Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering

Benoit Combemale[e], Jörg Kienzle[h], Gunter Mussbacher[h], Olivier Barais[a], Erwan Bousse[b], Walter Cazzola[c], Philippe Collet[d], Thomas Degueule[f], Robert Heinrich[g], Jean-Marc Jézéquel[a], Manuel Leduc[a], Tanja Mayerhofer[b], Sébastien Mosser[d], Matthias Schöttle[h], Misha Strittmatter[g], Andreas Wortmann[i]

[a] *Univ Rennes, Inria, CNRS, IRISA, France*
[b] *TU Wien, Austria*
[c] *Università degli Studi di Milano, Italy*
[d] *Université Nice Côte d'Azur, CNRS, I3S, France*
[e] *University of Toulouse, France*
[f] *Centrum Wiskunde & Informatica, Netherlands*
[g] *Karlsruhe Institute of Technology, Germany*
[h] *McGill University, Canada*
[i] *RWTH Aachen University, Germany*

Abstract

Domain-Specific Languages (DSLs) bridge the gap between the problem space, in which stakeholders work, and the solution space, i.e., the concrete artifacts defining the target system. They are usually small and intuitive languages whose concepts and expressiveness fit a particular domain. DSLs recently found their application in an increasingly broad range of domains, e.g., cyber-physical systems, computational sciences and high performance computing. Despite recent advances, the development of DSLs is error-prone and requires substantial engineering efforts. Techniques to reuse from one DSL to another and to support customization to meet new requirements are thus particularly welcomed. Over the last decade, the Software Language Engineering (SLE) community has proposed various reuse techniques. However, all these techniques remain disparate and complicate the development of real-world DSLs involving different reuse scenarios.

In this paper, we introduce the Concern-Oriented Language Development (COLD) approach, a new language development model that promotes modularity and reusability of *language concerns*. A language concern is a reusable piece of language that consists of usual language artifacts (e.g., abstract syntax, concrete syntax, semantics) and exhibits three specific interfaces that support (1) variability management, (2) customization to a specific context, and (3) proper usage of the reused artifact. The approach is supported by a conceptual model which introduces the required concepts to implement COLD. We also present concrete examples of some language concerns and the current state of their realization with metamodel-based and grammar-based language workbenches. We expect this work to provide insights into how to foster reuse in language specification and implementation, and how to support it in language workbenches.

Keywords: domain-specific languages, language concern, language reuse

[CKM+18] B. Combemale, J. Kienzle, G. Mussbacher, O. Barais, E. Bousse, W. Cazzola, P. Collet, T. Degueule, R. Heinrich, J. Jézéquel, M. Leduc, T. Mayerhofer, S. Mosser, M. Schöttle, M. Strittmatter, A. Wortmann: Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. In: Computer Languages, Systems & Structures, 54, pg. 139-155, 2018.
www.se-rwth.de/publications

1. Introduction

Domain-Specific (Modeling) Languages (DSLs) aim at bridging the gap between the problem and solution spaces. The stakeholders of a DSL work in the problem space, while the concrete software artifacts defining the target system forms the solution space. DSLs are most often small and intuitive software languages whose concepts and expressiveness focus on a particular domain. Since "software languages are software too" [1], the development of DSLs includes their supporting environment (e.g., editors, generators, simulators, etc.) and thus inherits the complexity of software development in general.

Despite recent advances, the development of DSLs is prone to errors and requires substantial engineering efforts. Moreover, using today's technologies, the same process is typically repeated from scratch for every new DSL or new version of a DSL. To foster their adoption in industry, the benefits in terms of productivity when using DSL technologies must offset the initial investment required in developing such DSLs. Therefore, tools and methods must be provided to assist language designers in the development of new DSLs and the evolution of legacy ones to mitigate development costs. Techniques for increasing reuse from one DSL to another and supporting the customization of legacy languages to meet new requirements are thus particularly welcomed.

Over the last decade, the Software Language Engineering (SLE) community has proposed various reuse techniques, mostly leveraging previous experiences in software reuse (e.g., aspects [2], polymorphic [3] and parametric [4] reuse, composition operators [5, 6], language product lines [7, 8, 9]). Although these techniques address a wide range of scenarios, the development of realistic languages is still difficult as it requires to combine them.

In this paper, we leverage previous efforts to integrate different software reuse techniques [10] to lift them up to the meta-level for language reuse. We introduce the Concern-Oriented Language Development (COLD) approach, a new language development model that promotes modularity and reusability of language concerns. A language concern is a reusable piece of language that is made of a set of usual language constituents: some abstract syntax, some concrete syntax, and some semantics definitions. In addition, a language concern exhibits three specific interfaces that support (1) variability management, (2) customization to a specific context, and (3) proper usage of the reused artifact.

In the remainder, Section 2 introduces background notions on language reuse and the existing conceptual model at the software level to foster modularity and reuse, as well as a common case study. Section 3 introduces the underlying conceptual model of COLD and presents the roles and the respective scenarios considered within COLD. Section 4 provides details about the *concernification* of languages, i.e., making reusable languages through the three interfaces promoted by concern-oriented development. We notably describe these three interfaces for a language concern and the underlying life cycle to use them for supporting various kinds of language reuse. To demonstrate the possible applications of COLD, we illustrate in Section 5 the current support across different technological spaces (e.g., grammarware and modelware) on the basis of the common case study. Finally, Sections 6 and 7 conclude the paper and draw some perspectives.

2. Background and Motivating Example

In this section we introduce the background material require for the rest of the paper, and we illustrate the motivation to provide a unifying paradigm for language reuse. In particular, we introduce the existing paradigm of concern-oriented reuse, demonstrate its usefulness at the language level to unify various disparate literature on language reuse, and motivate on the basis of an illustrative example used throughout the paper.

2.1. Concern-Oriented Reuse

Concern-Oriented Reuse (CORE) [10] is a new reuse paradigm for general-purpose software development that combines best practices from Model-Driven Engineering (MDE), Component-Based Software Engineering (CBSE), Software Product Lines (SPL), advanced Separation of Concerns (SoC) (including feature-oriented and aspect-oriented software development), and goal modeling.

In CORE, software development is structured around modules called *concerns* that provide a variety of reusable solutions (i.e., models and code) for recurring software development issues in a versatile, generic way. Concerns decompose software into reusable units according to some points of interest and may have varying scopes, e.g., encapsulating several authentication choices, communication protocols, or design patterns. The models within a concern may span multiple phases of software development and levels of abstraction (from requirements, analysis, architecture, and design models to implementation). The main premise of CORE is that recurring development concerns are made available in a concern library, which eventually should cover most recurring software development needs. Similar to class libraries in modern programming languages, this library should grow as new development concerns emerge, and existing concerns should continuously evolve as alternative architectural, algorithmic, and technological solutions become available. Applications are built by reusing existing concerns from the library whenever possible, following a well-defined reuse process supported by clear interfaces. To generate an executable in which concerns exhibit intricate crosscutting structure and behavior, CORE relies on additive software composition techniques, feature-oriented technology and aspect-oriented technology.

2.1.1. Concern Interfaces and Concern Reuse

A concern provides a well-defined, three-part interface [11]. The *Variation Interface* (VI) of a concern is composed of a feature model [12] that expresses the closed variability [1] of solutions and techniques encapsulated within the concern by its designer, similar to what is done in software product lines for a specific application domain. Additionally, the VI specifies the impacts of selecting a feature on non-functional goals and qualities with an impact model that is expressed using a variant of the Goal-oriented Requirement Language [14, 15]. This allows a concern user to perform trade-off analysis between offered features, e.g., for design-time exploration.

The *Customization Interface* (CI) of a concern designates the generic, partially defined structural/behavioral elements that enable an open form of variability, and how these

[1] Variability can be considered open when at least one variation point can still receive new variants, and closed instead [13].

elements have to be connected to application-specific elements when the concern is reused. For example, an *Authentication* concern might define a generic *Authenticatable* entity that needs to be mapped to one or several application-specific entities, e.g., *Administrator*, *ProjectManager*, and *Developer*.

The *Usage Interface* (UI), similar to classic APIs, designates the elements of the concern that can be accessed by the context in which the concern is reused. Any element that has its visibility set to *public* is part of the usage interface and can therefore be used. Other elements remain encapsulated within the concern.

Then, reusing a concern is based on three steps guided by the three-part interface:

1. choosing the desired solution among the available alternatives encapsulated within the concern from the VI,
2. adapting the realization of the chosen solution (generated based on the selection made in step one) to the specific reuse context using the CI, and,
3. using the chosen, adapted realization for the targeted purpose via the UI.

2.1.2. Component Interfaces vs. Concern Interfaces

While concerns and their interfaces (VI, CI and UI) show many similarities with components and their *required* (RI) and *provided* (PI) interfaces as defined by CBSE, one of the fundamental differences is that the unit of reuse in CORE—the concern—is broader than the unit of reuse in CBSE—the component. Similar to SPLs, a CORE concern groups and encapsulates a variety of reusable solutions for a specific domain of interest[2], and expresses this variability explicitly in the VI. The VI expresses the closed variability, i.e., the set of solutions that the designers of the concern have realized. Standard CBSE does not provide a means to group a set of functionally equivalent components together. If a developer has access to a library of existing components, then it is possible to determine the set of components S in the library that have the same RI and PI. Each component encapsulates a specific solution, and S groups them together, declaring that from an RI and PI perspective they are equivalent. The VI in CORE does more than just grouping, though. By exposing impacts on non-functional properties and qualities, it allows the developer to perform trade-off analysis, which ultimately allows to choose a specific solution from within the concern.

