The Journal of Systems & Software 202 (2023) 111704

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

On the granularity of linguistic reuse☆,☆☆

Francesco Bertolotti, Walter Cazzola *, Luca Favalli

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

ARTICLE INFO

Article history:
Received 6 October 2022
Received in revised form 19 February 2023
Accepted 9 April 2023
Available online 25 April 2023

Keywords:
Reuse and evolution
Language evolution
Domain specific languages
Feature modularity
Language product lines
Language composition

ABSTRACT

Programming languages are complex software systems integrated across an ecosystem of different applications such as language compilers or interpreters but also an integrated development environment comprehensive of syntax highlighting, code completion, error recovery, and a debugger. The complexity of language ecosystems can be faced using language workbenches—*i.e.*, tools that tackle the development of programming languages, domain specific languages and their ecosystems in a modular way.

As with any other software system, one of the priorities that developers struggle to achieve when developing programming languages is reusability. After all, the capacity to easily reuse and adapt existing components to new scenarios can dramatically improve development times. Therefore, as programming languages offer features to reuse existing code, language workbenches should offer tools to reuse existing language assets. However, reusability can be achieved in many different ways.

In this work, we identify six forms of linguistic reusability, ordered by level of granularity: (i) *sub-languages composition*, (ii) *language features composition*, (iii) *syntax and semantics assets composition*, (iv) *semantic assets composition*, (v) *actions composition*, and. (vi) *action extension*. We use these mechanisms to extend the taxonomy of language composition proposed by Erdweg et al. To show a concrete application of this taxonomy, we evaluate the capabilities provided by the Neverlang language workbench with regards to our taxonomy and extend it by adding explicit support for any granularity level that was originally not supported. This is done by instantiating two levels of reusability as actual operators—desugaring, and delegation. We evaluate these operators against the clone-and-own approach, which was the only form of reuse at that level of granularity prior to the introduction of explicit operators. We show that with the clone-and-own approach the design quality of the source code is negatively affected. We conclude that language workbenches can benefit from the introduction of mechanisms to explicitly support reuse at all granularity levels.

1. Introduction

As any other software product, domain-specific languages (DSLs) (Kosar et al., 2016; Mernik et al., 2005) and general purpose languages (GPLs) are products in constant evolution (Karaila, 2009; Chowdhary, 2013). New syntactic constructs and their semantics are continuously integrated. This constant evolution can easily become unmanageable (Mens et al., 2005; Serebrenik and Mens, 2015). Moreover, it can affect a wide range of applications that orbit around the language. As a consequence, each application needs to be updated accordingly and consistently. This process is extremely mechanical, can slow the language development and can cause the introduction of bugs.

While there is a wide literature to support the evolution of common software products (Chapin et al., 2001; Lehman et al., 2002; Hinterreiter et al., 2018), the evolution of programming language ecosystems is often overlooked (Thanhofer-Pilisch et al., 2017). Despite being software products, GPLs and DSLs can benefit from additional care—*e.g.*, automatic integrated development environment (IDE) generation (Kühn et al., 2019; Henriques et al., 2005). Moreover, these language products can deeply impact the creation of new source code. Given the critical role of these products, adding an explicit support to reuse of linguistic assets for evolution purposes can positively impact the development of a language.

A common approach to the creation of language ecosystems is to leverage *language workbenches* as a modeling framework. Following *software product line* (SPL) engineering, language workbenches can describe languages in terms of their features. A language feature encapsulates a language construct or a language concept (Kühn et al., 2015) that can be separately compiled and distributed. Language features consist of a syntactic part and a

☆ This work was partly supported by the Italian Ministry of University and Research (MUR), Italy on the funded project "T-LADIES" (PRIN 2020TL3X8X).
☆☆ Editor: Lingxiao Jiang.
* Corresponding author.
E-mail address: cazzola@di.unimi.it (W. Cazzola).

https://doi.org/10.1016/j.jss.2023.111704
0164-1212/

Table 1
Changes required by the introduction of language extensions according to Google's open source JavaScript enginer (V8[3]) project history.

Feature	Hash	Files	Insertions	Deletions
NullishOperator (??)	3ec1036526227af480f8516c99758ba5b4ee8749	14	227	15
AndAssignment (&&=), OrAssignment (||=) and NullishAssignement(??=)	b151d8db22be308738192497a68c2c7c0d8d4070	8	39	16
NumericSeparator (100_000_000 or 0.000_001)	517df5248847fa773159b465ebe713dc05c03fa5	8	226	52
ArrowFunction (=>)	7367720daa7d57ed3b9d92944bdddec61e5ab88a	9	678	174
OptionalCatchBinding (try{..} catch {..})	d0651bd108e0ee70ae822eda9bad7049cb2f3df4	8	78	52
Const	3b1aabc960ea7a1107c8b6ebb8f2b4ce48e9b610	3	24	16

semantic part. With this in mind, reuse[1] among language features can be at several levels of granularity, both within language features and between language features through semantics and syntax reuse (Combemale et al., 2018). Moreover, reuse can be performed through extension mechanisms that modify existing assets into new ones, or through composition mechanisms that combine different assets into new ones (ISO/IEC/IEEE International Standard, 2017). In both cases, old assets are reused to obtain newer assets.

Hinterreiter et al. (2018) already put reuse of SPLs among the main challenges of software evolution. Following the same sentiment, we believe reuse among linguistic assets should be encouraged through specific constructs and leveraged to support the evolution of programming languages.

In this work, we introduce six different levels of granularity in linguistic reuse. These levels of granularity complement a previous work from Erdweg et al. on language composition, that classifies the composition of complete languages with other languages or with language extensions. This work builds on this classification by addressing the composition of linguistic components at fine levels of granularity—*i.e.* those in which the result of the composition is smaller than a language extension. Then, we exemplify, compare and evaluate two explicit mechanisms for opportunistic reuse[2] that can be used to support the maintenance and evolution of language semantics by measuring how they perform against the clone-and-own (Dubinsky et al., 2017; Rubin et al., 2013; Antkiewicz et al., 2014) solution. For each approach, we highlight strengths and weakness and evaluate the quality of the resulting code by measuring several metrics. We show that, when linguistic reuse is not explicitly supported at all levels of granularity, the code suffers from increased duplication and coupling. The evaluation is based on the extension of a Neverlang implementation of ECMAScript 3 with some language features introduced in newer versions of the ECMAScript standard.

The contribution of this work is twofold and includes (i) an extension to the taxonomy on language composition by Erdweg et al. (2012) and (ii) a framework for the evaluation of composition mechanisms to support reuse of linguistic components in language workbenches. We express and validate these contributions by answering the following research questions.

RQ_1. What are the levels of granularity needed to support reuse of linguistic assets in the implementation of programming languages?

RQ_2. How can a composition operator that performs linguistic reuse in Neverlang be evaluated with respect to its reuse capabilities?

The rest of the paper is organized as follows. Section 2 discusses a running example highlighting the problem we are going to tackle. Here, we also provide an overview of the current state-of-the-art with regards to the considered problem. Section 3 introduces the needed terminology and sets a common background the paper is built on. Sections 4 and 5 present the proposed approaches and the evaluation framework respectively. Finally, Sections 6 and 7 present some related work and our conclusions, respectively.

2. Problem statement

Overview. In Section 2.1, we will discuss the main motivations behind this work. We will then compare them with the current state-of-the-art in terms of language workbenches in Section 3.2.

2.1. Motivation

Programming languages evolve overtime. For example, Table 1 shows several extensions that were introduced in the V8 JavaScript engine over the years. The introduction of such extensions may be costly in terms of line of code insertions and deletions to existing implementations. One example of such an implementation is shown in Fig. 1 (central panel): the `||=` operator that was added to ECMAScript in 2021—here reported as it would be implemented in the Neverlang language workbench. Notice that in this case the extension was implemented by applying clone-and-own reuse, since Neverlang does not provide mechanisms to perform reuse for linguistic components at this level of granularity. There is significant overlap between the implementation of the `OrAssignment` extension and the existing `JSOrExpression` and `JSAssignExpression`—all the lines of code from both `JSOrExpression` (left panel) and `JSAssignExpression` are present in the new `OrAssignment` (right panel). The main problem with the clone-and-own approach is that it creates several variants of the same linguistic component that can quickly become expensive to maintain due to code duplication and change propagation for updates (Rubin et al., 2013).

To summarize, in our view a language workbench should solve all of the above problems by striving towards three different goals: (i) enabling reuse of linguistic assets without changing them, (ii) favoring opportunistic reuse of existing assets through explicit constructs and (iii) preventing the need for code duplication. To achieve these goals, a language workbench must provide abstractions enabling reuse at every level of *granularity*—*i.e.*, the amount of computation performed by reusable components. Large grained components yield larger win in productivity (Tracz, 1990; Hemer and Lindsay, 2001; Long, 2001) but may implement a functionality too specific to be reused. Conversely, fine grained components are more likely to be reusable but provide smaller benefits (Xu et al., 2011; Maga et al., 2011).

3. Background

Overview. In this section, we will review some fundamental concepts useful for future sections. We will start by discussing the

[1] Reuse is the degree to which an asset can be used to build other systems and/or assets (ISO/IEC/IEEE International Standard, 2017).

[2] Opportunistic reuse is defined as the activity of reusing and combining software assets that were not originally designed to be used together (Sen, 1997; Mäkitalo et al., 2020).

[3] https://github.com/v8/v8

```
module JSOrExpression {
  reference syntax {
    OrExpr ← OrExpr "||" AndExpr;
  } role(evaluation) { 0 .{
    eval $1;
    JSType lval = $1.value;
    JSType result;

    if (lval.booleanValue()) result = lval;
    else {
      eval $2;
      result = $2.value;

    }
    $0.value = result;
  }. }
}
```

```
module OrAssignment {
  reference syntax {
    AssignExpr ← LeftExpr "||=" AssignExpr;
  } role(evaluation) { 0 .{
    eval $1;
    JSType lval = $1.value;
    JSType result;
    if (!(lval instanceof JSReference))
      throw new SyntaxError(
        lval + " is not a reference.");
    if (lval.booleanValue()) result = lval;
    else {
      eval $2;
      result = $2.value;
      JSReference lref = (JSReference) lval;
      lref.setValue(result);
    }
    $0.value = result;
  }. }
}
```

```
module JSAssignExpression {
  reference syntax {
    AssignExpr ← LeftExpr "=" AssignExpr;
  } role (evaluation) { 0 .{
    eval $1;
    JSType lval = $1.value;
    JSType result;
    if (!(lval instanceof JSReference))
      throw new SyntaxError(
        lval + " is not a reference.");

    else {
      eval $2;
      result = $2.value;
      JSReference lref = (JSReference) lval;
      lref.setValue(result);
    }
    $0.value = result;
  }. }
}
```

Fig. 1. The central panel shows the Neverlang implementation of the JavaScript ||= language feature (named `OrAssignment`). The left panel shows the Neverlang implementation of the || operator (named `JSOrExpression`). The right panel shows the Neverlang implementation of the assignment = (named `JSAssignExpression`). On top, highlighted in red, the respective syntaxes. Several code overlaps are shown. Yellow stresses the overlappings between || implementation and ||= implementation. Green stresses the overlappings between || implementation and = implementation. Finally, blue stresses the overlappings common to all (||, ||= and =) operators.

