw. Eng. 44 (4), 308–333.

Klint, P., van der Storm, T., Vinju, J., 2009. RASCAL: A domain specific language for source code analysis and manipulation. In: Walenstein, A., Schupp, S. (Eds.), Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM'09). IEEE, Edmonton, Canada, pp. 168–177.

Kollár, J., Forgáč, M., 2010. Combined approach to program and language evolution. Comput. Inform. 29 (6), 1103–1116.

Kossatchev, A.S., Posypkin, M.A., 2005. Survey of compiler testing methods. Program. Comput. Softw. 31 (1), 10–19.

Krahn, H., Rumpe, B., Völkel, S., 2010. Monticore: A framework for compositional development of domain specific languages. Int. J. Softw. Tools Technol. Transfer 12 (5), 353–372.

Kühn, T., Cazzola, W., 2016. Apples and oranges: Comparing top-down and bottom-up language product lines. In: Rabiser, R., Xie, B. (Eds.), Proceedings of the 20th International Software Product Line Conference (SPLC'16). ACM, Beijing, China, pp. 50–59.

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

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

Kühn, T., Leuthäuser, M., Götz, S., Seidl, C., Aßmann, U., 2014. A metamodel family for role-based modeling and programming languages. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J. (Eds.), Proceedings of the 7th International Conference Software Language Engineering (SLE'14). In: Lecture Notes in Computer Science, vol. 8706, Springer, Västerås, Sweden, pp. 141–160.

Lackner, H., Schmidt, M., 2014. Towards the assessment of software product line tests: A mutation system for variable systems. In: Heymans, P., Rubin, J. (Eds.), Proceedings of 18th International Software Product Line Conference (SPLC'14). ACM, Florence, Italy, pp. 62–69.

Lattner, C., Adve, V., 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In: Smith, M.D. (Ed.), Proceedings of the International Symposium on Code Generation and Optimization (CGO'04). San José, CA, USA, pp. 75–86.

Leduc, M., Degueule, T., Van Wyk, E., Combemale, B., 2020. The software language extension problem. Softw. Syst. Model. 19 (2), 263–267.

Lipton, R., 1971. Fault Diagnosis of Computer Programs. Student Report, Carnegie Mellon University.

Ma, Y.-S., Offutt, J., Kwon, Y.R., 2005. Mujava: An automated class mutation system. Softw. Test. Verif. Reliab. 15 (2), 97–133.

Ma, Y.-S., Offutt, J., Kwon, Y.R., 2006. Mujava: An automated class mutation system. In: Proceedings of the 28th International Conference on Software Engineering (ICSE'06). ACM, Shanghai, China, pp. 827–830, demo Paper.

Madeyski, L., Orzeszyna, Y., Torkar, R., Józala, M., 2014. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. IEEE Trans. Softw. Eng. 40 (1), 23–44.

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

Meinicke, J., Thüm, T., Schröter, R., Krieter, S., Benduhn, F., Saake, G., Leich, T., 2016. FeatureIDE: Taming the preprocessor wilderness. In: Procedings of the 38th International Conference on Software Engineering Companion (ICSE'16-Companion). IEEE, Austin, TX, USA, pp. 629–632.

Nagai, E., Hashimoto, A., Ishiura, N., 2014. Reinforcing random testing of arithmetic optimization of c compilers by scaling up size and number of expressions. Inf. Media Technol. 9 (4), 456–465.

Offutt, A.J., Lee, S.D., 1991. How strong is weak mutation? In: Howden, W.E. (Ed.), Proceedings of the 4th Symposium on Testing, Analysis and Verification (TAV'91). ACM, Victoria, BC, Canada, pp. 200–213.

Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C., 1996. An experimental determination of sufficient mutant operators. Trans. Softw. Eng. Methodol. 5 (2), 99–118.

Offutt, A.J., Untch, R.H., 2001. Mutation 2000: Uniting the orthogonal. In: Advances in Database Systems, Vol. 24. Springer, pp. 34–44.

Papadakis, M., Chekam, T.T., Le Traon, Y., 2018. Mutant quality indicators. In: Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW'18). IEEE, Västerås, Sweden, pp. 32–39.

Peacock, S., Deng, L., Dehlinger, J., Chakraborty, S., 2021. 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). IEEE, pp. 13–18.