The UI of a concern is equivalent to the PI of a component, as both kinds of interfaces designate the structural and behavioral properties/services that the units offer to the reuse context. However, the CI of a concern is different from the RI of a component in at least two situations.

Situation 1. In CBSE, when a more specific component A needs to use a service from a more generic component, A declares an RI which during assembly can be linked to, e.g., a component B that has a matching PI. As a result, A and B are only loosely coupled. In theory, any other component B' that has the same PI as B could be used instead of B. In CORE this situation is handled with the UI, not the CI. When a more specific concern X needs functionality provided by a generic concern Y, the feature in X that needs Y simply reuses the desired variant of Y (by making a selection in the VI) and then directly connects to the UI of that selected variant of Y. The reuse of Y is encapsulated within

[2]A practice already advocated almost 40 years ago in [16]

X in accordance with the information hiding principle, and as a result, the reuse of Y is not visible in the interfaces of X. Also, when X is reused, Y is always automatically reused, resulting in a stronger coupling.

Situation 2. In CBSE, when a developer wants to reuse a (generic) component B to augment a service S that is part of the PI of an existing component A, then B must declare the same provided interface as S. During assembly, any connections from components that require S offered by A should be assembled with the new S offered by B. Achieving the same effect in CORE is much simpler. B can declare a placeholder of the service S in its CI. The developer can then augment the service S offered by reusing a variant of B and mapping the service S from B's CI to a service in A.

2.2. On Language Reuse

DSLs are complex artifacts that require specialized development skills. Specifying and implementing their abstract syntax, concrete syntax, semantics, and supporting tools (e.g., analyzers, editors, transformations) is a complex endeavor that is prone to errors and requires substantial engineering efforts. Moreover, the same process is typically repeated from scratch for every new DSL or new version of a DSL. To foster their adoption, the benefits in terms of productivity when using DSL technologies must offset the initial investment required in developing such DSLs. Therefore, researchers have provided tools and techniques to assist language designers in the development of new DSLs and the evolution of legacy ones to mitigate development costs. DSLs are by their very nature meant to be tailored to a particular domain of application. Although this may suggest that there are few opportunities for reuse from one DSL to another, it is not uncommon to see different DSLs share recurring paradigms (e.g., state-transition, workflows, actions and queries, units, etc.) [17]. Over the years, researchers have proposed many approaches seeking to improve language reuse. We give a brief overview of them in the following. Many of these techniques ultimately materialize as features of a language workbench: LISA [18], Melange [5], MontiCore [19], Neverlang [6], Rascal [20], or Spoofax [21] to name a few.

Language Extension and Composition. Implementation techniques for DSLs are diverse, and so are the language composition techniques at hand. From attribute grammars [22] to Parsing Expression Grammars [23] and scannerless parsing [24], the literature is rich in extension and modularization techniques for grammars. The same can naturally be said of the metamodeling world, where the problem of composing language constituents often boils down to the problem of (meta-)model composition [25]. In both cases, solutions either require the definition of explicit language modules that have built-in modularity mechanisms (e.g., extension points) or propose composition operators that operate on existing languages to build new ones [5]. On the semantics side, formal approaches have been modularized (e.g., modular structural operational semantics [26], modular denotational semantics [27]) as well as their concrete realization in e.g., K [28], Redex [29], or DynSem [30]. DSLs are eventually implemented in a concrete programming language, for instance in the form of a set of classes materializing the language concepts and an associated interpreter implementing its execution semantics. Researchers have thus developed extensible and composable design patterns to implement such concrete artifacts modularly (e.g., [31, 32]). While most of these seek to compose languages that are

developed within the same technological space, or that run on the same virtual machine, other authors tackle the problem of composing heterogeneous languages [33]. Recent advances in projectional editing also pave the way for better language composition [34]. Overall, while many techniques have been developed, language composition approaches can rarely express the closed and open variability of language modules on their own.

Language Product Lines. A recent advance in SLE is the notion of language product lines [8, 9]. In analogy with software product lines [35], a language product line is a product line where the products are languages. Engineering a language product line requires expressing the variability of such a family of languages. Typically, features of the variability model correspond to language features. Language features are reusable pieces of languages that can be easily composed to derive a new specialized language variant from the family, according to a particular configuration. Language product lines typically make explicit the *closed* variability of a family of languages: the set of features of a language can be tailored, but it is hard to further adapt the resulting language to a new context of use.

To the best of our knowledge, there is no language reuse mechanism that adequately captures both the closed and open variability of language modules. The notion of language interfaces, similarly, has only recently been explored [36]. Overall, the vision of off-the-shelf language components that can be freely picked from a library, composed, and customized by language designers has yet to be fully realized.

2.3. Motivating Example

To illustrate the vision of COLD in the context of concern-oriented language reuse, we consider the following use case, which is referenced throughout the paper: A language engineer aims to implement a timed state machine language for modeling cyber-physical systems. The core concepts of this language are `State`, `Transition`, `Trigger`, and `Guard Expression`. However, to be able to define complex guard conditions for state transitions, an expression language needs to be integrated into the timed state machine language. Furthermore, to be able to represent sensed physical data (e.g., measured distances, speed, etc.), as well as time triggers, a type system supporting units is required. Developing the expression language and unit language from scratch for their integration into the targeted timed state machine language would impose significant additional engineering efforts, as they are non-trivial languages themselves. To mitigate increased initial investments for developing the timed state machine language, it would be beneficial to reuse an existing expression language as well as an existing unit language as illustrated in Fig. 1. From an existing expression language (e.g., Xbase [37]), concepts for representing literals, variables, arithmetic operations, and comparison operations could be reused. Comparison operations could then be used to define complex guard expressions. From an existing unit language, such as the one presented in [38], different kinds of units, such as time units (millisecond, second, minute, etc.) and length units (centimeter, meter, feet, etc.), as well as operations considering units in computations could be reused. This would allow the definition of the units of sensed values represented as literals assigned to variables. Furthermore, literals with a time unit could then be used to define time triggers. These reused languages are explained in more detail in the next paragraphs.

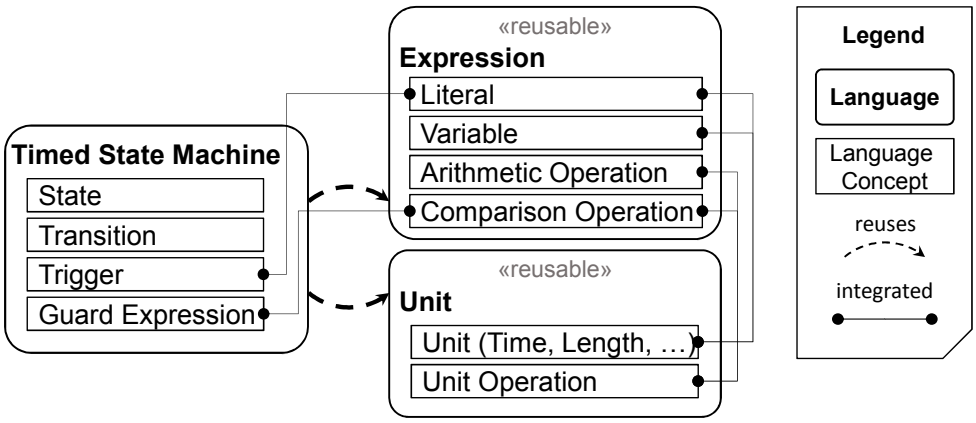

Figure 1: COLD use case of a state machine language reusing two different concerns

Expression Language. An expression language is needed in many DSLs. Therefore, it is a perfect candidate for reuse. An expression language provides the necessary elements to define calculations and comprises features for defining literals and operations. A language engineer can choose to reuse different types of literals (e.g., Integer literals, String literals, Boolean literals), as well as different types of operations, such as arithmetic operations (e.g., add, subtract, multiply) and comparison operations (e.g., smaller than, equals, greater than) that are provided for all supported literal types.

Unit Language. Most common programming languages (with the exception of Ada) do not have support for units within their type system. This lack of units attached to numeric values can lead to conversion issues and have serious consequences. One infamous example is the loss of NASA's Mars Climate Orbiter which was caused by English units and metric units being mixed up[3]. The unit language provides concepts for attaching units to numeric values ensuring type safety when assigning numerical values to variables. Along basic features for SI Units, this concern also provides necessary operations on these units. Possible support can also include the ability to provide automatic conversions, e.g., when performing an operation between two values with different units of the same dimensions (e.g., lengths measured in meter and feet).

Open Issues. While the principle of language reuse appears obvious at first glance, the actual implementation of concrete techniques raises many open issues. In particular, since software languages are initially defined in a given context, reusing an existing language in another language may require to precisely select what should be reused (e.g., String literals may not be required in the Timed State Machine), and to customize it in the context of the new language (e.g., renaming a particular construct to fit the new context). Eventually, the reused languages must be composed in the new one (e.g., integration links in Fig. 1), possibly employing complex composition mechanisms according to the artifact that belongs to the definition of the initial languages.