Erdweg taxonomy on language composition. Next, we will review few language workbenches. We will discuss language product lines, and the Neverlang ecosystem. Finally, we will define some of the terminology used later on.

3.1. Erdweg language composition taxonomy

According to Erdweg et al. (2012) the mechanisms for language composition can be divided into four different classes:

- *Language extension*: a base language is composed with a language fragment to add new capabilities. By itself the language fragment cannot be used meaningfully as it depends on the base language. For example, the statement ||= represents a language extension of ECMAScript 2020.
- *Language unification*: two existing languages are composed together. For example JavaScript and HTML are often used in combination to achieve more dynamic behaviors.
- *Self-extension*: a base language provides a program that encapsulates a different language. For example, the Java language is self-extended with the regex DSL using embedded strings and API calls.
- *Extension composition*: a base language uses two language extensions jointly.

3.2. State-of-the-art

Melange. Melange (Degueule et al., 2015; Degueule, 2016; Combemale, 2015) is a language workbench that integrates various tools from the Eclipse Modeling Framework (EMF) ecosystem (Steinberg et al., 2008). The abstract syntaxes are defined in Xtext and the corresponding semantics are specified with Xtend-based Kermeta 3 aspects. To build a language with Melange, the developer must first design an Ecore model containing a declarative specification of a set of classes for the abstract syntax, and then Kermeta semantic aspects. The latter are linked to Ecore-generated classes by specific annotations. According to the taxonomy by Erdweg et al. (2012), Melange supports language extension, language unification, self-extension and extension composition (Méndez-Acuña et al., 2016). Melange offers complex composition mechanisms such as the extension of an existing language or merging of multiple languages (Degueule et al., 2015). Melange ties the semantics to specific abstract data type (ADT) which may limit opportunistic reuse opportunities, although this issue could be tampered thanks to the support of renaming of Ecore elements (packages, classes, etc.).

MontiCore. MontiCore (Grönninger et al., 2008; Krahn et al., 2010; Rumpe et al., 2021) is a language workbench that adopts a unique DSL for the definition of both abstract and concrete syntax. Class models are automatically generated and their semantics are implemented in Java through abstract syntax tree (AST) visitors. Each visitor can access inherited and synthesized grammar attributes through a getter/setter API which is injected into the target ADTs. According to the taxonomy by Erdweg et al. (2012), MontiCore supports language extension, language unification, self-extension and extension composition (Méndez-Acuña et al., 2016). ADTs associated to AST nodes can be extended to reuse existing semantics and multiple grammar inheritance is supported, but it is not possible to add new grammar attributes to existing symbols. For similar reasons, separate compilation of artifacts is not supported which may limit the extensibility when the sources for the original base language are no longer available.

Meta programming system (MPS). MPS (Völter and Pech, 2012; Pech et al., 2013) is a development environment for non-textual DSLs. MPS avoids the need for any kind of parser by supporting the *projectional* (Völter et al., 2014) approach: the abstract syntax is defined in meta-models and stored in XML files which are not meant to be human-editable. Instead, programs are modeled by composing basic building blocks called *concepts*, each defining a type of AST node. Various kinds of components (mainly *behaviors* and *editors*) can be attached to each concept to implement a view for the AST and its semantics through a subset of Java called BaseLanguage. According to the taxonomy by Erdweg et al. (2012), MPS supports language extension, language unification, self-extension and extension composition (Méndez-Acuña et al., 2016). However, reuse opportunities are limited due to behaviors being tied to a specific concept. Some reuse can be achieved by concept extension/specialization and overriding, but the lack of multiple inheritance prevents the extension composition capability without a proper refactoring using the composite pattern.

Neverlang. Neverlang (Cazzola, 2012; Cazzola and Vacchi, 2013; Vacchi and Cazzola, 2015; Cazzola and Shaqiri, 2017) is a language workbench for the development of programming languages compilers, interpreters and their ecosystem in a modular way. The Neverlang development cycle is based on the *language feature* concept. 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. Modules can also piggyback meta-data for the construction of the language ecosystem—such as an IDE. The Automatic Integrated Development Environment (AiDE) for Neverlang adds support for language product line development (Kühn et al., 2015; Kühn and Cazzola, 2016; Favalli et al., 2020). It copes with all aspects of the development of a language family by supporting three roles: the language developer, the language deployer and the language user. According to the taxonomy by Erdweg et al. (2012), Neverlang supports language extension, language unification, self-extension and extension composition (Méndez-Acuña et al., 2016). However, The Neverlang DSL composition mechanisms are too coarse—variability between variants of a language family is at language feature granularity.

Spoofax. Spoofax (Wachsmuth et al., 2014; Voelter, 2013; Kats et al., 2010) is a language workbench that provides several DSLs for language development. Most importantly, Syntax Definition Formalism (SDF3) is used for (possibly ambiguous) grammar specification and Stratego for the semantics as a sequence of AST transformations called *rules* and *strategies*. The abstract syntax of a program is represented in a data structure and stored in a format called ATerm. According to the taxonomy by Erdweg et al. (2012), Spoofax supports language extension, language unification, self-extension and extension composition (Méndez-Acuña et al., 2016). Performing semantic reuse in Spoofax may require additional glue code for the explicit transformation of types—both those involved in the pattern matching and the return types.

Rascal. Rascal (Klint et al., 2009b; Basten et al., 2015; Klint et al., 2019) is a meta-programming language for the development of language processing tools. The abstract syntax is defined using algebraic data types. Rascal library supports the parsing of textual input and its *implosion* to perform tree transformations. ASTs can be evaluated based on user-defined functions and pattern matching of the algebraic data types on function arguments (a technique called *pattern-based dispatch* Basten et al., 2015). Rascal workflow and capabilities are very similar to Spoofax with the same benefits and limitations.

CBS. CBS (Churchill et al., 2015; Mosses, 2019b,a) is a language workbench partially developed in using Rascal for the component based development of programming languages and DSLs. CBS is based on the concept of funcons. Funcons are modular components that can be reused in the specification of several languages (Churchill et al., 2014; van Binsbergen et al., 2019). The PlanCompS (van Binsbergen et al., 2016) aims to collect a substantial library of funcons as a basis for the development languages in CBS. To the best of our knowledge CBS does not offer language construct to explicitly encapsulates developed components (apart from funcons). Therefore reuse may be difficult. However, it should be noted that CBS is still in the early stages of development.

Truffle. Truffle (Wimmer and Würthinger, 2012; Würthinger et al., 2012; Latifi et al., 2021) is an library to build language implementations and interpreters. Truffle applies rewriting to abstract syntax tree nodes to perform optimizations and semantic operations. Truffle combined with the GraalVM (Würthinger et al., 2013; Würthinger, 2014) has been used successfully in variety of projects (Niephaus et al., 2019; Kloibhofer et al., 2020). However, Truffle does not provide the tooling necessary to parse or define the language syntax which are delegate to external tools. This approach limits the reusability of syntactic components.

All things considered. To the best of our knowledge, language workbenches should strive to provide abstractions that enable opportunistic reuse at all levels of granularity. In our view, reuse should not be limited by the definition of visitors for specific ADTs. Language components should be reusable in any scenario unless stated otherwise since the developer cannot foresee all the future use cases that may arise—any limitation should be declared explicitly through preconditions and/or post-conditions. Therefore, language workbenches may benefit from providing constructs to adapt existing semantics to new scenarios without requiring changes or re-compilation of existing source code.

3.3. Language product lines

Software product lines. Product lines are a staple in industrial production. Following the same ideas, 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.

Feature model. 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 (ter Beek et al., 2019) such as FeatureIDE (Thüm et al., 2014), software engineers can cope with all the aspects of the development of a software family: domain analysis, domain implementation, requirements analysis, and product derivation.

Language product lines. The idea of applying SPL concepts to the creation of language families 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). This approach may prove useful to the creation of both variants of the same DSL (Crane and Dingel, 2005; Tratt, 2008; Vacchi et al., 2014) and *dialects* of a GPL (Cazzola and Olivares, 2016).

3.4. The Neverlang ecosystem

Overview. In this section we will introduce the Neverlang ecosystem to discuss how it can be used to develop programming languages (Section 3.4) and their evolution (Section 4.1). As a running example, we will use LogLang. 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. Listing 1 shows part of this implementation, including the syntax and semantics of the backup utility and any code necessary to compose this utility into a complete language specification. The Neverlang language workbench can be viewed as a collection of constructs that are used to develop different aspects of language compilers and their ecosystems. Listing 1 shows several of the Neverlang constructs grouped by color. Each marker between ❶ and ❾ represents a different construct. Each color represents a different concern with regards to the modeling of programming languages and their ecosystems:

1. green (frame ❷)—modeling syntax concern;
2. purple (frames ❹, ❻, ❾)—modeling semantics concern;
3. black (frames ❺, ❼, ❽)—modeling composition concern;
4. red (frame ❸)—modeling IDE concern;
5. blue (frame ❶)—modeling feature variability concern.

In this example we show only a portion of a language implementation. This section serves as an introduction to the Neverlang language workbench. Please refer to Vacchi and Cazzola (2015) for a full overview.

```
1  module Backup {
2    reference syntax {
3      provides { Backup: backup, statement;                    ❶
4                 Cmd   : statement;          }
5      requires { String;                      }
6
7      Backup <— "backup" String  String;                       ❷
8      Cmd    <— Backup;
9
10     categories  : Keyword = { "backup" };                    ❸
11     in-buckets  : $1 <— { Files }, $2 <— { Files };
12     out-buckets : $1 —> { Files }, $2 —> { Files };
13   }
14   role(debug) {
15     0 .{
16       $0.isExecutionStep = true;
17     }.
18   }
19
20   role(execution) {                                          ❹
21     0 .{
22        String src  = $1.filename;
23        String dest = $2.filename;
24        $$FileOp.backup(src, dest);
25     }.
26   }
27 }
28
29 slice BackupSlice {
30   concrete syntax from Backup                                ❺
31   module Backup          with role execution
32   module BackupPermCheck with role permissions
33
34     mapping { 0 ⇒ 2, 1 ⇒ 3, 2 ⇒ 4,                           ❻
35               3 ⇒ 0, 4 ⇒ 1          }
36 }
37
38 bundle Tasks {
39   slices BackupSlice RemoveSlice                             ❼
40          RenameSlice MergeSlice
41 }
42
43 language LogLang {
44   slices bundle(Tasks)  Task                                 ❽
45          Main           Types
46   endemic slices FileOpEndemic PermEndemic
47
48   roles syntax      < terminal-evaluation                    ❾
49     < permissions : execution
50 }
```

Listing 1: Language modeling concerns of the LogLang implementation in Neverlang.

Modeling syntax. The `reference syntax` section (Listing 1-frame ❷) is used to model the syntax of programming language features. Neverlang is based on attribute grammars and the `reference syntax` provides a hook to which semantic actions are bound. The productions defined in a `reference syntax` usually become part of the grammar of the final language and the semantic actions in the same module are used only when the specific nodes generated as a result of those productions are encountered in the AST. Nonterminals are numbered from left to right, from top to bottom starting from 0 and can be referenced with the $ operator. Therefore Listing 1 shows two productions and 5 nonterminals: `$0` refers to the `Backup` nonterminal, `$1` and `$2` are the two `String` nonterminals, `$3` refers to `Cmd` and `$4` refers to the second `Backup` nonterminal.