Pfeiffer, J., Wortmann, A., 2021. Towards the black-box aggregation of language components. In: Ciccozzi, F., Degueule, T., Eramo, R., Gérard, S. (Eds.), Proceedings of the 3rd International Workshop on Modelling Language Engineering (MLE'21). IEEE, Fukuoka, Japan, pp. 576–585.

Polo, M., Piattini, M., Garcia-Rodriguez, I., 2009. Decreasing the cost of mutation testing with second-order mutants. J. Softw.: Test. Verif. Reliab. 19 (2), 111–131.

Polo Usaola, M., Reales Mateo, P., 2010. Mutation testing cost reduction techniques: A survey. IEEE Softw. 27 (3), 80–86.

Schuler, D., Zeller, A., 2010. (Un-)covering equivalent mutants. In: Cavalli, A.R., Ghosh, S. (Eds.), Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10). IEEE, Paris, France, pp. 45–54.

Shin, D., Yoo, S., Bae, D.-H., 2018. A theoretical and empirical study of diversity-aware mutation adequacy criterion. IEEE Trans. Softw. Eng. 44 (10), 914–931.

Siami Namin, A., Andrews, J.H., Murdoch, D.J., 2008. Sufficient mutation operators for measuring test effectiveness. In: Dwyer, M.B., Gruhn, V. (Eds.), Proceedings of the 30th International Conference on Software Engineering (ICSE'08). IEEE, Leipzig, Germany, pp. 351–360.

Smith, B.H., Williams, L., 2009. Should software testers use mutation analysis to augment a test set? J. Syst. Softw. 82 (11), 1819–1832.

Sousa, M., Sen, A., 2012. Generation of TLM testbenches using mutation testing. In: Chang, N., Fummi, F. (Eds.), Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'12). ACM, Tampere, Finland, pp. 323–332.

Tanter, E., 2009. Reflection and Open Implementation. Tech. Rep. TR-DCC-20091123-013, DCC, University of Chile.

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T., 2014. FeatureIDE: An extensible framework for feature-oriented software development. Sci. Comput. Program. 79 (1), 70–85.

Tratt, L., 2008. Domain specific language implementation via compile-time meta-programming. ACM Trans. Program. Lang. Syst. 30 (6), 31:1–31:40.

Vacchi, E., Cazzola, W., 2015. Neverlang: A framework for feature-oriented language development. Comput. Lang. Syst. Struct. 43 (3), 1–40. http://dx.doi.org/10.1016/j.cl.2015.02.001.

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

Vacchi, E., Cazzola, W., Pillay, S., Combemale, B., 2013. Variability support in domain-specific language development. In: Erwig, M., Paige, R.F., Wyk, E. Van (Eds.), Proceedings of 6th International Conference on Software Language Engineering (SLE'13). In: Lecture Notes on Computer Science, vol. 8225, Springer, Indianapolis, USA, pp. 76–95.

van Hijfte, L., Oprescu, A., 2021. Mutantbench: an equivalent mutant problem comparison framework. In: Proceedings of the IEEE 14th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'21). IEEE, pp. 7–12.

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

Wachsmuth, G.H., Konat, G.D.P., Visser, E., 2014. Language design with the spoofax language workbench. IEEE Softw. 31 (5), 35–43.

Wadler, P., 1998. The expression problem, java genericity mailing list.

Ward, M.P., 1994. Language oriented programming. Softw.—Concept Tools 15 (4), 147–161.

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

Zhang, L., Marinov, D., Khurshid, S., 2013. Faster mutation testing inspired by test prioritization and reduction. In: Harman, M. (Ed.), Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'13). ACM, Lugano, Switzerland, pp. 235–245.

Zhang, L., Marinov, D., Zhang, L., Khurshid, S., 2012. Regression mutation testing. In: Su, Z. (Ed.), Proceedings of International Symposium on Software Testing and Analysis (ISSTA'12). ACM, Minneapolis, MN, USA, pp. 331–341.

Zhang, J., Zhang, L., Harman, M., Hao, D., Jia, Y., Zhang, L., 2019. Predictive mutation testing. IEEE Trans. Softw. Eng. 45 (9), 898–918.

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

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