[3]See `https://mars.jpl.nasa.gov/msp98/news/mco991110.html`

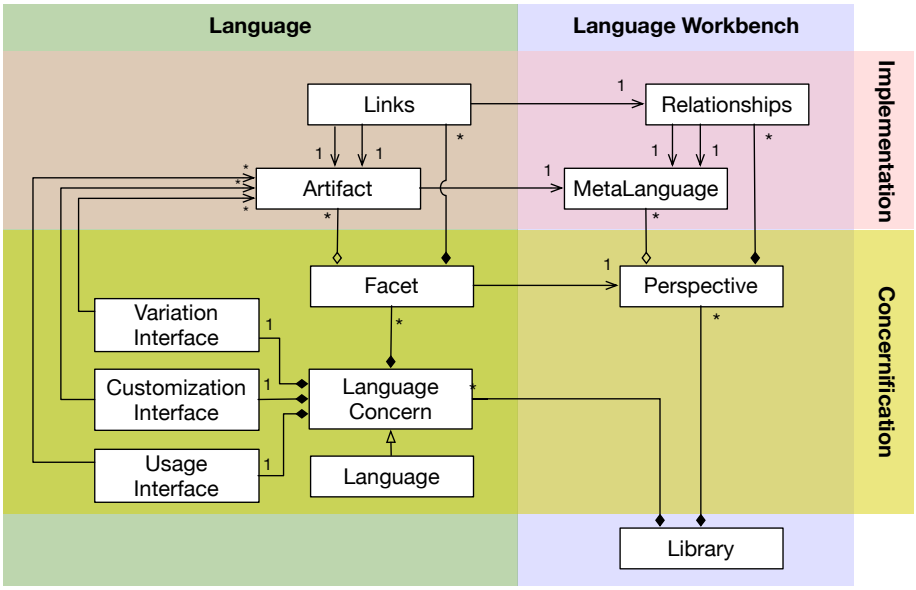

Figure 2: Conceptual model of COLD

Section 3 defines the vocabulary required to express the different concepts of COLD. The definitions are illustrated using the expression language reused for implementing the timed state machine language. This use case is also used in Section 4, which develops further the implications of the definitions of the COLD interfaces in the context of software language definition. Finally, Section 5 demonstrates how the use case can be realized with COLD in different language workbenches.

3. COLD: A Unifying Framework for Language Reuse

This section introduces COLD, the proposed unifying framework for language reuse, which bring together diverse existing reuse mechanisms in the common concept of language concern. We first give the definitions of the main concepts, and then discuss the different roles involved in the use of such a framework, and the expected scenarios of reuse to be supported.

3.1. Definitions

This subsection is dedicated to the definition of the vocabulary required to express the concepts introduced in COLD. Fig. 2 shows the conceptual model of COLD and the relationships between its concepts. An example instantiation of those concepts can be found in Fig. 3. Each term defined in the conceptual model of COLD is shown in *italic* font in this section.

Language Concern: A *Language Concern* is a configurable unit of reuse that encapsulates a specific user-visible set of constructs of a *language* (such as support for expressions,

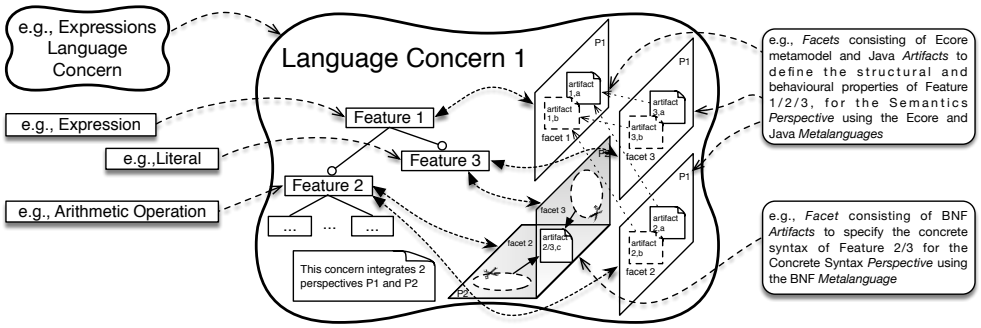

Figure 3: Illustration of a Language Concern

support for exceptions, units, uncertainty management, etc.). A Language Concern may reuse other Language Concerns.

A Language Concern encompasses the definition of the expected constituents among abstract syntax, concrete syntax and behavioral semantics. Those constituents are concretized in various *artifacts* (e.g., Ecore metamodels, BNF grammars, OCL constraints, ATL model transformations, Java visitors, etc.). *Artifacts* are organized in *facets* that correspond to the *perspectives* (e.g., domain model, graphical concrete syntax, context conditions, interpreter, compiler, etc.) of a specific language workbench.

Language Concern reuse is achieved thanks to three interfaces: A *variation interface* expressing variabilities in the Language Concern in terms of features (shown in `textwriter` font in this section) from which a user, i.e., a language engineer, can select his desired configuration; a *customization interface* allowing to integrate a Language Concern into another *language* or Language Concern; and a *usage interface* exposing the relevant information needed to use the functionality that a concern provides to a *language* or another Language Concern. Section 2.3 introduced two examples of Language Concerns (Expression and Unit), albeit without explicitly defined interfaces yet.

Language: A *Language* is a specific kind of *language concern* where there is no more variability and no more customization to be done. For instance, if the Expression *language concern* is configured by selecting `Literal` and `Arithmetic Operation` (but not `Variable` and `Comparison Operation`) and customized by defining the `Literal` as `Double` values, and nothing is left in the *variation interface* and *customization interface*, then the result is a specific Expression Language, i.e, one among all the potential Expression Languages derivable from the Expression *Language Concern*. Section 2.3 introduced the exemplary Timed State Machine Language, which by design does not offer any variability or customization possibilities.

Metalanguage: All *artifacts* within a *language concern* are defined by means of *Metalanguages*. For instance, an Expression *language concern* can be defined with *artifacts* written in two metalanguages: Ecore (and possibly OCL) for specifying the structural properties and Java for specifying the behavioral properties of a semantics *perspective*.

Perspective: A *Perspective* groups a set of *metalanguages* and defines relationships among them. The goal of the Perspective is to define how several *metalanguages* are supposed to be used together in a coherent way to achieve a specific purpose. It is also

possible to define a Perspective containing only one *metalanguage* in order to make it explicit that the *metalanguage* is being used for a specific purpose. For example, to express the concrete syntax of a *language*, a concrete syntax Perspective can be defined that uses a specific *metalanguage*, e.g., BNF. Another concrete syntax Perspective can be defined that uses a different *metalanguage*, e.g., Xtext.

Artifact: An *Artifact* belongs to a *facet* of a *language concern* and conforms to a specific *metalanguage* of the *facet's perspective*. In our Expression *language*, an example would be a text file containing the BNF grammar of the concrete syntax for the `Literal` *facet* of the `Literal` feature.

Facet: A *Facet* realizes part of a given *language concern* using the *metalanguages* of the corresponding *perspective*. For example the grammar of the concrete syntax of an Expression *language* is a Facet of the Expression *language concern*. *Artifacts* of a given facet are kept consistent according to the *relationships* among the used *metalanguages* of the respective *perspectives*.

Variation Interface (VI): The *Variation Interface* of a *language concern* exposes a set of end user relevant *language concern features* and the constraints between them. A *language concern feature* represents any user visible configurable part of a *language concern* (e.g., design choices, alternative implementations, selection of a language capability...). A common approach for specifying variations is the use of Feature Models (FMs). Such a FM may be complemented by Goal Models that specify the impact of a particular configuration. For example, a State Machine concern could define different variants of state machines, such as Timed State Machines and Stochastic State Machines, with semantics implemented either with a compiler or an interpreter and two concrete syntax alternatives: a tree editor and a textual editor.

Customization Interface (CI): The *Customization Interface (CI)* of a *language concern* explicitly defines which elements of the *artifacts* need to be tailored to integrate the *language concern* into a given context (i.e., into a *language* or *language concern*). For example a Collection *language concern* could define the construct `OrderedList` that is used to order `Comparable` elements. How the elements are compared is outside of the scope of the Collection *language concern*. It is the task of the user to specify the notion of `Comparable` element in terms of its context.

Usage Interface (UI): The *Usage Interface (UI)* of a *language concern* is the aggregation of the relevant information of the *artifact* that is required to achieve a specific functionality. For example, in order to statically check the soundness of an instance of the *language* the user only needs to have access to the `check` operation. Note that a Usage Interface (UI) might introduce additional specifications (e.g., a particular protocol) in order to ensure the proper use of the functionality.

Library of Language Concerns: A *Library of Language Concerns* is a collection of unambiguously identifiable reusable *language concerns*, and a collection of *perspectives* (i.e., that are usable to define facets in that library).

As shown in Fig. 2, the left part (swimlane *Language*) provides the concepts required to define language concerns, while the right part correspond to the facilities provided by the language workbench (swimlane *Language Workbench*). Hence, there is a type-instance relationship in between these two parts: a facet conforms to a perspective in terms of the artifacts and links defined according to the metalanguages and relationships provided by

the corresponding perspective. The bottom part of Fig. 2 (swimlane *Concernification*) provides the concepts required to encapsulate as language concern and structure the various facets of the actual language implementation (top part of Fig. 2, swimlane *Implementation*).

Fig. 3 provides a possible instantiation of the conceptual model for a particular language concern. The CI is provided in the form of a feature model on the left, while the right part of the figure illustrates different facets, each composed of different artefacts. These artefacts can be defined from scratch, or sliced from a 150% model (see in gray the facets 2 & 3 conforming to the perspective P2).

3.2. Roles

Creating, evolving, and (re-)using concern-oriented modeling languages requires various skill sets. Within COLD we distinguish four different roles.

Language Workbench Engineer: A language workbench engineer defines the *perspectives* provided by language workbenches and the related tooling (e.g., editors for the *metalanguages* of choice).

Language Concern Engineer: A language concern engineer creates reusable *language concerns*. To this end, he defines *interfaces* and the *facets* implementing a *language concern*. Each of these *facets* is compatible with the *perspective* of the language workbenches of choice.