Modeling semantics. The `role` construct (Listing 1-frame ❹) can bind semantic actions (lines 21–25) to nonterminals in the productions, referenced with the `$` notation. The semantic action for the current compilation phase is performed when the corresponding production from the reference syntax is recognized in the AST. Semantic actions are implemented in Java by default, but Scala, Kotlin and Ruby are also supported. Semantic action execution is optional and can be prevented through arbitrary guards, following the *restraint semantic dispatch* (RSD) model (Cazzola and Shaqiri, 2016). Semantic actions can access or provide attributes to the grammar using the *syntax directed translation* technique (Aho et al., 2006). The order in which the roles – and thus the semantic actions – are executed is declared by the `roles` clause of a `language` unit (Listing 1-frame ❾). A role included in a slice can be adapted to a syntax different to the originally intended one by appending a `mapping` (Listing 1-frame ❻) directing to the role inclusion.

Modeling composition. In SPL engineering a software product is the result of three steps: (i) requirements analysis, (ii) features selection, and (iii) product derivation. The result of step (ii) is usually called *configuration* whereas step (iii) takes a configuration and feeds it to a program called *composer* to generate a product. Neverlang is no different: each product is the result of the execution of the AiDE (Vacchi et al., 2013, 2014; Cazzola and Favalli, 2022) composer on a valid configuration. Slices combine syntactic and semantic aspects of modules from the FM into a concrete language feature (Listing 1-frame ❺). Bundles collect several slices (Listing 1-frame ❼). Finally, languages are the result of the composition of several slices and/or bundles into a complete and executable language specification (Listing 1-frame ❽).

Modeling development environments. The IDE modeling capabilities of Neverlang were first introduced in Kühn et al. (2019). Modules piggyback IDE service specifications through the `categories`, `in-buckets` and `out-buckets` constructs (Listing 1-frame ❸). Categories are based on the `reference syntax` and model the syntax highlighter for a language. Similarly, buckets model the auto-completion aspect of an editor by feeding and retrieving tokens from a named pool. Semantic services are instead integrated with the semantic actions block that can be used to model the execution steps of a debugger. Both syntactic and semantic services are fully LPL-driven: given the modular nature of Neverlang, the actual implementation of an IDE arises when models are composed into a language specification. In this context, for each language variant of the LPL, a different IDE variant can be generated.

3.5. Terminology

Production. Following the definition of Aho et al. (2006), a production is a pair (H, B) where H is a nonterminal and B is a sequence of terminals and/or nonterminals. Productions are usually expressed as elements of an BNF grammar. For example:

```
Addition <— Term "+" Term;
```

Action. An action is a function $f : \mathcal{A} \times \mathcal{C} \to \mathcal{C}$ where $\mathcal{A}$ is the set of all possible ASTs and $\mathcal{C}$ is the set of *evaluation contexts*. The *evaluation context* refers to the program state in which the action is executed. It includes the type and value of local and global variables, as well as the scope and any return values. For example, the following, is an action in Neverlang that performs the sum between two terms.

```
$0.value = $1.value + $2.value;
```

The parameters and return values are made implicit. Language workbenches implement actions in several different ways such as Neverlang's semantic actions, Spoofax strategies and MPS behaviors.

Syntactic asset. A syntactic asset is a collection of productions. Usually a syntactic asset is a subset of the BNF grammar for a language representing a language construct. For example, the following is a syntactic asset representing a non-empty list.

```
List <- "[" Seq "]";
Seq  <- Seq "," Elem;
Seq  <- Elem;
```

Semantic asset. A semantic asset is a collection of actions. Usually, a semantic asset can be implemented as a visitor[4] Gamma et al. (1995) for an AST and represents the semantics for a syntactic asset. For example, the following is a semantic asset implemented as a Scala visitor that populates a non-empty list.

```
0 <scala> .{ $0.list = $1. list        }.
2 <scala> .{ $2.list = $3.list :+ $4.elem }.
5 <scala> .{ $5.list = List($6.elem)     }.
```

Language feature. A language feature (Vacchi and Cazzola, 2015; Cazzola et al., 2018) is a pair (ϕ, ψ) where ϕ is a syntactic asset and ψ is a semantic asset. A language feature represents a language construct together with its behavior. For example, the following is a language feature for non-empty lists implemented by composing the syntactic and semantic assets from the previous examples.

```
slice List {
  concrete syntax from ListSyntax
  module ListEval with role evaluation
}
```

Sub-language. A sub-language (Cazzola et al., 2018) is a collection of language features. The notion of sub-languages usually refers to a specific concern of a host language such as numerical expressions or variable definition. A sub-language relies on the presence of other sub-languages to be usable. For example, the following is a sub-language dealing with the concern of variable definition and their assignment.

```
bundle VariablesConcern {
  slices
    Identifiers
    VariableStatement
    AssignmentOperators
    AssignmentExpression
}
```

Language. A language is a collection of sub-languages. A language is represented in terms of its syntax (the entire grammar) and its semantics. For example, the following is a snippet of the ECMAScript language definition.

```
language EcmaScript {
  slices
    bundle (CoreConcern)
    bundle (BasicMathConcern)
    bundle (VariablesConcern)
    /*...*/
  roles syntax <+ debug <+ evaluation
}
```

In this example, the `VariablesConcern` sub-language, defined earlier, is completed by composing it with other sub-languages. In this case composing `CoreConcern`, `BasicMathConcern`, and *etc.* results in ECMAScript. However, different compositions result in different languages (look at Cazzola and Olivares (2016) for details).

4. Modeling reuse

While the taxonomy proposed by Erdweg et al. mainly focuses on the composition between entire languages or between a language and a language extension, both yielding a new language, smaller linguistic components can also be composed to allow for additional reuse opportunities. In fact, language workbenches can support composition mechanisms in which neither of the composed elements nor the final result are a complete language or even a language extension. For example, the action performing the addition could be composed with the action performing the type checking, yielding an action that performs both. Erdweg et al. addressed the problem of growing a language modularly via the *extension unification* mechanism (Erdweg et al., 2012) in which two independent extensions can be composed and used together by using some glue code. However, modular growth of programming languages may benefit from reusing linguistic components beyond language extensions, in favor of a more granular composition mechanism.

Consider the previously discussed linguistic components: actions, syntactic assets, semantic assets, language features, sub-languages, and languages. Each of these components represents a viable level of reusability, even though the result of the composition cannot be considered a language extension since it is not complex enough to be directly used in a complete language. Recall the above example: the composition of two actions reuses small components to yield a new action that is not a complete language extension, since it lacks any syntax. Conversely, reuse is limited to potentially very large components when it can be performed on languages and language extensions only: in this case, fine-grained reuse must resort to clone-and-own.

To fill this gap, in this section we discuss some examples of composition mechanisms that operate at each level of granularity. Then, we will showcase how the Neverlang language workbench can benefit from introducing initially unsupported composition mechanisms and answer RQ_1 based on this discussion.

4.1. Workbench agnostic linguistic reuse

Sub-languages composition. Given $\{Ł_1, \ldots, Ł_n\}$ a set of sub-languages taken from a set of base languages, a language workbench supports sub-languages composition if we can create a language L by joining any subset of $\{Ł_1, \ldots, Ł_n\}$ without changes to any $Ł_i$.

Neverlang supports sub-languages composition through the `language` compilation unit (Listing 1-frame ❸). For example, the following shows an extended version of ECMAScript with the `NumSeparators` sub-language which implements numeric separators.[5]

```
language EcmaScriptExtended {
  slices
    bundle (EcmaScriptCore)
    bundle (NumSeparators) //From another language
  roles syntax <+ debug <+ evaluation
}
```

[4] Visitors define the semantics to be performed on the elements of an AST data structure. Visitors visit each node and execute all the associated actions without changing the implementation of the nodes themselves.

[5] Numeric separators are a language feature that improves the readability of numbers with many digits. For example, `10000` can be written in ECMAScript as `10_000` where the `_` separators can be placed only to group the digits per thousand.

Language feature composition. given a set of language features $\{l_1, \ldots, l_n\}$, a language workbench supports language feature composition if we can create a sub-language Ł by joining any subset of $\{l_1, \ldots, l_n\}$ without changes to any l_i.

Neverlang supports language feature composition through the `bundle` compilation unit (Listing 1-frame ❼). For example, the following shows the `NumSeparators` sub-language created by the composition of the language features `FloatSeparators` and `IntegerSeparators`.

```
bundle NumSeparators {
  slices
    FloatSeparators
    IntegerSeparators
  rename {
    Double -> NumericLiteral;
    Integer -> NumericLiteral;
  }
}
```

The `rename` compilation unit is used to solve any syntactic incompatibility issues among the composed language features by redefining any nonterminals in the sub-languages grammar.

Syntactic and semantic assets composition. Given a syntactic asset ϕ and a semantic asset ψ, a language workbench supports syntax and semantics composition if we can create a language feature l by composing ϕ and ψ.

Neverlang supports syntax and semantics composition through the `slice` compilation unit (Listing 1-frame ❺). For example, the following composes the `DoubleSyntax` syntactic asset with the evaluation semantic asset from the `DoubleEvaluation` module.

```
slice FloatSeparators {
  concrete syntax from DoubleSyntax
  module DoubleEvaluation with role evaluation
  module Separators with role desugar
}
```

Neverlang requires syntax and semantics assets to refer the same number of productions, each with the same number of nonterminals. The main limitation of this approach is that semantic assets can be difficult to reuse outside of their original intended syntax. However, the composition can be accommodated using mappings: the `mapping` compilation unit (Listing 1-❽) can be used to reorder the nonterminals for a specific semantic asset if required to match the given syntactic asset.

Semantic assets composition. Given two semantic assets ψ_1 and ψ_2 a language workbench supports semantic asset composition if we can create a visitor ψ_3 by chaining the semantics of both ψ_1 and ψ_2.

Neverlang supports semantic asset composition through the `slice` compilation unit (Listing 1-frame ❺). For example, the `FloatSeparators` slice – from the previous example – composes the evaluation semantic asset from the module `DoubleEvaluation` with the `desugar` semantic asset from the `Separators` module. Notice that the `slice` compilation unit can be used to perform both Semantic asset composition and Syntax asset composition.

Action composition. Given a set of actions $F = \{f_1, \ldots, f_n\}$, a language workbench supports action composition if we can create a semantic asset by chaining any number of actions from F, in any order and with any number of repetitions.

An example of action composition mechanism is *desugaring*: any AST sub-tree can be converted to a different sub-tree to compose existing semantic actions. Neverlang does not support desugaring nor any other form of action composition, although desugaring is supported by other language workbenches such as Rascal. However, Neverlang could be extended to support desugaring as follows: the `desugar` role of the Separators module performs a tree rewriting that swaps the `$0` node of the `IntegerSyntax` module with a node created by taking the production in position 0 of the `Literal` reference syntax.

```
module Separators {
  reference syntax from DoubleSyntax
  role(desugar) {
    0 <desugar> .{
      Literal[0]($0.value)
    }.
  }
}
```

This way, all the actions defined for the `Literal` syntax can be reused.

Action extension. Given an action f, a language workbench supports action extension if we can create an action f' by extending the semantics of f and without changes to f itself.