Language Engineer: A language engineer creates *languages* by reusing *language concerns*. In the process, he selects the required features of the individual *language concerns' VIs*, tailors these according to their *CIs*, and obtains a *language* usable through the reused *language concerns' UIs*.

Language User: A language user uses a *language* to model. Some configuration choices of *language concerns* may also be left to the language users, such that she can configure the *language concerns* using their *Variation Interface* before using the resulting *language*.

3.3. Scenarios

COLD is expected to cover various scenarios related to language reuse, to create a brand new language, to evolve or adapt an existing one, or to contribute new reusable *language concerns*. The underlying principle of COLD is to create a new *language* through the combination of *language concerns* by individually configuring (i.e., selecting features and/or re-exposing some of them) and customizing them. For example, one wants to create a new *language* by combining the *language concern* of a State Machine, with an Action *language concern* to be used for expressing actions on the transitions of the state machine. The language engineer can select the expected features in the two *language concerns*, and then customize them in order to symmetrically integrate them in a final *language*.

A closely related scenario consists in extending an existing *language* by integrating an additional *language concern* into it. For example, one wants to integrate a Parallel Loop *language concern* into an existing Action *language concern*. In this case, the composition is asymmetric, composing the *facets* of the reused Parallel Loop *language concern* with the *facets* of the Action *language concern*.

Beyond the planned variability and customization of a *language concern*, or beyond a given *language*, a language engineer may want to add new features to an existing *language*

concern by adding a feature to the *VI* and adding the corresponding *language facet*, or to adapt an existing *language facet* to fit a particular purpose. In the later case, the change of the *language facet* should not have an impact on the *UI*. Similarly, a language engineer may want to adapt an existing *language* to evolve it according to an evolution of the understanding of a given domain. Finally, a language engineer may want to extract a slice of an existing *language* to be used as a standalone *language* or *language concern*.

In all the aforementioned scenarios, when the time comes for a language engineer to build a new *language* or *language concern*, she first needs to understand whether existing *language concerns* already offer the expected features to be configured to satisfy the requirements for the desired *language*.

4. Concernification of Languages

The COLD approach relies on the definition and the reuse of Language Concerns. We use the term *concernification* for the process of defining a piece of language as a language concern, including an explicit definition of the three proposed interfaces (see bottom part of Fig. 2, which structure as a language concern the actual language implementation in the top part of Fig. 2). In this section, we first explain in more detail the three interfaces that characterize a Language Concern. We then present the life cycle of Language Concerns, which involves different actors around a common Language Concern Library.

4.1. Language Concern Interfaces

A Language Concern is primarily characterized by its interfaces. In this subsection, we take a closer look at each of the three Language Concern interfaces defined in Section 3.1, namely the Variation Interface (VI), the Customization Interface (CI) and the Usage Interface (UI).

4.1.1. Language Concern Variation Interface

In order to define reusable Language Concerns, the Language Concern Engineer must explicitly express the *variation points* implemented in the Language Concern facets, which enable the closed variability of the Language Concern. Those design choices are formalized in the Variation Interface (VI). Defining the VI requires constructs for specifying and resolving variability at an abstract level, i.e., without exposing details about the Language Concern implementation.

Variability specifications are technically similar to *features* in feature modeling. Additionally, the set of features exposed in a VI can be linked to *goal models* provided by the Language Concern Engineer to inform the Language Engineer about non-functional properties of features (e.g., impact on performance). Based on the exposed features, their relationships, and the provided goal models, the Language Engineer can choose a *configuration* for the Language Concern, i.e., a valid selection of features.

Once a feature has been defined in the VI, the impact of the selection of the feature on the different facets must be specified in the Language Concern. For example, selecting a feature may trigger the addition of one or several constructs (with their own syntax and semantics) to the Language Concern, or alter specific parts of artifacts scattered in multiple facets, or may even condition the presence of a complete facet of the Language Concern. Due to the large number of possibilities, the COLD approach does not make any

assumption on the *feature granularity* of the VI (i.e., the number of features, and which elements are conditioned by which feature) and it is the responsibility of the Language Concern Engineer to define a relevant and well structured VI for the Language Concern.

Consider the use case described in Section 2.3. A possible VI for the Expression Language Concern could include two features: one conditioning the presence of the *Literal* concept, and another conditioning the presence of the *Variable* concept. When reusing this concern, the Language Engineer can use the VI to tailor the Language Concern by selecting features of interest, e.g., to create a Language without variables.

In order to define the VI and the corresponding variation points of a Language Concern, the metalanguages provided by the considered language workbench must provide several facilities. First, it must be possible to declare features and the relationships between them (e.g., with a dedicated metalanguage for feature modeling). Second, it must be possible to specify how the choice of features affects the different facets, which may require the use of *operators* to manipulate the artifacts of the Language Concern. These operators are specific to the provided metalanguages, and may include the possibility to compose multiple artifacts (also called *positive variability*) or to remove parts of artifacts (also called *negative variability*).

The definition and use of the VI raises many challenges. The impacts of the choice of granularity on the Language Concern definition are studied in Section 6.1.1. The implication of the goal model in the context of COLD is studied in details in Section 6.2.1.

4.1.2. Language Concern Customization Interface

The Customization Interface (CI) allows Language Concerns to be designed as generically as possible to foster their reuse in new contexts. Its purpose is to describe how a Language Concern can be *adapted* (or *molded*) to the specific needs of a given context of reuse. In other words, the CI enables the open variability of the Language Concern.

The CI of a Language Concern can be thought of as the contract that must be fulfilled by another reusing Language Concern or Language. This is expressed by a set of *customizable* elements (i.e., "holes") that must be *bound* to concrete elements of the reusing Language Concern or Language (i.e., "filling the holes"). Note that what is exposed in the CI depends on the choices made by the Language Engineer with the VI, since the resulting configuration may condition the presence or absence of elements, and consequently of their own CI.

Consider the use case described in Section 2.3, where the Unit Concern must expose a CI for both its abstract syntax and semantics facets. Regarding the abstract syntax, a customizable concept called *Value* can be defined with a link to *Unit*. This means that to reuse the Unit Concern, one must provide a concrete concept to play the role of *Value*. Regarding the semantics, the customizable *Value* concept can expose a customizable *evaluate* operation where unit conversion logic is called before the actual evaluation logic, the latter being provided by the concrete operation bound to *evaluate*. A Language Engineer can then use this CI to bind a concrete concept *Expression* from the Expression Concern, to the customizable concept *Value* of the Unit Concern, leading to expressions with units and automatic unit conversion facilities.

In order to define the CI of a Language Concern, and to be able to use this CI later on, the used metalanguages must provide several facilities. First, each considered metalanguage must provide a way to identify which elements are customizable, i.e., which elements are part of the CI. For instance, a grammar language can identify as customizable

every non terminal elements without corresponding production rule, and a metamodeling language can identify as customizable every class that is *abstract* or that has *generics*. Additional consistency constraints may be required either within the CI, or across the CIs of the different facets of a concern. For example, the customization of a specific part of the abstract syntax (e.g., binding a generic type n times) may have consequences on the possible or required customization within the corresponding part of the operational semantics (e.g., managing n cases within a `switch` expression).

Second, each metalanguage must provide a set of *composition operators* that can be used to bind concrete elements to customizable elements. For example, both *inheritance* or *package merge* can be used to bind a concrete class to an abstract class in an abstract syntax, and an abstract method can be *implemented* by a concrete method in the semantics. Aspect-oriented approaches are also possible for more fine grained composition operators. Depending on how the composition operators are implemented, there may be consequences on the non-functional properties of the generated language, such as incremental compilation, execution speed, or analyzability.

We identify several challenges related to the definition of the CI and the composition of Language Concerns in Section 6.2.2.

4.1.3. Language Concern Usage Interface

The Usage Interface (UI) is the set of the relevant functionalities of the Language Concern that can be externally requested. The UI definition, appearing from the Language Concern configuration and customization, is the cornerstone of the encapsulation strategy promoted by the COLD approach and initially proposed by Parnas [39, 40].

The functionalities exposed in the UI may either be structural or behavioral elements of the artifacts. The UI represents the comprehensive set of operations provided by the concern, and consequently the only interface exposed to Language Users (to create and use instances) and to both Language Engineers and Language Concern Engineers (to understand what a Language Concern provides). The UI is not only an explicit set of the exposed functionalities of the concern, but also a mean to analyze Language Concerns and Languages, e.g., to see whether different Language Concerns and Languages provide the same functionalities (for conformance checking, substitutability, etc.). Besides, the UI can be used as an explicit contract exported by the Language Concern engineers, ensuring that the reuse of the Language Concern does not lead to undocumented breaking changes in the UI.

From the use case described in Section 2.3, consider an Expression Language that has been defined using the Expression Language Concern and composed of two perspectives: the abstract syntax and the operational semantics. The UI of such a language would contain both the model manipulation operations for the abstract syntax perspective (such as the instantiation of an *Add* operation or the assignment of a *Literal* to the left part of the *Add*) and an evaluation operation for the semantics perspective (such as returning a numerical value from an expression model).

In order to define the UI of a Language Concern, the considered metalanguages must provide a way to expose parts of the artifacts of a facet. The Language Concern Engineer is free to use either the encapsulation capabilities of the metalanguages (e.g., Java's `public` classes keyword), or some external specification (e.g., Eclipse plugin's `Export-Package` property), to define which parts of the artifact's functionalities can be used by the Language Concern user.

We identify one main challenge related to the definition of the UI in the context of Language globalization in Section 6.2.3.