An example of action extension is overriding: the semantics of an existing implementation are composed with new semantics from a subclass. For instance, MPS behaviors can override other existing behaviors. Neverlang does not support action extension through overriding nor any other construct. However, we can extend Neverlang with a semantic delegation operator (►). In the following example, the semantic action in position `$0` of role evaluation from module `Separators` is implemented by performing the semantic action in position `$0` of role terminal-evaluation from module `DoubleEvaluation`, then translating the `$0.value` grammar attribute into a representation accepted by the ECMAScript interpreter (line 6).

```
module Separators {
  reference syntax from DoubleSyntax
  role(evaluation) {
    0 @{
      DoubleEvaluation►terminal-evaluation[0];
      $0.value = new JSNumber((Double)$0.value);
    }.
  }
}
```

The discussed composition mechanisms extends the Erdweg taxonomy by introducing dedicated types of compositions for different components of languages.

4.2. Linguistic reuse in Neverlang

In the previous section we discussed the composition mechanisms provided by Neverlang. These mechanisms were part of previous works (Vacchi and Cazzola, 2015), thus we will not discuss them in detail. However, Neverlang lacks constructs to support action composition and action extension. In this work, we complemented the Neverlang reuse capabilities by adding explicit constructs to support action composition and action extension, so that Neverlang can perform reuse at all levels of granularity. In this section, we discuss how these constructs fit the classes of our taxonomy.

Desugaring. Desugaring is an instance of action composition. It can be seen as a tree rewriting technique and is well-known in the literature (Krishnamurthi, 2015; Lorenzen and Erdweg, 2016; Erdweg et al., 2011). Intuitively, desugaring is a technique to strip any *syntactic sugar* from a language to ease the compilation

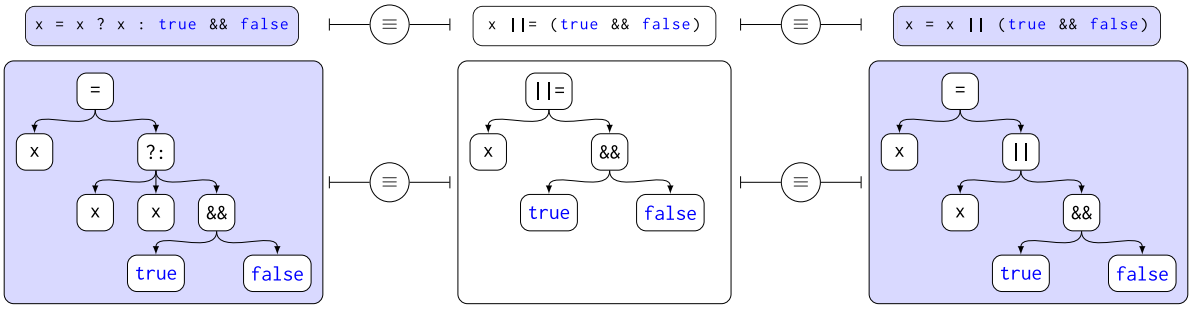

Fig. 2. The central panel shows the AST relative to the JavaScript expression `x ||= (true && false)`. The ASTs yield the same final *evaluation context* when executed in the same initial *evaluation context*. Replacing either one with the others in a program will not affect the results. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

and simplify the compiler back-end. This is done by replacing an AST node with a sub-tree equivalent made of simpler language features. Desugaring is a novel introduction in Neverlang and has the advantage – compared to other language workbenches – of not being tied to any ADTs. This means that: (i) mapping a semantic action to a new type of node does not require any glue code and (ii) any child of an AST node can be swapped with any other sub-tree with no restrictions. These advantages should provide more flexibility with regards to opportunistic reuse when extending programming languages with new constructs.

Let us now take up the example introduced in Section 2 (Fig. 1). To leverage any reuse opportunity toward the development of the new language feature, we must first notice that any expression using the `||=` operator can be rewritten by desugaring into a combination of two pre-existing features from the JavaScript LPL. For instance, take the following expression into account:

```
x ||= (true && false)
```

In all the following examples we will assume that `x` is assigned to `true` in the initial *evaluation context*. There are a few possible AST transformations that yield the same final *evaluation context*—i.e., both the expression and the `x` variable evaluate to `true`:

1. using the `JSTernaryExpression` and the `JSAssignExpression` language features[6]

```
x = x ? x : true && false
```

2. using the `JSOrExpression` and `JSAssignExpression` language features

```
x = x || (true && false)
```

Fig. 2 shows the original expression (central panel) and both transformations (left and right panel) as well as their respective ASTs. Henceforth, we will stick to the second option even if the two transformations are semantically equivalent.[7]

In Neverlang version 2.1.2 – the version prior to the one presented in this work – none of the constructs reported in Section 3.4 could solve the code duplication nor the evolution problems introduced in Section 2 thus Neverlang 2.1.2 does not support action composition. The ability to optimize ASTs by rewriting for better performance is well-known (Würthinger et al., 2012). Initially, Cazzola and Shaqiri (2016) presented a dynamic tree rewriting-based technique to optimize the Neverlang RSD mechanism for multiple semantic actions. The proposed API included primitives to swap AST nodes, prune sub-trees and define new semantic actions. However the applicability of the same technique to perform reuse of linguistic components was not explored, nor was there a Neverlang construct that natively supported desugaring. Thus, we introduced action composition capabilities in Neverlang 2.2.0 through a dedicated *desugar* DSL. Listing 2 shows how it can be used to describe the `||=` operator in terms of the `=` and `||` operators by rewriting the AST in a new `tree-rewrite` role. Once the AST is rewritten, the change will be permanent and the composed actions will be used on the modified AST fragment. Clone-and-own solutions should no more be needed: any changes made to the `JSAssignExpression` and/or `JSOrExpression` features and their actions will propagate to the `OrAssignment` feature. Moreover the desugaring affects any subsequent tree traversal, thus the action composition is propagated to all roles defined on the `JSOrExpression` and `JSAssignExpression` features. This includes roles that may be implemented in the future. For example one might want to log all Boolean expressions to spot places in which the execution path diverges. Adding a `logging` role to the `JSOrExpression` feature will cause the logging to be performed on the `||=` operator too. It should be noted that chances of performing static analysis are also severely limited, both on the validity of the translation and on the soundness of the attribute grammar: the substitute AST is stored in the Neverlang run-time whereas any link to the original AST is compromised, which may be problematic for IDE features such as debugging and refactoring. For example, desugaring may compromise information about breakpoints as

```
1  module OrAssignment {
2    reference syntax {
3      AssignExpr <- LeftExpr "||=" AssignExpr;
4    }
5    role(tree-rewrite) {
6      0 <desugar> .{
7        JSAssignExpression[1] (
8          $1,
9          JSOrExpression[1] ($1, $2)
10       )
11     }.
12   }
13 }
```

Listing 2: `||=` operator semantics implemented via desugaring.

[6] Notice that the left-hand side of the `||=` assignment allows only for identifiers. Therefore, rewriting the AST so that the left-hand side is evaluated more than once cannot cause side effects. This is generally not true for languages in which the left-hand side of an assignment can cause side effects.

[7] Here, equivalence means that the evaluation of both ASTs (or their respective source code) yields the same final *evaluation context* when executed in the same initial *evaluation context*.

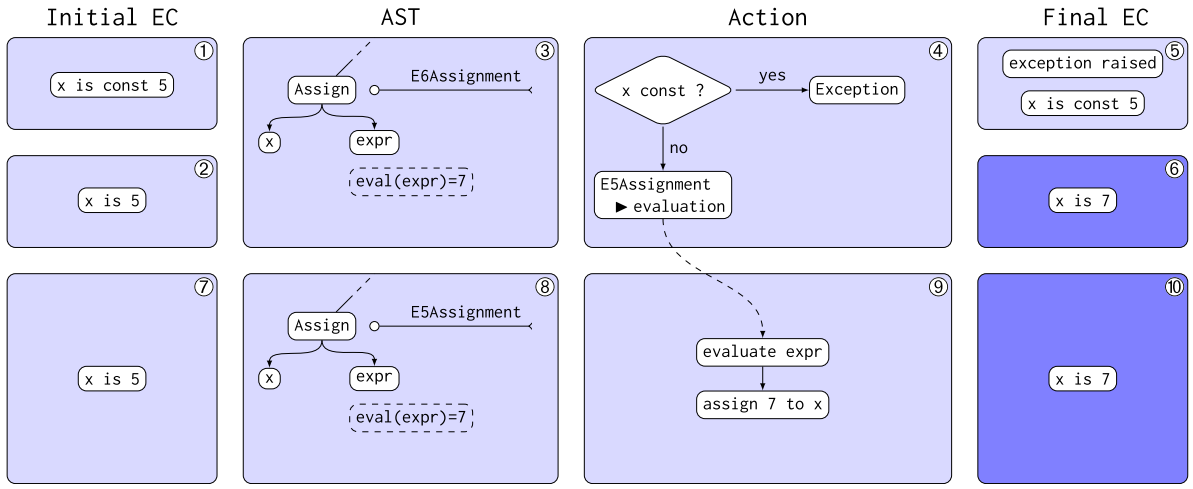

Fig. 3. ① and ② show two possible evaluation contexts (EC) for ECMAScript 6. ③ shows the AST of an assignment. ④ shows the behavior of the `E6Assignement` action. `E6Assignement` controls that the assigned variable is non-constant and performs the assignment. `E6Assignement` delegates parts of its behavior to a previous version of ECMAScript—`E5Assignement` (⑨). ⑤ shows the final EC given the initial EC shown in ①: An exception is raised since `x` was declared constant. ⑥ shows the final EC given the initial EC shown in ②: the value 7 is correctly assigned to the variable `x`. ⑦ shows an initial EC for ECMAScript 5. ⑧ shows the AST of an assignment. ⑨ shows the behavior of the action `E5Assignement`. Finally, ⑩ shows the final EC given the initial EC shows in ⑦.

```
1  module ConstAssign {
2    reference syntax {
3      AssignExpr <— LeftExpr "=" AssignExpr;
4    }
5    role (evaluation) {
6      0 .{
7        eval $1
8        JSReference lref = $1.value;
9        if(isConstant(lref.getName()))
10         throw new ConstantException();
11       else
12         assign▶evaluation[2];
13     }.
14   }
15 }
```

Listing 3: Implementation of constant-aware assignment in Neverlang. If the considered reference is not a constant, the evaluation is delegated to the standard assignment.

the AST sub-tree is swapped with another unless specific precautions are taken (Lindeman et al., 2011). In exchange, each time desugaring is performed, we can assume the (bidirectional) semantic equivalence between two ASTs: any source code using the old features can be automatically migrated to the new syntax and back without changing the semantics. This can be useful in the scope of language evolution because it helps with the migration of old pieces of software to new language versions.

Delegation. In our taxonomy, the finest level of granularity of linguistic reuse is action extension, because any action can also be viewed as an extension of the empty action. From this perspective, action extension is more versatile then overriding because the latter is limited to extension of actions of a specific type. One possible way to implement action extension is by taking an existing action and extending it with new functionalities. In this work, we extended Neverlang to achieve this result through *delegation*. In practice, one can extend and/or combine multiple actions into a new one by delegating the evaluation to said actions. Neverlang delegation fits the definition of action extension. However, due to the finer granularity, the win in productivity is more limited than other composition mechanisms (Xu et al., 2011; Maga et al., 2011). Consider that in Neverlang, delegation can be thought as an instance of action composition as well: it can be used to combine multiple actions into one. For example, in Fig. 3, one could have delegated exception throwing to an existing action. However, as we will see, each of the two mechanisms (desugaring and delegation) has advantages over the other.