4.2. COLD Life Cycle

The Language Concern *Library* contains a set of language concerns to ease the development of new software languages and to support a wide spectrum of scenarios. In this subsection, we discuss the life cycle of the language concerns in the context of the different scenarios of reuse introduced in the previous section.

Language Engineers may use pre-existing Language Concerns, or build new ones from scratch, or act as Language Concern Engineer to contribute new Language Concerns to the Language Concern Library. For example, unlike in the use case presented in Section 2.3, a Finite State Machine (FSM) may be created by a Language Engineer by selecting applicable features inside a pre-configured Language Concern dedicated to classical FSM. On the other hand, a Language Engineer may use the FSM Language Concern as a starting point and integrate additional Language Concerns to yield the desired FSM, for example a FSM dealing with Units – following the development scenario presented in Section 2.3. And lastly, a Language Engineer acting as Language Concern Engineer, may start with a blank slate and integrate several Language Concerns to build a new Language Concern. This gives him/her complete freedom and flexibility about how the different concerns are integrated, but requires in-depth knowledge of the elements to integrate. Starting from a blank slate does not allow the Language Concern engineer to benefit from the implementation choices done by a previous Language Concern Engineer when defining the integration of many Language Concerns and can be seen as a missed opportunity to reuse existing components.

When several Concerns are composed with each other into a language, a natural order of composition is imposed by the existence of required language elements. For example, to compose the Unit Concern, an element needs to exist in the language to which the `Literal` can be mapped. In addition to this natural order, the order of composition is largely influenced by the properties of the composition operator. The way language components associated to Language Concerns can be composed has a strong impact on the usage of the composition framework used to assemble the concerns into a single language. A commutative and associative composition operator—while simple to use theoretically—is known as difficult to apply to real case studies because of its restrictiveness, and must usually be degraded to support real-world needs. For example, considering the commutativity property, the class merging operator was less used than the AspectJ weaving one (which is asymmetric and non commutative). By losing the commutativity property, it is up to the user to know how the different concerns must be composed together, to avoid the capture of unintended elements during the composition (so called *fragile pointcut* in the aspect community). To address this problem, one solution is to enforce a composition process with less freedom, and as the user cannot deviate from a given process, composition conflicts can be anticipated. Consider for example staged configuration in software product lines: "*The process of specifying a family member may also be performed in stages, where each stage eliminates some configuration choices. We refer to this process as staged configuration. Each stage takes a feature model and yields a specialized feature model, where the set of systems described by the specialized model is a proper subset of the systems described by the feature model to be specialized.*" [41]. Applied to Language Concern composition, a staged approach supports the Language

Engineer by restricting his/her choices and guiding him/her in the realization of his/her language. On the other side of the spectrum, the Language Engineer can be freed from this multi-staged process and benefit from more flexibility. In COLD, the composition order is implicitly specified by the feature hierarchy within a language concern, and the concern reuse hierarchy between language concerns.

5. Supporting COLD Across the Technological Spaces

Concern-oriented language development is already supported to varying degrees by existing language workbenches. This section presents how popular metamodel-driven and grammar-driven language workbenches support aspects of COLD.

5.1. The GEMOC Studio

The GEMOC Studio [42] is an extensible metamodel-based language workbench built on top of the Eclipse Modeling Framework (EMF) [43]. It includes Melange [5], a metalanguage for assembling and composing different DSLs using language-level operators. During the execution of a model, several generic or generated runtime services are provided, such as a graphical animator, a trace manager, and an omniscient debugger.

Perspectives. The abstract syntax of a DSL is defined by an Ecore metamodel (possibly complemented with OCL constraints), while the operational semantics is defined with Kermeta [44], possibly complemented with MoCCML [45] and ECL [46]. Experimental support is also provided for specifying the semantics using xMOF [47] and ALE [32]. For the concrete syntax, other tools of the Eclipse ecosystem, such as Xtext or Sirius, are supported. The BCOoL metalanguage [48] allows to specify coordination patterns between heterogeneous languages to coordinate the concurrent execution of conforming models. In summary, the GEMOC Studio offers the following perspectives: abstract syntax, concrete syntax, various kinds of semantics, and coordination.

Language Concerns. From the abstract syntax and semantics of a DSL, Melange extracts a structural interface called the *model type*, which specifies what can be publicly accessed or executed on a model [3]. Hence, model types roughly correspond to the UI in COLD. In addition, the ECL metalanguage can be used to explicitly expose the *symbolic event structure* of the concern, which correspond to a purely behavioral UI. Regarding the CI, the metalanguages can define *abstract* elements that must be implemented later on (e.g., an abstract Ecore class, or an abstract Kermeta method). There is also experimental support with the Puzzle tool [49, 50], which can be used to define the VI as a feature model using the Common Variability Language (CVL). To support configuration, the concern must be broken down into different language modules that can be composed together using Melange operators (i.e., inheritance, slicing, merging, weaving) based on a valid configuration of features.

Libraries. As the GEMOC Studio is based on Eclipse, Language Concern Engineers can package Language Concerns as Eclipse plugins and push them on a remote update site, allowing Language Engineers to install and compose them on demand.

Use Case. The Expression and Unit reusable Language Concerns can each be developed independently as a GEMOC executable DSL composed of an abstract syntax and a semantics. If Puzzle is used, the VI of the Expression Concern can be defined as two features *Literal* and *Variable*. The CI of the Unit Concern can include an abstract class

Value linked to the *Unit* class. Using Melange operators (e.g., *merge* and *rename*), both Language Concerns can be reused into a Timed State Machine Language, where the *Value* class of Unit is merged with the *Expression* class of Expression. Glue code is required to combine the semantics of Unit and Expression (e.g., to realize unit conversions before arithmetic operations). The remainder of the Timed State Machine Language can then be defined, including a *Trigger* relying on *Literal* reused from Expression (which now also has a *Unit*, e.g., "seconds"), and a *GuardExpression* that contains a *ComparisonOperator* from *Unit* (which now convert units if required).

Towards COLD with The GEMOC Studio. Some of the GEMOC Studio's features, such as structural language interfaces and language-level composition operators, could be directly reused to implement the COLD philosophy. Support for the VI with Puzzle is still experimental as of today, and can be improved for better COLD support. Lastly, while model types can be used to define the UI, they could also be used for the CI by using model typing to describe the structure of the customizable elements.

5.2. MontiCore

MontiCore [51] is an extensible workbench for the development of compositional languages. Its languages are defined in terms of grammars that specify the concrete and abstract syntax of modeling languages. These grammars moreover support multiple inheritance and embedding, which MontiCore leverages to compose languages. From these grammars, MontiCore generates model-processing infrastructure (e.g., parsers) to facilitate checking and transforming models of the language.

Perspectives. The quintessential perspectives of MontiCore are integrated concrete and abstract syntax, well-formedness, and behavior. Syntax is realized through extended context-free grammars of the MontiCore metalanguage from which MontiCore generates parsers, abstract syntax classes, and additional model processing infrastructure. The latter includes an extensible visitor-based model checking framework that applies Java well-formedness rules (context conditions) to the models. Behavior can be realized through FreeMarker-based code generators that process models and translate these into arbitrary target language artifacts or through domain-specific model-to-model transformations [52]. Facets realize these perspectives through specific grammars, Java rules, and templates.

Language Concerns. A MontiCore language comprises at least three facets that correspond to the quintessential MontiCore perspectives (syntax, well-formedness, and behavior). The notion of a *variation interface* of the FSM language in MontiCore is realized by the selection of context conditions to be activated. The language engineer selects from context conditions provided by the language concern. The *customization interface* is reflected in syntax and well-formedness. Syntax can be customized by grammar extension, via implementing interface rules that serve as dedicated grammar extension points, or through adding alternatives by overriding productions. The well-formedness checking infrastructure supports additional customization through an interface that enables adding new composition-specific well-formedness rules. The *usage interfaces* are the methods generated into their abstract syntax classes. E.g., these include methods to compare models, which can be used by language developers for various analyses.

Libraries. MontiCore supports reusable grammars, context conditions, and model transformations. Languages comprising these are available for reuse online. A dedicated language (concern) library is not available.

Use Case. For integration of the three language concerns presented in subsection 2.3, the resulting grammar (syntax facet) inherits from the grammars of the syntax facet of the other concerns and implements their interfaces (e.g., `Guard Expression`) to define a new syntax for the Timed State Machine language concern. Context conditions of the other concerns are selected as appropriate and new inter-language context conditions are added.

Towards COLD with MontiCore. Concern-oriented composition currently is purely syntactical. While there have been approaches to compose code generators [53], these yet have to be generalized. Also, MontiCore currently does not support 150% grammars, i.e., grammars that support variability through features. Also, MontiCore does not support libraries for concerns yet.

5.3. Neverlang

The *Neverlang* [6] language workbench is built around the *language feature* concept. Language components, called *slice*s, embodying the language features are developed as separate units by tying together some *module*s. A module may contain a `syntax` definition and/or one or more semantic `role`s. Each slice defines and needs one or more types. These are declared in `provides` and `requires` clauses, respectively. Finally, the `language` descriptor indicates which slices are to be composed together, which roles should be used and in which order the generated tooling will use them.

Perspectives. Some of Neverlang's perspectives are: composable concrete and abstract syntax, staged and dynamically customized behavior, and *ad hoc* modular DSLs to support the generation of language artifacts. Syntax is expressed by context-free attributed grammars. The semantic roles define the behavior. Each role represents one Neverlang production step whose execution defines one of the possible language artifacts: parser, interpreter, compiler, debugger, IDE, etc.

Language Concerns. A language concern is realized by a slice (or a set of slices). A set of slices without any further *need* for composition—i.e., all used nonterminals are provided by at least one syntax rule—is a language in the COLD parlance. The concrete syntax is given through EBNF rules and the abstract syntax—as an AST—is the internal representation created by Neverlang over slices composition [54]. Limited to the composition functionality, the *language concern usage interface*—in Neverlang—is realized by the `provides` clause. `provides` and `requires` clauses are also used by the AiDE [8] tool to automatically build a configuration tree (a CVL [55]) out of the available slices making explicit the existing variability. In the COLD parlance, the configuration tree built by AiDE realizes the *language concern variability interface*. Neverlang provides several ways to customize a language feature, e.g., substituting a semantic role, substituting its syntax role and—if needed—re-mapping the existing semantic actions on the new rules, or dynamically injecting new semantic actions on a specific node of the AST [56]. The `requires` clause is the only explicit support for the *language concern customization interface*; this clause declares which elements to customize in order to make the language concern composable.

Libraries. Neverlang supports separate compilation: a slice or a set of slices can be compiled even if their requirements are not satisfied—i.e., one or more nonterminals are undefined—, and shipped pre-compiled—i.e., with an (incomplete) parser that will be extended when a further composition adds new rules [54]. Neverlang provides a `bundle`

COLD Feature	The GEMOC Studio	MontiCore	Neverlang
Variation Interfaces	Puzzle (CVL)	context condition selection API	AiDE (CVL)
Customization Interfaces	inheritance, type parameters	required grammar interfaces, adding context conditions	implicit, "`requires`" clauses
Usage Interfaces	model types, and symbolic event structure	implicit	implicit, "`provides`" clauses
Perspectives	Abstract Syntax (**AS**), Concrete Syntax (**CS**), Operational Semantics (**OS**), Coordination (**Coord**)	Integrated Abstract and Concrete Syntax (**IS**), Well-formedness (**WF**), Translational Semantics (**TS**)	Parser, Abstract Syntax (**AS**), Operational Semantics (**OS**), type checker, code generator (**TS**), code optimizer (**TS**), debugger (**UI**), IDE (**UI**), code translator (**TS**)...
Metalanguages	**AS:** Ecore; **CS:** Xtext, Sirius; **OS:** Kermeta, MoCCML, ECL, Ale, xMOF, Java (or JVM based languages) **Coord:** BCOoL	**IS:** MontiCore Grammar Language; **WF:** Java Context Conditions; **TS:** FreeMarker, Java	**Parser:** LALR parse tree; **AS:** EBNF; **OS**, **TS:** Java (or JVM based languages); **UI:** DSL
Reuse Operators	inherit, merge, slice, import references, compose, extend coordination	inherit, aggregate, embed	import, re-map, aggregate, slice, cross-references

Table 1: Summary of current COLD support in different language workbenches

statement to import as-it-is a pre-compiled language concern into the one under development. Separate compilation and bundles enable Neverlang to support the development of libraries of language concerns as envisaged by the COLD methodology.

Use case. In Neverlang, the considered language concerns are realized by a set of slices. *Unit* provides the expression units—i.e., time and length units—and the conversion operations; it needs the numeric literals. Whereas, *Expr* provides dimensionless numeric expressions—i.e., literals and related operations. Their composition occurs on the numeric literals. Unit conversions are applied in a separate role—a facet of the final language—before expressions evaluation to maintain the concern separation.

Towards COLD in Neverlang. The concern composition is mostly syntactical with limited control on semantic composition [57]. Moreover, even if Neverlang provides the mechanisms to customize a language concern, no explicit interface is available.

5.4. Summary

Table 1 describes the feasibility of COLD across the technological spaces. The first column shows COLD features, the subsequent columns represent the technological spaces and their cells summarize how the respective feature may be supported.

6. Discussion

In this section, we discuss some of the open issues related to COLD. First, Section 6.1 focuses on open issues related to the implementation and design of Language Concerns, and the support of COLD in language workbenches. Then, Section 6.2 discusses the open issues resulting from the use of Language Concerns.

6.1. Open Issues Related to Language Concern Implementation

This subsection discusses open issues related to the implementation and design of Language Concerns, i.e., open issues that only impact the Language Concern Engineer. We first discuss the granularity of a Language Concerns, then the breakdown Language Concerns, and finally the impacts of composition operators.

6.1.1. Language Concern Granularity

The long studied field of object-oriented software development led to the definition of well defined guidelines for choosing the right *granularity* of objects [58], i.e., how to distribute roles and constructs among objects. In particular, this lead to the definition of *design patterns* [59] that guide developers with well studied and reusable patterns of object-oriented software design. In the context of COLD, to produce highly reusable language concerns requires to address the problem of defining their granularity. In principle, language concerns may range from small generic language concerns (e.g., Expressions) to large framework-like language concerns (e.g., for the definition of automotive business processes). Yet, a set of generalized and well-studied granularity guidelines would be of great benefit for the production of language concerns.

6.1.2. Language Concern Breakdown

Breaking down software is a key concept of software modularization that has already been studied both for General Purpose Languages (GPLs) [8, 60] and Domain Specific Languages (DSLs) [61]. Essentially, breaking down software into modules (or units) facilitates both software development and maintainability by easing the creation and modification of modules independently of other modules. Besides the well-studied feature interaction problem[4], the following open issues need to be considered.

In the context of COLD, the proper way to break down a language is still an open issue. However, two possible kinds of modules naturally emerge for the breakdown of Language Concerns: language constructs (i.e., basic blocks of a language possibly implemented in many facets, as in Neverlang) and facets (as in Rascal). If a module is defined as a language construct, adding constructs to the Concern (or removing constructs) is facilitated. This has already been identified as a valid strategy for languages, especially in the context of GPLs [7, 8, 62]. However, if a whole facet (e.g., a concrete syntax) must be added to the Concern, then all modules must be modified (e.g., to define the concrete syntax of each construct) even if this effect can be alleviated by the use/definition of proper composition operators (see the introduction of a "permission check" facet in [61] as an example). Conversely, if a module is defined as a facet (similarly to what is done in Rascal [20]), adding a new facet is facilitated while adding a new construct requires modifying all modules. In summary, both strategies come with a strong trade-off, and picking a strategy requires anticipating how a given Language Concern may evolve in the future.

Since software is designed to grow over time [63], it is important to facilitate the extension of Language Concerns both on the axes of language constructs and facets. Therefore, future studies should explore more sophisticated breakdown strategies, possibly combining aspects of both the strategies presented above.

6.1.3. Impacts of Composition Operators

As explained in Section 4, syntactic and semantic composition operators [64] are required by COLD both to realize a valid configuration of the VI, and to bind concrete elements to customizable elements of the CI. However, in practice, the use of specific

[4]e.g., International Conference Series on Feature Interactions in Telecommunications and Software Systems; http://www27.cs.kobe-u.ac.jp/wiki/icfi/index.php?History

composition operators can impact several aspects of concern implementations, such as incremental compilation, type safety, or independent extensibility. For instance, while an abstract syntax defined using the Ecore metalanguage can be extended by inheritance and referenced without recompilation, *merging* two Ecore models would allow the introduction of new fields in an existing Ecore class forcing the recompilation.

Consequently, future work must both study the impacts and trade-offs of composition operators, and identify (or define) relevant composition operators for given implementations of the COLD approach. This problem is strongly related to the topics of granularity (see Section 6.1.1) and breakdown (see Section 6.1.2), as composition operators are of first importance in both cases.

6.1.4. Language Concern Composition Issues

The availability of well-defined language reuse interfaces can greatly facilitate the production of sound Languages through the reuse of independently engineered Language Concerns. However, issues may *emerge* from the composition of valid Language Concerns (i.e., with sound interfaces and implementations), which would result in an unsound Language. For instance, a concept exposed in a Language Concern interface might come with hidden assumptions (e.g., stateful initialization or hidden dependencies) – unknown from the Concern Language Engineer, or the composition of artifacts can lead to unsound behaviors (e.g., the composition of two valid BNF grammars can produce an ambiguous grammar).

To automatically detect and prevent such emerging issues may be difficult. However, the detection of composition issues is a critical aspect to support the language engineer in identifying possible issues with the language he/she is building, which may lead to either erroneous or unexpected behaviors. While some potential issues can be identified before the composition (e.g., detection of ambiguity), others, especially related to soundness of the resulting language semantics are still subjected to research.

Experience from engineering a variety of languages for research and industry indicates that the guidance provided by the different language concern interfaces as implemented in the different workbenches (see Section 5) already substantially facilitate language engineering. Nonetheless, further research must be dedicated to investigate issues emerging from Language Concern composition and evaluate COLD in the context of real industrial language reuse scenarios.

6.2. Language Concern Reuse

This subsection discusses open issues resulting from the reuse of Language Concerns, i.e., open issues impacting both the Language Concern Engineer and the Language Engineer. We discuss first the difficulty of defining comprehensible goal models, then the overall reuse process, and finally the impacts of COLD on the globalization of software languages.

6.2.1. Exposing Comprehensible Goal Models

As explained in Section 4, the VI of a Language Concern can include different goal models to expose the impacts of features on non-functional properties. This information can be used by the Language Engineer to make informed decisions when selecting features. For instance, a Language Engineer building a language for rapid web language prototyping

may want to improve the reactiveness of the auto-completion of the language but may not require a formal type system for the language.