Let us consider a simple example one may encounter in language development—illustrated in Fig. 3 and implemented in Listing 3: the introduction of constants in ECMAScript 6 in 2015. The assignment operator for ECMAScript 6 must take into account the existence of constants to prevent any constant to be re-assigned. However we want to avoid code duplication since variable assignment already exists in the language. To leverage this reuse opportunity, this new feature can be implemented with a simple check on the execution environment (Fig. 3-④ and line 9 of Listing 3), then delegate to the pre-existing assignment semantics if the reference is not a constant (line 12). Otherwise, an exception must be thrown (line 10). Yet, it is impossible to implement this feature using desugaring—because accessing the execution environment is impossible in ECMAScript 6. Therefore, we need a different mechanism to address this extension without neglecting the opportunity for reuse.

With delegation, we can execute an existing action that was defined elsewhere, on the current node and evaluation context. Following the example in Fig. 3, we can create a new action that (i) checks the execution environment; and (ii) reuses the existing variable declaration action. The new feature is extended naturally from the existing one. As the features evolve, the changes are naturally propagated throughout the whole language.

Clone-and-own. To leverage opportunistic reuse, one can consider a third strategy: clone-and-own. Clone-and-own consists of copying and pasting snippets of code to reuse their content. It is a simple and efficient technique. It can perform both action composition and action extension. Moreover, it requires no modification to the Neverlang workbench to be performed. Yet, it has several disadvantages with regards to code maintainability and readability (Rubin et al., 2013). Fig. 1 shows an example of clone-and-own other features. To implement the new language feature `OrAssignment`, snippets from `JSOrExpression` and `JSAssignExpression` are copied, introducing several instances of code duplication. We can consider the clone-and-own approach as a simple baseline against which desugaring

Table 2
Comparison between extension strategies: clone-and-own (C&O), desugaring (DS) and delegation (DG).

Reuse property	C&O	DS	DG
Action extension	✓	✗	✓
Action composition	✓	✓	✓
Removes code duplication	✗	✓	✓
Changes are automatically propagated	✗	✓	✓
Automatic code migration	✗	✓	✗
Tool-assisted code migration	✗	✓	✓
Attribute grammar can be checked statically	✓	✗	✓
Affects subsequent compilation phases	✗	✓	✗
Preserves link between AST and source code	✓	✗	✓
Can perform AST optimizations	✗	✓	✗
Explicit semantic feature dependency	✗	✗	✓
Fine grained semantic variability	✓	✗	✓
Natively supported in Neverlang	✓	✓	✗
Language implementation independent	✗	✗	✓
Reference syntax independent	✗	✓	✗
Always applicable	✗	✗	✗
No additional manipulation required	✗	✗	✗

and delegation can be compared to determine if explicit reuse mechanisms bring any improvements.

Comparison. Let us discuss the differences among desugaring, delegation, and clone-and-own, whose strengths and weaknesses are summarized in Table 2. Clone-and-own can always perform both action extension and action composition—given the original source code is available. Delegation can also perform both action extension and composition. Meanwhile, Desugaring cannot be used to perform action extension. Consider the example shown in Fig. 3, checking whether x is a variable or a constant requires direct access to the symbol table, a feature which is usually not natively supported in programming languages. Despite this disadvantage desugaring is the most natural way to define new language features. Being based on the semantic equivalence between ASTs, desugaring enables the automatic translation from one AST to another and vice versa. However, it should be noted that performing desugaring at run-time breaks the link between the original AST and the substitute one, which may be problematic for IDE services such as refactoring and debugging. Instead, delegation and clone-and-own can introduce arbitrary changes to the semantic actions, therefore semantic equivalence cannot be guaranteed: code migration can only be tool-assisted, for instance in a dedicated source editor. Desugaring is dynamic and requires an additional preprocessing tree traversal to substitute the AST. Desugaring can be used to perform optimizations on the AST and has the advantage of automatic propagation to all subsequent roles. Delegation and clone-and-own must be implemented on a per-semantic action basis. The trade-off is the reduced capability to perform static analysis. This includes the soundness of the attribute grammar and feature dependencies. Moreover, Neverlang relies on a cache to optimize the access to AST sub-trees in semantic actions, but this cache is invalidated by the delegation process. In exchange, we obtain a language-independent mechanism that enables fine-grained variability in LPLs. Given two LPLs with the same number of features, the number of valid language configurations should be increased in the LPL that applies this technique.

On a surface level, there is no clear winner between desugaring and delegation: both approaches solve the problems caused by clone-and-own but – as we will show in Section 5 – neither can be applied to all use cases. Moreover, both approaches may require additional manipulation of the AST to be effectively used. Therefore, the usage of either techniques should be considered on a case-by-case basis.

4.3. Language composition taxonomy

We can now answer RQ_1.

What are the levels of granularity needed to support reuse of software artifacts in the implementation of programming languages?

We identified six levels of granularity in the implementation of programming languages, then we defined composition mechanisms for each language component. Together with the four language composition mechanisms presented in Erdweg et al. (2012), developers can adopt a wide range of mechanism to foster opportunistic reuse at all levels of granularity. Sorted from coarse to fine grained, those composition mechanisms are:

1. language unification
2. language extension;
3. self-extension;
4. extension composition;
5. sub-language composition;
6. language feature composition;
7. syntax and semantics composition;
8. semantic asset composition;
9. action composition;
10. action extension.

The linguistic reuse capabilities of language workbenches can be evaluated against this framework. For instance, Neverlang was found to provide constructs to perform linguistic reuse only for points 1 to 8 prior to this research, whereas point 9 and 10 were introduced as a part of this work. Similarly, other language workbenches could be evaluated against the same framework to estimate their ability to support reuse of linguistic components. For a language workbench to support linguistic reuse at all levels of granularity, it means for it to provide explicit constructs for all ten points.

5. Evaluation

Overview. In the previous section, we focused on reuse mechanisms that can be used to achieve action composition and action extension: desugaring, delegation, and clone-and-own. Here, we evaluate these mechanisms with real-world use cases. We compare the quality of the produced source code with regards to several metrics. The goal of this evaluation is to answer RQ_2, which involves both qualitative and quantitative aspects. From a qualitative standpoint, we want to investigate which kind of features can be implemented with reuse as a unique concern. From a quantitative standpoint, we want to measure the effort of applying reuse strategies when modeling the evolution of LPLs.

We collected several features introduced in ECMAScript 3 or later. We implemented (whenever possible) each feature using clone-and-own, delegation, and desugaring approaches. The clone-and-own represents an implementation made from scratch which does not reuse any other language features neither semantic nor syntactic. To keep the implementations consistent and to avoid bias, clone-and-own implementations are generated. The generation process takes the delegation (or the desugaring) implementation and substitutes the reference to the reused semantics with referenced source code.

We only evaluate desugaring and delegation with regards to clone-and-own baseline as they are a novel introduction in Neverlang.

Data setup. Starting from a base ECMAScript 3 implementation written in Neverlang (Cazzola and Olivares, 2016), we manually examined the changelogs of ECMAScript since 2015. From the changelogs, we extracted a set of 10 language features to be used as a demonstration case study. Each considered feature (Table 3) should have the following characteristics:

Table 3
ECMAScript features selected for the evaluation. On the last column, the existing language features that were reused to implement the new ones.

Feature	Year	Uses
`ArrowFunction`	2015	`JSFunctionExpression` `JSReturnStatement`
`Const`	2015	`JSVariableStatement`
`CheckAssignExpression`	2015	`JSAssignExpression`
`OptionalCatchBinding`	2019	`JSTryCatch`
`OrAssignment`	2021	`JSOrExpression` `JSAssignExpression`
`AndAssignment`	2021	`JSAndExpression` `JSAssignExpression`
`NullishCoalescingAssignment`	2021	`NullishCoalescing` `JSAssignExpression`
`IntegerSeparators`	2021	`IntegerSyntax` `DoubleEvaluation`
`NumericSeparators`	2021	`DoubleSyntax` `DoubleEvaluation`
`NullishCoalescing`	2021	`JSOrExpression`

- it must have been introduced in 2015 or later;
- it must not be an extension to the API;
- it must not require changes to the original implementation of ECMAScript 3 (from Cazzola and Olivares, 2016);
- it must be overlapping with existing features;
- it must be a composition of the existing features of ECMAScript 3 or with other considered features.

Original implementation cannot be changed to adhere to the definition of language development system in Erdweg et al. (2012), Leduc et al. (2020). Changing the original implementation might enable additional opportunities for reuse, but we measure the benefits of introducing reused strategies to a project which components are not designed to be reused. Of course, designing the entire project around reuse can bring further benefits.

With the last two points we would like to stress that we are not measuring the design overhead of detecting reuse opportunities in the development of programming languages nor the effort of refactoring a project to improve reusability. We focus on the implementation aspects of semantic reuse in a situation in which the opportunity for reuse was already highlighted during the design phase instead.

Metrics. Over the years, the software engineering community has proposed several source code metrics with the purpose of measuring several aspects of the source code qualities. To investigate the considered approaches – clone-and-own, desugaring, and delegation – we adopt the following metrics:

- Lines of Code (LoC): LoC measures the size of modules by counting the number of lines in the source code. LoC gives a quick estimate of the module complexity. The lower the better. Ideal value: 1.
- The number of roles (#Roles): #Roles measures the number of roles implemented in a module. It is the number of separate AST traversal required. The lower the better. Ideal value: 1.
- The number of semantic actions (#SA): #SA measures the number of semantic actions required to implement a feature. It measures the complexity of a feature. The lower the better. Ideal value: 1.
- Lack of cohesion in actions (LCOA): LCOA is the number of pairs of actions in the module that do not reference any attributes in common on the same nonterminal symbols. LCOA measures how many actions implemented in the same module actually belong to the same module. The lower the better. Ideal value: 0.
- Coupling between modules (CBM): A module is coupled with another if actions in either module refer to attributes which are also referred by the other module. CBM measures how many changes to a module affect the other modules. The lower the better. Ideal value 0.
- Cyclomatic Complexity (CC): the number of execution paths. CC measures the complexity of a code snippet. The lower the better. Ideal value 1.
- Maintainability Index (MI): MI measures how easy is to support new changes and it is measured as in Cazzola and Favalli (2022). The higher the better. $MI < 65$ is considered difficult to maintain. $65 \leq MI \leq 85$ is considered moderately difficult to maintain. $MI > 85$ is considered highly maintainable.
- Cognitive Complexity (CoCo): CoCo is a measure of understandability of the code. It is high when the branching factor of a program is high (the lower the better).

For a full overview of these metrics, please refer to Cazzola and Favalli (2022) and Campbell (2018), Lavazza et al. (2023) for the cognitive complexity.