Consequently, Language Concern Engineers need to share a glossary of non-functional properties with the Language Engineer, and tooling [65] is required to help them quantifying the intentional elements (goals or tasks/features) of the Language Concerns. While existing templates for requirements documents form a good starting point for defining such a shared glossary, as they cover common non-functional requirements and qualities, determining the set of intentional elements useful to the Language Engineer is still a topic of research.

6.2.2. Simplifying the Reuse Process

While the COLD reuse process relies on three well identified phases (selection of features from the VI; customization of elements from the CI; use of the Concern with the UI), the Language Engineer must still face different intricacies.

A first problem is the large number of possible combinations of choices when using the VI and the CI, which leads to a very large number of possible combinations of artifacts. Furthermore, features selected in the VI will impact the content of the CI, and will therefore restrict the number of remaining choices in the customization phase, which can be hard to forecast for the Language Engineer. Finding ways to help the Language Engineer to deal with the vast number of possibilities is an open issue.

A second problem is related to the use of composition operators by the Language Engineer. When composition operators are used by Language Concern Engineers to compose artifacts and facets, these engineers are aware of the internal details of the Language Concern, and are experts in Language reuse. But when composition operators are used by Language Engineers to customize concerns, the implementation details are encapsulated. Consequently, a Language Engineer does not have the same level of knowledge—and expertise—as a Language Concern Engineer, and therefore cannot be expected to use the same type of language composition operations. Identifying or defining relevant composition operations for Language Engineers is currently an open issue.

6.2.3. Language Concern Globalization

The globalization of Software Language Engineering has already been discussed by Bryant et al. [66] and is defined by Clark et al. [67] as follows: "*Globalization deals with the purposeful construction, adaptation, coordination and integration of explicitly defined languages, to be amenable to mechanical and cognitive processing, with the goal of improving quality and reducing the cost of system development.*" A conceptual model for globalization is also proposed by Clark et al. [67], with concepts such as *composition* and *relationships*. They also identify a wide range of challenges that must be addressed in the context of globalization, both from a socio-technical point of view (*How to communicate across language engineering teams?*), and from a purely technical point of view (e.g., *How to model the reuse and composition of languages?*).

COLD already covers a large portion of the concepts identified in this conceptual model, and can therefore be considered as a relevant conceptual approach to address part of the globalization challenge. Yet, further research is required to establish which socio-technical and technical challenges of globalization are addressed by COLD, and how to further extend COLD to fully address the globalization challenge.

7. Conclusion

DSL development has reached a level of maturity where creating DSLs has become ordinary, despite requiring substantial development efforts. Fostering and supporting reuse in DSL engineering is hence of crucial importance. Based on existing language reuse approaches, we introduce Concern-Oriented Language Development (COLD) to unify DSL engineering. COLD applies separation of concerns to language constituents, making them *language concerns*, which are configurable units of reuse that encapsulate constructs of a language visible to the language engineer. A language concern can reuse other concerns recursively. They consist of artifacts representing abstract syntax, concrete syntax, and behavioral semantics. Artifacts are arranged in facets and perspectives to organize relationships between language constituents, languages, metalanguages, and language workbenches. COLD usage is organized around three language concern interfaces (variation, customization, usage), following principles already applied to general-purpose software development with concerns [11]. This enables systematic language (concern) reuse life cycles. To illustrate the COLD notions, we presented some language concerns and the current state of their realization with metamodel-based and grammar-based language workbenches.

COLD reveals many interesting challenges, which we surveyed in Section 6. To support language concern implementation, guidelines to the language engineers must be provided as many forms and sizes of concerns are likely to emerged, e.g., generic constituents or domain concerns. Such guidelines should cover how to determine the right granularity of a language concern, how to break a language into concerns, and how to deal with cross-concern semantic issues. We also expect this work to provide insights into how to identify a relevant set of metalanguage composition operators in order to foster, or ideally maximize, language concern reusability while taking into account expressiveness, safety, or encapsulation of the artifacts. Hence, devising a set of intentional elements for language concerns while considering non-functional properties is a research line we believe is worth investigating, taking goal modeling as a starting point.

Finally, in order to realize the vision of COLD, we are convinced that two main steps are to be taken. First, we need to experiment its application to gather feedback, especially in terms of complexity and applicability. To do so, we expect to explore adapted tool support, and socio-technical issues on communication and coordination of such concerns among teams. Second, we plan to investigate appropriate means to share and distribute language concerns libraries on a large scale, establishing a form of globalization [66] of language concerns.

References

[1] J.-M. Favre, D. Gasevic, R. Lämmel, E. Pek, Empirical Language Analysis in Software Linguistics, in: SLE, Springer, 2010, pp. 316–326.

[2] P.-A. Muller, F. Fleurey, J.-M. Jézéquel, Weaving executability into object-oriented meta-languages, in: Proceedings of MODELS/UML'2005, Montego Bay, Jamaica, 2005.

[3] T. Degueule, B. Combemale, A. Blouin, O. Barais, J. Jézéquel, Safe model polymorphism for flexible modeling, Computer Languages, Systems & Structures 49 (2017) 176–195.

[4] J. de Lara, E. Guerra, Generic Meta-modelling with Concepts, Templates and Mixin Layers, in: 2010 ACM/IEEE 13th International Conference on Model-Driven Engineering Languages and Systems (MODELS), Oslo, Norway, 2010, pp. 16–30. doi:10.1007/978-3-642-16145-2_2.

[5] T. Degueule, B. Combemale, A. Blouin, O. Barais, J. Jézéquel, Melange: a meta-language for modular and reusable development of DSLs, in: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015, 2015, pp. 25–36. doi:10.1145/2814251.2814252.
[6] E. Vacchi, W. Cazzola, Neverlang: A Framework for Feature-Oriented Language Development, Computer Languages, Systems & Structures 43 (3) (2015) 1–40. doi:10.1016/j.cl.2015.02.001.
[7] E. Vacchi, W. Cazzola, S. Pillay, B. Combemale, Variability Support in Domain-Specific Language Development, in: M. Erwig, R. F. Paige, E. Van Wyk (Eds.), Proceedings of 6$^{\text{th}}$ International Conference on Software Language Engineering (SLE'13), LNCS 8225, Springer, Indianapolis, USA, 2013, pp. 76–95.
[8] T. Kühn, W. Cazzola, D. M. Olivares, Choosy and Picky: Configuration of Language Product Lines, in: G. Botterweck, J. White (Eds.), Proceedings of the 19th International Software Product Line Conference (SPLC'15), ACM, Nashville, TN, USA, 2015, pp. 71–80.
[9] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, B. Baudry, Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review, Computer Languages, Systems & Structures 46 (2016) 206–235.
[10] M. Schöttle, O. Alam, J. Kienzle, G. Mussbacher, On the Modularization Provided by Concern-Oriented Reuse, in: Companion Proceedings of the 15th International Conference on Modularity, MODULARITY Companion 2016, ACM, 2016, pp. 184–189. doi:10.1145/2892664.2892697.
[11] J. Kienzle, G. Mussbacher, O. Alam, M. Schöttle, N. Belloir, P. Collet, B. Combemale, J. DeAntoni, J. Klein, B. Rumpe, VCU: The Three Dimensions of Reuse, in: Software Reuse: Bridging with Social-Awareness - 15th International Conference, ICSR 2016, Limassol, Cyprus, June 5-7, 2016, Proceedings, 2016, pp. 122–137. doi:10.1007/978-3-319-35122-3_9.
[12] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA (Nov. 1990).
[13] M. Svahnberg, J. Van Gurp, J. Bosch, A taxonomy of variability realization techniques, Software: Practice and Experience 35 (8) (2005) 705–754.
[14] International Telecommunication Union (ITU-T), Recommendation Z.151 (10/12): User Requirements Notation (URN) - Language Definition (approved October 2012).
[15] M. B. Duran, G. Mussbacher, Investigation of feature run-time conflicts on goal model-based reuse, Information Systems Frontiers 18 (5) (2016) 855–875. doi:10.1007/s10796-016-9657-7.
[16] P. Naur, B. Randell, Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968, NATO, 1969.
[17] A. Pescador, A. Garmendia, E. Guerra, J. S. Cuadrado, J. de Lara, Pattern-based development of domain-specific modelling languages, in: 2015 ACM/IEEE 18th International Conference on Model-Driven Engineering Languages and Systems (MODELS), IEEE, 2015, pp. 166–175.
[18] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer, LISA: An Interactive Environment for Programming Language Development, in: N. R. Horspool (Ed.), Proceedings of the 11th International Conference on Compiler Construction (CC'02), Lecture Notes in Computer Science 2304, Springer, Grenoble, France, 2002, pp. 1–4.
[19] H. Krahn, B. Rumpe, S. Völkel, MontiCore: A Framework for Compositional Development of Domain Specific Languages, International Journal on Software Tools for Technology Transfer 12 (5) (2010) 353–372.
[20] P. Klint, T. van der Storm, J. J. Vinju, RASCAL: A domain specific language for source code analysis and manipulation, in: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009, 2009, pp. 168–177. doi:10.1109/SCAM.2009.28.
[21] L. C. L. Kats, E. Visser, The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs, in: M. Rinard, K. J. Sullivan, D. H. Steinberg (Eds.), Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10), ACM, Reno, Nevada, USA, 2010, pp. 444–463.
[22] E. Van Wyk, D. Bodin, J. Gao, L. Krishnan, Silver: An extensible attribute grammar system, Science of Computer Programming 75 (1-2) (2010) 39–54.
[23] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, in: ACM SIGPLAN Notices, Vol. 39, ACM, 2004, pp. 111–122.
[24] E. Visser, Scannerless generalized-LR parsing, Universiteit van Amsterdam. Programming Research Group, 1997.
[25] M. Emerson, J. Sztipanovits, Techniques for metamodel composition, in: OOPSLA–6th Workshop

on Domain Specific Modeling, 2006, pp. 123–139.
[26] P. D. Mosses, Modular structural operational semantics, The Journal of Logic and Algebraic Programming 60 (2004) 195–228.
[27] S. Liang, P. Hudak, Modular denotational semantics for compiler construction, Programming Languages and Systems—ESOP'96 (1996) 219–234.
[28] G. Roșu, T. F. Șerbănuță, An overview of the k semantic framework, The Journal of Logic and Algebraic Programming 79 (6) (2010) 397–434.
[29] M. Felleisen, R. B. Findler, M. Flatt, Semantics engineering with PLT Redex, Mit Press, 2009.
[30] V. Vergu, P. Neron, E. Visser, DynSem: A DSL for dynamic semantics specification, Technical Report Series TUD-SERG-2015-003.
[31] M. Gouseti, C. Peters, T. v. d. Storm, Extensible language implementation with object algebras (short paper), in: ACM SIGPLAN Notices, Vol. 50, ACM, 2014, pp. 25–28.
[32] M. Leduc, T. Degueule, B. Combemale, T. Van Der Storm, O. Barais, Revisiting Visitors for Modular Extension of Executable DSMLs, in: 2017 ACM/IEEE 20th International Conference on Model-Driven Engineering Languages and Systems (MODELS), 2017, pp. 112–122.
[33] E. Barrett, C. F. Bolz, L. Diekmann, L. Tratt, Fine-grained Language Composition: A Case Study, in: Proceddings of the 30th European Conference on Object-Oriented Programming (ECOOP'16), 2016.
[34] M. Voelter, K. Solomatov, Language modularization and composition with projectional language workbenches illustrated with mps, Software Language Engineering, SLE 16 (2010) 3.
[35] K. Pohl, G. Böckle, F. J. van Der Linden, Software product line engineering: foundations, principles and techniques, Springer Science & Business Media, 2005.
[36] T. Degueule, B. Combemale, J.-M. Jézéquel, On Language Interfaces, Springer International Publishing, 2017, pp. 65–75. doi:10.1007/978-3-319-67425-4_5.
[37] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, M. Hanus, Xbase: implementing domain-specific languages for java, in: Generative Programming and Component Engineering, GPCE'12, Dresden, Germany, September 26-28, 2012, 2012, pp. 112–121. doi:10.1145/2371401.2371419.
[38] T. Mayerhofer, M. Wimmer, A. Vallecillo, Adding uncertainty and units to quantity types in software models, in: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, ACM, 2016, pp. 118–131.
[39] D. L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, Communications of the ACM 15 (12) (1972) 1053–1058. doi:10.1145/361598.361623.
[40] D. L. Parnas, A Technique for Software Module Specification with Examples, Communications of the ACM 15 (5) (1972) 330–336. doi:10.1145/355602.361309.
[41] K. Czarnecki, S. Helsen, U. Eisenecker, Staged Configuration Using Feature Models, in: D. Weiss, R. van Ommering (Eds.), Proceedings of the 3rd International Conference on Software Product-Line (SPLC'04), Lecture Notes in Computer Science 3154, Springer, Boston, MA, USA, 2004, pp. 266–283.
[42] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, B. Combemale, Execution framework of the gemoc studio (tool demo), in: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2016, ACM, 2016, pp. 84–89.
[43] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: Eclipse Modeling Framework, Pearson Education, 2008.
[44] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, F. Fouquet, Mashup of metalanguages and its implementation in the kermeta language workbench, Software & Systems Modeling 14 (2) (2015) 905–920. doi:10.1007/s10270-013-0354-4.
[45] J. Deantoni, I. P. Diallo, C. Teodorov, J. Champeau, B. Combemale, Towards a Meta-language for the Concurrency Concern in DSLs, in: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE '15, EDA Consortium, San Jose, CA, USA, 2015, pp. 313–316.
[46] B. Combemale, J. De Antoni, M. V. Larsen, F. Mallet, O. Barais, B. Baudry, R. B. France, Reifying Concurrency for Executable Metamodeling, Springer International Publishing, Cham, 2013, pp. 365–384. doi:10.1007/978-3-319-02654-1_20.
[47] T. Mayerhofer, P. Langer, M. Wimmer, G. Kappel, xmof: Executable dsmls based on fuml, in: International Conference on Software Language Engineering, Springer, 2013, pp. 56–75.
[48] M. E. V. Larsen, J. DeAntoni, B. Combemale, F. Mallet, A behavioral coordination operator language (bcool), in: 2015 ACM/IEEE 18th International Conference on Model-Driven Engineering Languages and Systems (MODELS), 2015, pp. 186–195. doi:10.1109/MODELS.2015.7338249.
[49] D. Méndez-Acuña, J. A. G. Galindo, B. Combemale, A. Blouin, B. Baudry, Reverse engineering

language product lines from existing DSL variants, Journal of Systems and Software 133 (Supplement C) (2017) 145 – 158. doi:https://doi.org/10.1016/j.jss.2017.05.042.

[50] D. Méndez-Acuña, J. A. Galindo, B. Combemale, A. Blouin, B. Baudry, Springer International Publishing, Cham, 2016, pp. 393–396. doi:10.1007/978-3-319-35122-3_26.

[51] A. Haber, M. Look, P. Mir Seyed Nazari, A. Navarro Perez, B. Rumpe, S. Völkel, A. Wortmann, Composition of Heterogeneous Modeling Languages, Model-Driven Engineering and Software Development Conference (MODELSWARD'15) 580 (2015) 45–66.

[52] K. Hölldobler, B. Rumpe, I. Weisemöller, Systematically Deriving Domain-Specific Transformation Languages, in: 2015 ACM/IEEE 18th International Conference on Model-Driven Engineering Languages and Systems (MODELS), ACM/IEEE, 2015, pp. 136–145.

[53] J. O. Ringert, A. Roth, B. Rumpe, A. Wortmann, Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems, Journal of Software Engineering for Robotics (JOSER) 6 (1) (2015) 33–57.

[54] W. Cazzola, E. Vacchi, On the Incremental Growth and Shrinkage of LR Goto-Graphs, ACTA Informatica 51 (7) (2014) 419–447. doi:10.1007/s00236-014-0201-2.

[55] F. Fleurey, A. Solberg, A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems, in: A. Schürr, B. Selic (Eds.), 2009 ACM/IEEE 12th International Conference on Model-Driven Engineering Languages and Systems (MODELS), ACM, Denver, CO, USA, 2009, pp. 606–621.

[56] W. Cazzola, A. Shaqiri, Open Programming Language Interpreters, The Art, Science, and Engineering of Programming Journal 1 (2) (2017) 5–1–5–34. doi:10.22152/programming-journal.org/2017/1/5.

[57] W. Cazzola, P. Giannini, A. Shaqiri, Formal Attributes Traceability in Modular Language Development Frameworks, in: L. Aceto, S. Micali (Eds.), Proceedings of the 16th Italian Conference on Theoretical Computer Science (ICTCS'15), Firenze, Italy, 2015.

[58] J. B. Martin, Principles of object-oriented analysis and design, notThenot James Martin Books, Prentice Hall, 1993.

[59] E. Gamma, Design patterns: elements of reusable object-oriented software, Pearson Education India, 1995.

[60] W. Cazzola, D. M. Olivares, Gradually Learning Programming Supported by a Growable Programming Language, IEEE Transactions on Emerging Topics in Computing 4 (3) (2016) 404–415, special Issue on Emerging Trends in Education. doi:10.1109/TETC.2015.2446192.

[61] W. Cazzola, D. Poletti, DSL Evolution through Composition, in: Proceedings of the 7$^{\text{th}}$ ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'10), ACM, Maribor, Slovenia, 2010.

[62] E. Vacchi, W. Cazzola, B. Combemale, M. Acher, Automating Variability Model Inference for Component-Based Language Implementations, in: P. Heymans, J. Rubin (Eds.), Proceedings of the 18th International Software Product Line Conference (SPLC'14), ACM, Florence, Italy, 2014, pp. 167–176.

[63] G. L. Steele, Jr., Growing a Language, Higher-Order and Symbolic Computation 12 (3) (1999) 221–236. doi:10.1023/A:1010085415024.

[64] S. Erdweg, P. G. Giarrusso, T. Rendel, Language composition untangled, in: International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012, 2012, p. 7. doi:10.1145/2427048.2427055.

[65] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, J. C. S. do Prado Leite, Reverse Engineering Goal Models from Legacy Code, in: 13th IEEE International Conference on Requirements Engineering (RE 2005), 29 August - 2 September 2005, Paris, France, 2005, pp. 363–372. doi:10.1109/RE.2005.61.

[66] B. R. Bryant, J. Jézéquel, R. Lämmel, M. Mernik, M. Schindler, F. Steinmann, J. Tolvanen, A. Vallecillo, M. Vöelter, Globalized domain specific language engineering, in: Globalizing Domain-Specific Languages - International Dagstuhl Seminar Dagstuhl Castle, Germany, October 5-10, 2014 Revised Papers, 2014, pp. 43–69. doi:10.1007/978-3-319-26172-0_4.

[67] T. Clark, M. van den Brand, B. Combemale, B. Rumpe, Conceptual model of the globalization for domain-specific languages, in: Globalizing Domain-Specific Languages - International Dagstuhl Seminar Dagstuhl Castle, Germany, October 5-10, 2014 Revised Papers, 2014, pp. 7–20. doi:10.1007/978-3-319-26172-0_2.