5.1. Results

In Table 4, we show our measurements on all subject language features. When applicable, we show the total and the average aggregated results for each measurement. Clone-and-own features are implemented in 46.56 LoC on average (419 LoC total) against the 23.11LoC on average (208LoC total) of delegation features and 24.63 LoC on average (197 LoC total) of desugaring. The average reuse percentage is 43.37%for delegation features and 39.51% for desugaring features. The reuse percentage goes up to 74.46% for delegation and to 47.58% for desugaring if only semantic actions are considered. There are only two exceptions to this trend: `IntegerSeparators` and `NumericSeparators` for the desugaring in which reuse of semantic actions was -20.00% in both cases. This is caused by the fact that both `IntegerSeparators` and `NumericSeparators` have small semantic actions and the desugaring approach has a fixed cost (in terms of LoC) due to API calls. The same fixed cost is highlighted by the fact that the variance per semantic action is smaller ($0.21LoC^2$) but the average is higher (6.75LoC) for the desugaring approach compared to the delegation strategy.

Both delegation and desugaring strategies are fairly maintainable and achieve or get close to achieve the ideal value in #Roles, #SA, LCOA, CC, and CoCo. With respect to MI, in Table 4 features in red are poorly maintainable, features in yellow are moderately maintainable and features in green are highly maintainable. The average MI is 90.20 for delegation and 88.33 for desugaring. Instead, the choice of the reuse strategy has limited effect on coupling: all three strategies achieve similar results regarding the CBM (8.67 for delegation, 9.63 for desugaring and 10.67 for clone-and-own on average). In particular, the CBM for `Assign` features is particularly high since they were created by composing at least two other features; moreover all `Assign` features operate on the same grammar fragment (that of expressions) and on the same attribute (`value`). As a result, all of them are coupled with one another, for a total of 16 coupled modules. CoCo is 0 for all semantic actions implemented using a desugaring or a delegation technique, since each of these actions can be implemented by performing a call to the Neverlang API and adding a few statements. The only exception is the implementation of `CheckAssignExpression` based on delegation, whose semantic

Table 4
Measurement results on the language features from Table 3.

Feature	Reuse Strategy	LoC (Neverlang)		LoC (Java)		Roles	Actions	LCOA	CBM	CC	MI	CoCo
		Total	%Reuse	Total	%Reuse							
AndAssignment	Delegation	26	46.94%	3	88.46%	1	1	0	16	1	87.70	0
	Desugaring	27	44.90%	5	80.77%	1	1	0	16	2	86.90	0
	Clone-and-own	49	0.00%	26	0.00%	1	1	0	16	5	72.52	3
OrAssignment	Delegation	26	45.83%	3	88.00%	1	1	0	16	1	87.70	0
	Desugaring	27	43.75%	5	80.00%	1	1	0	16	2	86.90	0
	Clone-and-own	48	0.00%	25	0.00%	1	1	0	16	5	71.76	3
Const	Delegation	16	15.79%	2	60.00%	1	1	0	1	1	95.87	0
	Desugaring	-	-	-	-	-	-	-	-	-	-	0
	Clone-and-own	19	0.00%	5	0.00%	1	1	0	1	1	88.46	0
CheckAssignExpression	Delegation	45	39.19%	12	70.00%	1	2	1	16	5	74.57	4
	Desugaring	-	-	-	-	-	-	-	-	-	-	–
	Clone-and-own	74	0.00%	40	0.00%	1	2	0	16	11	61.07	6
OptionalCatchBinding	Delegation	16	27.27%	2	77.78%	1	1	0	2	1	98.10	0
	Desugaring	20	9.09%	5	44.44%	1	1	0	1	2	92.94	0
	Clone-and-own	22	0.00%	9	0.00%	1	1	0	3	1	88.66	0
ArrowFunction	Delegation	24	25.00%	3	75.00%	1	3	3	2	3	88.54	0
	Desugaring	24	25.00%	5	58.33%	1	1	0	3	1	89.16	0
	Clone-and-own	32	0.00%	12	0.00%	1	3	3	4	3	80.40	0
NullishCoalescing	Delegation	-	-	-	-	-	-	-	-	-	-	0
	Desugaring	29	30.95%	5	72.22%	1	1	0	16	3	84.57	0
	Clone-and-own	42	0.00%	18	0.00%	1	1	0	16	2	74.74	2
NullishCoalescingAssignment	Delegation	23	57.41%	3	90.91%	1	1	0	16	1	90.02	0
	Desugaring	24	55.56%	5	84.85%	1	1	0	16	2	88.88	0
	Clone-and-own	54	0.00%	33	0.00%	1	1	0	16	5	67.76	3
IntegerSeparators	Delegation	12	63.64%	2	60.00%	1	1	0	5	1	101.50	0
	Desugaring	15	54.55%	6	-20.00%	1	1	0	5	1	97.28	0
	Clone-and-own	33	0.00%	5	0.00%	1	1	0	5	1	80.88	0
NumericSeparators	Delegation	20	69.23%	6	60.00%	1	3	0	4	3	87.79	0
	Desugaring	31	52.31%	18	-20.00%	1	3	0	4	3	80.02	0
	Clone-and-own	65	0.00%	15	0.00%	1	3	0	4	3	64.33	0
Average	Delegation	23.11	43.37%	4.00	74.46%	1.00	1.56	0.44	8.67	1.89	90.20	0.40
	Desugaring	24.63	39.51%	6.75	47.58%	1.00	1.25	0.00	9.63	2.00	88.33	0.00
	Clone-and-own	46.56	0.00%	20.33	0.00%	1.00	1.56	0.33	10.67	4.00	73.57	1.70
Variance per semantic action	Delegation	58.87	0.04	0.50	0.08	0.10	0.00	0.13	54.25	0.00	904.58	1.60
	Desugaring	42.30	0.03	0.21	0.17	0.06	0.00	0.00	52.87	0.50	509.52	0.00
	Clone-and-own	217.69	0.00	115.11	0.00	0.09	0.00	0.10	47.39	4.29	671.26	4.23
Total	Delegation	208	–	36	–	9	14	–	–	–	–	–
	Desugaring	197	–	54	–	8	10	–	–	–	–	–
	Clone-and-own	419	–	183	–	9	14	–	–	–	–	–

actions need the usage of an additional try-catch blocks, bringing the CoCo value to 4.

For most of the considered features, it was possible to apply all of the reuse strategies. The only exceptions are the `Const`, `CheckAssignExpression` and `NullishCoalescing` features. `Const` and `CheckAssignExpression` cannot be modeled using desugaring since syntactic modifications do no suffice. Instead, they extend the original `JSVariableStatement` and `JSAssignExpression` with a new endemic slice in which constants are stored. The `NullishCoalescing` feature cannot be modeled using delegation due to incompatibility of its `reference syntax` with the reused assets. In fact, the `??` operator cannot be mapped to any other feature and is instead rewritten as

```
a ?? b → (a == undefined || a == null) ? b : a
```

5.2. Composition mechanisms evaluation framework

We can now answer RQ_2.

How can a composition operator that performs linguistic reuse in Neverlang be evaluated with respect to its reuse capabilities?

To be considered viable with respect to its reuse capabilities, the composition operator should bring an improvement over the clone-and-own approach, which we consider as a baseline, since it is applicable without any explicit language construct. We can measure the improvement in terms of software metrics measuring the development effort and the design quality of the final product. For example, a good composition construct should reduce the LoC and increase the MI on average. Please note that this evaluation is limited to Neverlang and may not be applicable to other language workbenches. For instance, linguistic components developed by means of different language workbenches may need to be evaluated against different quality metrics.

5.3. Threats to validity

Internal validity. There might be computational bias with regards to our measurements: we developed both delegation features and desugaring features. To mitigate this issue, we explicitly used the original implementation of ECMAScript 3 (Cazzola and Olivares, 2016) that was developed by a different group.[8] This choice avoids an artificial increase in the number of reuse opportunities. Moreover, all clone-and-own features are generated. We substitute the call to the semantic action of the delegation feature with its source code. This choice keeps both implementations consistent with each other. Real-world implementations of the same features from scratch might provide better optimizations.

[8] Only one of the authors is co-author of both.

However, it should be considered that doing so would negate any opportunity of reuse during the language evolution, because even clone-and-own is excluded.

The language features selection process was not automated nor complete. There may be other ECMAScript features that met our eligibility requirements. To mitigate this issue, we selected several ECMAScript features from the changelog of the language specification since 2015. As a consequence, we can be fairly certain that our conclusions are valid.

External validity. Our research might not be generalizable to other language evolution scenarios and other language workbenches. To mitigate this aspect we implemented evolution through mechanisms that should be shared among most language workbenches: desugaring is a known technique in the literature and delegation can be similarly achieved with additional glue code in any language workbench that supports attribute grammars. Desugaring, in particular, is supported out-of-the-box by most language workbenches—despite with varying degrees of flexibility. Our evaluation uses metrics that are specific to Neverlang, such as #Roles and #SA. This was mitigated by measuring metrics from the literature, such as CC and MI and evaluating their ideal values based on empirical research made by third parties.

6. Related work

Most of our work is framed and focuses on the Neverlang language workbench. At the same time, several other language workbenches could benefit from exposing the reuse mechanism of all levels of granularity—discussed in Section 4. Notice that, to the best of our knowledge, popular language workbenches do expose reuse mechanisms, but rarely do so at all levels of granularity. For instance, MPS (Völter and Pech, 2012) lacks multiple inheritances between concepts. This prevents reuse at the action composition level of granularity. Contrary to MPS, CBS (Mosses, 2019b) has fine-grained reuse mechanisms but lacks coarse ones. JastAdd (Ekman and Hedin, 2007) allows to define syntactic and semantic assets separately; these elements are then composed using an aspect-oriented approach, but since aspects must directly refer to the syntax they apply to, it may be difficult to reuse them in a different scenario. LISA (Mernik et al., 2002; Henriques et al., 2005) can realize all types of language composition defined by Erdweg, but to the best of our knowledge the finest supported granularity is that of language feature. In Silver (Van Wyk et al., 2010) concrete syntax of each production must be repeated in each attached semantic aspect. This choice creates a dependency between syntax and semantics, hindering reusability. Truffle (Wimmer and Würthinger, 2012) does allow semantic reuse. However, the lack of capabilities for modeling syntax prevents reuse in the syntax domain. We found Rascal (Klint et al., 2009a) and Spoofax (Kats et al., 2010) to be the most advanced language workbenches in terms of reuse mechanisms. However, both bind AST nodes to static ADTs. While this is an advantage in terms of static analysis, it hinders the opportunities for reuse. MontiCore (Krahn et al., 2010) supports language extension but the semantics are directly hooked to the grammar thus limiting the opportunities for reuse when the evolution requires changes to the syntax.

Some other techniques for language definition exploit previously developed techniques in the domain of software engineering. Several techniques leverage LPLs (Krahn et al., 2010). Both Degueule et al. (2015) and Cazzola and Vacchi (2016) introduced other forms of language feature compositions. Several other works (Freeman and Pryce, 2006; van Amstel et al., 2010; van Deursen and Klint, 1998) discuss experiences on DSL development. Both Object Algebras (Oliveira and Cook, 2012) and Revisitor (Leduc et al., 2017) are language implementation patterns that focus on language extensibility. Both patterns are implemented by using mainstream object-oriented languages. While this choice broadens the applicability of Object Algebras and Revisitor, they cannot benefit from the ecosystem offered by language workbenches—which includes a compiler, interpreter, debugger, and even an IDE. Mernik (2013) show how inheritance can be exploited to implement language extension, language unification, and self-extension. MontiCore uses strongly-kind typed symbol table to ensure consistency during DSML composition (Butting et al., 2022).

Feature models represent an important formalism to express highly variable software systems. Among tools for the definition of feature models, FDL (van Deursen and Klint, 2002) is a DSL to represents feature diagrams. SXML (Mendonca et al., 2009) uses XML to represent feature models. Classen et al. (2011) introduced TVL, which extends the feature model with attributes. μTVL (Clarke et al., 2010) is a subset of TVL that can express feature models with multiple roots to introduce orthogonal expressivity. Similarly, Velvet (Rosenmüller et al., 2011), is a DSL for multi-dimensional variability modeling and the definition of feature interfaces.

However, to the best of our knowledge, there is no contribution to the study of techniques to foster opportunistic reuse to the evolution of programming languages and their ecosystem.

7. Conclusion

Language workbenches deal with all aspects of language development, from the syntax definition to the specification of a development environment. Among the most important aspects of language workbenches, their reuse capabilities are too often overlooked or lack the proper focus. We argue that this is due to the lack of a taxonomy dedicated to different reuse granularities. In this work, we extended the taxonomy by Erdweg et al. (2012) and introduced several mechanisms to perform composition between language components—each mechanism is specific to a granularity level. We argue that language workbenches should render these reuse mechanisms available at all levels through dedicated language constructs to foster opportunistic reuse. We took Neverlang as a proof of concept case study. Neverlang already implemented constructs for coarse-grained reuse, but lacked fine-grained ones. We extended Neverlang with two new language constructs: desugaring as an instance of action composition and delegation as an instance of action extension. The alternative to these mechanisms is the clone-and-own approach which literature proved to be undesirable because it leads to increased code duplication and decreased maintainability (Rubin et al., 2013). Moreover, clone-and-own is never applicable when the original source code is no longer available. We show that both reuse mechanisms can be used interchangeably to avoid code duplication in many instances based on real-world examples and that they ultimately result in better code. However, one reuse mechanism may be preferable over the other because performing reuse at a different level of granularity. Note that in this work we only evaluated instances of action composition and action extension due to them being a novel introduction in Neverlang; different implementations may lead to better designed code and/or larger wins in productivity. However, we conclude that both mechanisms are necessary or at least desirable to foster different opportunistic reuse scenarios in evolving programming languages.

CRediT authorship contribution statement

Francesco Bertolotti: Conceptualization, Methodology, Software, Investigation, Writing – original draft, Writing – review & editing. **Walter Cazzola:** Conceptualization, Methodology, Writing – original draft, Writing – review & editing, Supervision, Project administration, Funding acquisition. **Luca Favalli:** Conceptualization, Methodology, Software, Investigation, Writing – original draft, Writing – review & editing.

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

Data will be made available on request.

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.

van Amstel, M., van den Brand, M., Engelen, L., 2010. An exercise in iterative domain-specific language design. In: Capiluppi, A., Cleve, A., Moha, N. (Eds.), Proceedings of the Workshop on Software Evolution. IWPSE-EVOL'10, ACM, Antwerp, Belgium, pp. 48–57.

Antkiewicz, M., Ji, W., Berger, T., Czarnecki, K., Schmorleiz, T., Lämmel, R., Stănciulescu, S., Wąsowski, A., Schaefer, I., 2014. Flexible product line engineering with a virtual platform. In: Jalote, P., Briand, L., van der Hoek, A. (Eds.), Companion Proceedings of the 36th International Conference on Software Engineering. ICSE'14 Companio, ACM, Hyderabad, India, pp. 532–535.

Basten, B., van den Bos, J., Hills, M., Klint, P., Lankamp, A., Lisser, B., van der Ploeg, A., van der Storm, T., Vinju, J., 2015. Modular language implementation in rascal—Experience report. Sci. Comput. Program. 114, 7–19.

ter Beek, M.H., Schmid, K., Eichelberger, H., 2019. Textual variability modeling languages: An overview and considerations. In: Thüm, T., Duchien, L. (Eds.), Proceedings of the 23rd International Systems and Software Product Line Conference. SPLC'19, ACM, Paris, France, pp. 151–157.

van Binsbergen, L.T., Mosses, P.D., Sculthorpe, N., 2019. Executable component-based semantics. J. Log. Algebraic Methods Program. 103 (2), 184–212.

van Binsbergen, L.T., Sculthorpe, N., Mosses, P.D., 2016. Tool support for component-based semantics. In: Companion Proceedings of the 15th International Conference on Modularity. Companion Modularity'16, ACM, Málaga, Spain, pp. 8–11.

Butting, A., Michael, J., Rumpe, B., 2022. Language composition via kind-typed symbol tables. J. Object Technol. 21 (4), 4:1–13.

Campbell, G.A., 2018. Cognitive complexity: An overview and evaluation. In: Buschmann, F., Kruchten, P. (Eds.), Proceedings of the International Conference on Technical Debt. TechDebt'18, ACM, Gothenburg, Sweden, pp. 57–58.

Cazzola, W., 2012. Domain-specific languages in few steps: The Neverlang approach. In: Gschwind, T., De Paoli, F., Gruhn, V., Book, M. (Eds.), Proceedings of the 11th International Conference on Software Composition. SC'12, 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., 2016. Modularity and optimization in synergy. In: Batory, D. (Ed.), Proceedings of the 15th International Conference on Modularity. Modularity'16, ACM, Málaga, Spain, pp. 70–81.

Cazzola, W., Shaqiri, A., 2017. 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., Vacchi, E., 2013. Neverlang 2: Componentised language development for the JVM. In: Binder, W., Bodden, E., Löwe, W. (Eds.), Proceedings of the 12th International Conference on Software Composition. SC'13, Lecture Notes in Computer Science, Springer, Budapest, Hungary, pp. 17–32.

Cazzola, W., Vacchi, E., 2016. Language components for modular DSLs using traits. Comput. Lang. Syst. Struct. 45, 16–34. http://dx.doi.org/10.1016/j.cl.2015.12.001.

Chapin, N., Hale, J.E., Khan, K.M., Ramil, J.F., Wui-Gee, T., 2001. Types of software evolution and software maintenance. J. Softw. Maint. Evol.: Res. Pract. 13 (1), 3–30.

Chowdhary, K.R., 2013. On the evolution of programming languages. In: Proceedings of the UGC National Conference on New Advances in Programming Languages and their Implementation. APL'13, Jodhpur, India, available as arXiv:2007.02699.

Churchill, M., Mosses, P.D., Schulthorpe, N., Torrini, P., 2015. Reusable components of semantic specifications. In: Transaction on Aspect-Oriented Software Development. pp. 132–179.

Churchill, M., Mosses, P.D., Torrini, P., 2014. Reusable components of semantic specifications. In: Ernst, E. (Ed.), Proceedings of the 13th International Conference on Modularity. Modularity'14, ACM, Lugano, Switzerland, pp. 145–156.

Clarke, D., Muschevici, R., Proença, J., Schaefer, I., Schlatte, R., 2010. Variability modelling in the ABS language. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (Eds.), Proceedings of the 9th International Symposium on Formal Methods for Components and Objects. FMCO'10, In: Lecture Notes in Computer Science, vol. 6957, Springer, Graz, Austria, pp. 204–224.

Classen, A., Boucher, Q., Heymans, P., 2011. A text-based approach to feature modelling: Syntax and semantics of TVL. Sci. Comput. Program. 76 (12), 1130–1143.

Combemale, B., 2015. Towards Language-Oriented Modeling (Habilitation À Diriger Des Recherches). Université de Rennes 1, Rennes, France.

Combemale, B., Kienzle, J., Mussbacher, G., Barais, O., Bousse, E., Cazzola, W., Collet, P., Degueule, T., Heinrich, R., Jézéquel, J.-M., Leduc, M., Mayerhofer, T., Mosser, S., Schöttle, M., Strittmatter, M., Wortmann, A., 2018. Concern-Oriented Language Development (COLD): Fostering reuse in language engineering. Comput. Lang. Syst. Struct. 54, 139–155. http://dx.doi.org/10.1016/j.cl.2018.05.004.

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

Degueule, T., 2016. Interoperability and composition of DSLs with Melange. In: ACM Student Research Competition Grand Finals.

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

van Deursen, A., Klint, P., 1998. Little languages: Little maintenance? J. Softw. Maint.: Res. Pract. 10 (2), 75–92.

van Deursen, A., Klint, P., 2002. Domain-specific language design requires feature descriptions. J. Comput. Inf. Technol. 10, 1–17.

Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., Czarnecki, K., 2017. An exploratory study of cloning in industrial software product lines. In: Cohen, M., Acher, M. (Eds.), Proceedings of the 21st International Systems and Software Product Line Conference. SPLC'17, ACM, Sevilla, Spain, pp. 25–34.

Ekman, T., Hedin, G., 2007. The JastAdd extensible Java compiler. In: Proceedings of the 22nd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPSLA'07, ACM, Montréal, Québec, Canada, pp. 1–18.

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., Rendel, T., Kästner, C., Ostermann, K., 2011. SugarJ: Library-based syntactic language extensibility. In: Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming. OOPSLA'11, ACM, Portland, Oregon, USA, pp. 391–406.

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.

Freeman, S., Pryce, N., 2006. Evolving an embedded domain-specific language in Java. In: Proceedings of the Dynamic Languages Symposium. DLS'06, ACM, Portland, Oregon, USA, pp. 855–865.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. In: Professional Computing Series, Addison-Wesley, Reading, Ma, USA.

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

Grönninger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S., 2008. Monticore: A framework for the development of textual domain specific languages. In: Schäfer, W., Dwyer, M., Gruhn, V. (Eds.), Companion Proceedings of the 30th International Conference on Software Enginering. Companion ICSE'08, IEEE, Leipzig, Germany, pp. 925–926.

Hemer, D., Lindsay, P., 2001. Specification-based retrieval strategies for module reuse. In: Grant, D.D., Sterling, L. (Eds.), Proceedings of the 5th Australian Software Engineering Conference. ASEC'01, IEEE, Canberra, Australia, pp. 235–243.

Henriques, P.R., Varanda Pereira, M.J., Mernik, M., Lenič, M., Gray, J., Wu, H., 2005. Automatic generation of language-based tools using the LISA system. In: IEE Proceedings — Software, Vol. 152, No. 2. pp. 54–69.

Hinterreiter, D., Linsbauer, L., Reisinger, F., Prähofer, H., Grünbacher, P., Egyed, A., 2018. Feature-oriented evolution of automation software systems in industrial software ecosystems. In: Mahulea, C., Seatzu, C. (Eds.), Proceedings of the 23rd International Conference on Emerging Technologies and Factory Automation. ETFA'18, IEEE, Torino, Italy, pp. 107–114.

ISO/IEC/IEEE International Standard, 2017. Systems and Software Engineering—Vocabulary. Standard 24765-2017, http://dx.doi.org/10.1109/IEEESTD.2017.8016712.

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.

Karaila, M., 2009. Evolution of a domain specific language and its engineering environment—Lehman's laws revisited. In: Tolvanen, J.-P., Gray, J., Rossi, M., Sprinkle, J. (Eds.), Proceedings of Te 9th OOPSLA Workshop on Domain-Specific Modeling. DSM'09, Orlando, FL, USA, pp. 1–7.

Kats, L.C.L., Visser, E., Wachsmuth, G., 2010. Pure and declarative syntax definition: Paradise lost and regained. In: Proceedings of ACM Conference on New Ideas in Programming and Reflections on Software. Onward! 2010, ACM, Reno-Tahoe, Nevada, USA.

Klint, P., van der Storm, T., Vinju, J., 2009a. EASY meta-programming with rascal. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (Eds.), Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering III. GTTSE'09, In: Lecture Notes in Computer Science, vol. 6491, Springer, Braga, Portugal, pp. 222–289.

Klint, P., van der Storm, T., Vinju, J., 2009b. 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.

Klint, P., van der Storm, T., Vinju, J., 2019. Rascal, 10 years later. In: Proceedings of the 19th International Working Conference on Source Code Analysis and Manipulation. SCAM'19, IEEE, Cleveland, OH, USA, p. 139.

Kloibhofer, S., Pointhuber, T., Heisinger, M., Mössenböck, H., Stadler, L., Leopoldseder, D., 2020. SymJEx: Symbolic execution on te GraalVM. In: Marr, S. (Ed.), Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes. MPLR'20, ACM, pp. 63–72.

Kosar, T., Bohra, S., Mernik, M., 2016. Domain specific languages: A systematic mapping study. Inf. Softw. Technol. 71, 77–91.

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

Krishnamurthi, S., 2015. Desugaring in practice: Opportunities and challenges. In: Asai, K., Sagonas, K. (Eds.), Proceedings of the Workshop on Partial Evaluation and Program Manipulation. PEPM'15, ACM, Mumbai, India, pp. 1–2.

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. ACM, SPLC'16, 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.

Latifi, F., Leopoldseder, D., Wimmer, C., Mössenböck, H., 2021. CompGen: Generation of fast JIT compilers in a multi-language VM. In: Guha, A. (Ed.), Proceedings of the 17th International Symposium on Dynamic Languages. DLS'21, ACM, Chicago, IL, USA, pp. 35–47.

Lavazza, L., Abualkishik, A.Z., Liu, G., Morasca, S., 2023. An empirical evaluation of the cognitive complexity measure as a predictor of code understandability. J. Syst. Softw. 197.

Leduc, M., Degueule, T., Combemale, B., van der Storm, T., 2017. Revisiting visitors for modular extension of executable DSMLs. In: Gray, J. (Ed.), Proceedings of 20th International on Model Driven Engineering Languages and Systems. MoDELS'17, IEEE, Austin, TX, USA, pp. 112–122.

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

Lehman, M.M., Kahen, G., Fernández-Ramil, J.C., 2002. Behavioural modelling of long-lived evolution processes - Some issues and an example. J. Softw. Maint. Evol. 14 (5), 335–351.

Lindeman, R.T., Kats, L.C.L., Visser, E., 2011. Declaratively defining domain-specific language debuggers. In: Schultz, U. Pagh (Ed.), Proceedings of the 10t International Conference on Generative Programming and Components. GPCE'11, ACM, Portland, OR, USA, pp. 127–136.

Long, J., 2001. Software reuse antipatterns. ACM SIGSOFT Softw. Eng. Notes 26 (4), 68–76.

Lorenzen, F., Erdweg, S., 2016. Sound type-dependent syntactic language extension. In: Majumdar, R. (Ed.), Proceedings of the 43rd Annual Symposium on Principles of Programming Languages. PoPL'16, ACM, St. Petersburg, FL, USA, pp. 204–216.

Maga, C.R., Jazdi, N., Göhner, P., 2011. Reusable models in industrial automation: Experiences in defining appropriate levels of granularity. In: Bittanti, S., Colaneri, P. (Eds.), Proceedings of the 18th IFAC World Congress, Vol. 44. IFAC'11, Elsevier, Milan, Italy, pp. 9145–9150.

Mäkitalo, N., Taivalsaari, A., Kiviluoto, A., Mikkonen, T., Capilla, R., 2020. On opportunistic software reuse. Computing 102 (11), 2385–2408.

Méndez-Acuña, D., Galindo, J.A., Degueule, T., Combemale, B., Baudry, B., 2016. Leveraging software product lines engineering in the development of external DSLs: A systematic literature review. Comput. Lang. Syst. Struct. 46, 206–235.

Mendonca, M., Branco, M., Cowan, D., 2009. S.P.L.O.T.—Software Product Lines Online Tools. In: Companion Proceedings of the 24th Conference on Object-Oriented Programming Systems Languages and Applications. Companion OOPSLA'09, ACM, Orlando, FL, USA, pp. 761–762.

Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M., 2005. Challenges in software evolution. In: Proceedings of the Eighth International Workshop on Principles of Software Evolution. IWPSE'05, IEEE Press, Lisbon, Portugal, pp. 13–22.

Mernik, M., 2013. An object-oriented approach to language compositions for software language engineering. J. Syst. Softw. 86 (9), 2451–2464.

Mernik, M., Heering, J., Sloane, A.M., 2005. When and how to develop domain specific languages. ACM Comput. Surv. 37 (4), 316–344.

Mernik, M., Lenič, M., Avdičaušević, E., Žumer, V., 2002. LISA: An interactive environment for programming language development. In: Horspool, N.R. (Ed.), Proceedings of the 11th International Conference on Compiler Construction. CC'02, In: Lecture Notes in Computer Science, vol. 2304, Springer, Grenoble, France, pp. 1–4.

Mosses, P.D., 2019a. A component-based formal language workbench. In: Monahan, R., Prevosto, V., Proença, J. (Eds.), Proceedings of the 5th Workshop on Formal Integrated Development Environment. F-IDE'19, Porto, Portugal, pp. 29–34.

Mosses, P.D., 2019b. Software meta-language engineering and CBS. J. Comput. Lang. 50, 39–48.

Niephaus, F., Felgentreff, T., Hirschfeld, R., 2019. GraalSqueak: Toward a smalltalk-based tooling platform for polyglot programming. In: Finocchi, I. (Ed.), Proceedings of the 16th International Conference on Managed Programming Languages and Runtimes. MPLR'19, ACM, Athens, Greece, pp. 14–26.

Oliveira, B.C.d.S., Cook, W.R., 2012. Extensibility for the masses: Practical extensibility with object algebras. In: Noble, J. (Ed.), Proceedings of the 26th European Conference on Object-Oriented Programming. ECOOP'12, In: Lecture Notes in Computer Science, vol. 7313, Springer, Beijing, China, pp. 2–27.

Pech, V., Shatalin, A., Völter, M., 2013. JetBrains MPS as a tool for extending java. In: Binder, W. (Ed.), Proceedings of the 10th International Conference on Principles and Practices of Programming on the Java Platform. PPPJ'13, ACM, Stuttgart, Germany, pp. 165–168.

Rosenmüller, M., Siegmund, N., Thüm, Saake, G., 2011. Multi-dimensional variability modeling. In: Czarnecki, K., Eisenecker, U.W. (Eds.), Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems. VaMoS'11, ACM, Namur, Belgium, pp. 11–20.
Rubin, J., Czarnecki, K., Chechik, M., 2013. Managing cloned variants: A framework and experience. In: Jarzabek, S., Gnesi, S. (Eds.), Proceedings of the 17th International Software Product Line Conference. SPLC'13, ACM, Tokyo, Japan, pp. 101–110.
Rumpe, B., Hölldobler, K., Kautz, O., 2021. MontiCore: Language Workbench and Library, Handbook. Aachen, Germany.
Sen, A., 1997. The role of opportunism in the software design reuse process. IEEE Trans. Softw. Eng. 23 (7), 418–436.
Serebrenik, A., Mens, T., 2015. Challenges in software ecosystems research. In: Crnkovic, I. (Ed.), Proceedings of the 2015 European Conference on Software Architecture Workshops. ECSAW'15, ACM, Dubrovnik, Croatia, pp. 1–6.
Steinberg, D., Budinsky, D., Paternostro, M., Merks, E., 2008. EMF: Eclipse Modeling Framework. Addison-Wesley.
Thanhofer-Pilisch, J., Lang, A., Vierhauser, M., Rabiser, R., 2017. A systematic mapping study on DSL evolution. In: Felderer, M., Olsson, H. Holmström, Skavhaug, A. (Eds.), Proceedings of the 43rd Euromicro Conference on Software Engineering and Advanced Applications. SEAA'17, IEEE, Vienna, Austria, pp. 149–156.
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.
Tracz, W., 1990. Where does reuse start? ACM SIGSOFT Softw. Eng. Notes 15 (2), 42–46.
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., Van Wyk, E. (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 Wyk, E., Bodin, D., Gao, J., Krishnan, L., 2010. Silver: an extensible attribute grammar system. Sci. Comput. Program. 75 (1–2), 39–54.
Voelter, M., 2013. DSL engineering.
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.
Völter, M., Siegmund, J., Berger, T., Kolb, B., 2014. Towards user-friendly projectional editors. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (Eds.), Proceedings of the 7th International Conference on Software Language Engineering. SLE'14, In: Lecture Notes in Computer Science, vol. 8706, Springer, Västerås, Sweden, pp. 41–61.
Wachsmuth, G.H., Konat, G.D.P., Visser, E., 2014. Language design with the Spoofax language workbench. IEEE Softw. 31 (5), 35–43.
Wimmer, C., Würthinger, T., 2012. Truffle: A self-optimizing runtime system. In: Leavens, G.T. (Ed.), Proceedings of the 3rd Annual Conference on Systems, Programming and Applications: Software for Humanity. SPLASH'12, ACM, Tucson, AZ, USA, pp. 1–2.
Würthinger, T., 2014. Graal and truffle: Modularity and separation of concerns as cornerstones for building a multipurpose runtime. In: Companion Proceedings of the 13th International Conference on Modularity. Companion Modularity'14, ACM, Lugano, Switzerland, pp. 3–4.
Würthinger, T., Wimmer, C., Woß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M., 2013. One VM to rule them all. In: Hirschfeld, R. (Ed.), Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. Onward!'13, ACM, Indianapolis, IN, USA, pp. 187–204.
Würthinger, T., Wöß, A., Stadler, L., Duboscq, G., Simon, D., Wimmer, C., 2012. Self-optimizing AST interpreters. In: Warth, A. (Ed.), Proceedings of the 8th Symposium on Dynamic Languages. DSL'12, ACM, Tucson, AZ, USA, pp. 73–82.
Xu, Y., Ramanathan, J., Ramnath, R., Singh, N., Deshpande, S., 2011. Reuse by placement: A paradigm for cross-domain software reuse with high level of granularity. In: Schmid, K. (Ed.), Proceedings of the 12th International Conference on Software Reuse. ICSR'11, In: Lecture Notes in Computer Science, vol. 6727, Springer, Pohang, South Korea, pp. 69–77.

Francesco Bertolotti is currently a Computer Science Ph.D. student at Università degli Studi di Milano and a member of the ADAPT Laboratory. Previously, he was an assistant researcher at the same University where he also got his master degree in Computer Science. His research interests are programming languages, software quality, machine/deep learning techniques and their reciprocal cross-fertilization. He can be contacted at francesco.bertolotti@unimi.it for any question.

Walter Cazzola is currently an Associate Professor in the Department of Computer Science 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 http://cazzola.di.unimi.it and he can be contacted at cazzola@di.unimi.it for any question.

Luca Favalli is currently a Computer Science Postdoctoral Researcher at Università degli Studi di Milano. He got his Ph.D. in computer science from the 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.