When Systems Engineering Meets Software Language Engineering

Jean-Marc Jézéquel, David Mendez, Thomas Degueule, Benoit Combemale, Olivier Barais

When Systems Engineering Meets Software Language Engineering

Jean-Marc Jézéquel[1], David Méndez-Acuña[1], Thomas Degueule[1], Benoit Combemale[1,2], and Olivier Barais[1]

Abstract. The engineering of systems involves many different stakeholders, each with their own domain of expertise. Hence more and more organizations are adopting Domain Specific Languages (DSLs) to allow domain experts to express solutions directly in terms of relevant domain concepts. This new trend raises new challenges about designing DSLs, evolving a set of DSLs and coordinating the use of multiple DSLs for both DSL designers and DSL users. This paper explores various dimensions of these challenges, and outlines a possible research roadmap for addressing them. The message of this paper is also to claim that if language engineering techniques to design any single (disposable) language are mature, the language engineering community needs to fundamentally change its view on software language design. We need to take the next step and adopt the perspective that a software language is, fundamentally, software too and thus the result of a composition of design decisions. These design decisions should be represented as first-class entities in the software languages workbench and it should be possible, during the language lifecycle, to add, remove and change language design decisions with limited effort to go from continuous design to continuous meta-design.

1 A Language-Oriented Vision for Systems Engineering

The engineering of complex software intensive systems involves many different stakeholders, each with their own domain of expertise. It is particularly true in the context of systems engineering in which rather than having everybody working with code/model defined in general-purpose (modeling/programming) languages, more and more organizations are turning to the use of Domain Specific Languages (DSLs). DSLs allow domain experts to express solutions directly in terms of relevant domain concepts, and use generative mechanisms to transform DSL specifications into software artifacts (e.g., code, configuration files or documentation), thus abstracting away from the complexity of the rest of the system and the intricacies of its implementation.

The adoption of DSLs has major consequences on the industrial development processes. This approach, a.k.a. Language-Oriented Programming [19], breakdowns the development process into two complementary stages (see Figure 1): the development, adaptation or evolution by *language designers* of one or several DSLs, each capitalizing the knowledge of a given domain, and the use of such

[1] IRISA, University of Rennes 1, France
[2] Inria, France

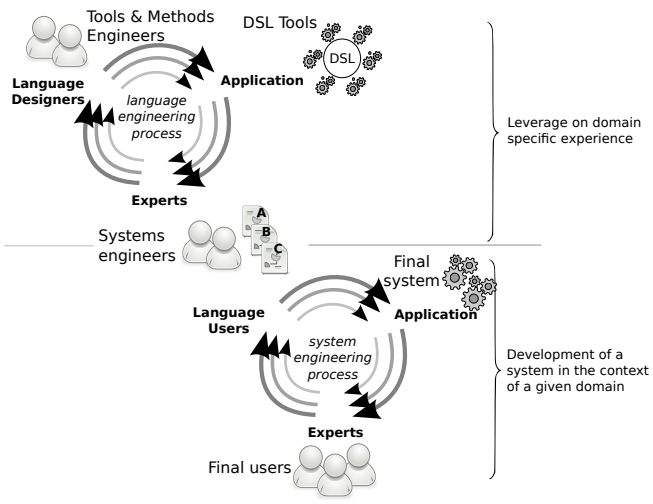

Fig. 1: Language engineering stakeholders

DSLs by *language users* to develop the different system concerns. Each stage has specific objectives and requires special skills. Figure 1 depicts the two interdependent processes that continuously drive each other's. The main objective of the language engineering process is to produce a DSL which tackles a specific concern encountered by engineers in the development of a complex system, together with its tooling. Once an appropriate DSL is made available to systems engineers, it is used to express the solution to this specific concern in the final system. However, by definition, DSLs are bounded to evolve with the domain they abstract. Consequently, systems engineers need to be well aware of end users' expectations in order to report their new requirements to the language designers. A new evolved DSL is then produced by the language designers, which is in turn used by systems engineers and so on and so forth. It is worthwhile to note that, although this is unlikely in large companies, these roles can be alternatively played by the same people in smaller organizations.

As a matter of fact, while DSLs have been found useful for structuring development processes and providing abstractions to stakeholders [10], their ultimate value has been severely limited by their user-understanding ambiguity, the cost of tooling and the tendency to create rigidity, immobility and paralysis (the evolution of such languages is costly and error-prone). The development of software languages is a challenging task also due to the specialized knowledge it requires. A language designer must own not only quite solid modeling skills but also the technical expertise for conducting the definition of specific artifacts such as grammars, metamodels, compilers, and interpreters. *"Software languages are software too"* [7] and, consequently, languages development inherits all the complexity of general software development; concerns such as maintainability, reusability, evolution, user experience are recurring requirements in the daily work of software language engineers. As a result, there is room for application of software engineering techniques that facilitate the DSL construction process. This

fact permitted the emergence of what we know as *Software Language Engineering* that is defined as the application of systematic, disciplined, and measurable approaches to the development, use, deployment, and maintenance of software languages [11].

The message of this paper is twofold. First, we claim that language engineering techniques for designing disposable DSLs are close to maturity. However, as we will see, some challenges such as composition, modularity or evolution still need to be addressed. Hopefully, decades of research in software engineering already paved the way and software language engineering should leverage these facilities in order to tackle these challenges. Second, we claim that the common view on software language design should fundamentally evolve. Rather than abstract syntax trees, metamodels, type checkers, parsers, code generators, compilers, etc., we need to model and represent a software language as the composition of a set of language design decisions, concerning, among others, the existing language-units solutions, variation points, features and usage scenarios that are needed to satisfy the requirements. Once we are able to represent software languages, in several phases of the lifecycle, in terms of the aforementioned concepts, changing and evolving software languages will be considerably simplified.

The remainder of this paper is organized as follows. We investigate the underlying challenges of the adoption of DSLs in the development of complex software intensive systems firstly from the points of view of the language designer (Section 2) and secondly from the language user point of view (Section 3). Finally, Section 4 draws some perspectives and concludes.

2 Challenges for SLE from Language Designers' Point of View

From a language designer point of view, the development of new DSLs as well as the evolution of existing ones becomes daily activities. Evolving a DSL usually requires the co-evolution of all its tooling (parsers, textual syntax and graphical syntax editors, compilers, code generators, ...). Besides, language users generally require backward compatibility or tooling for supporting the migration. Consequently, each evolution is costly and error-prone and the software language engineering community still needs to come up with new solutions. To enable this vision that the language design decisions should be represented as first-class entities in the software languages workbench and the it should, during the language lifecycle, be possible to add, remove and change language design decisions against limited effort, his section explores some required software engineering techniques that have been used in the context of software languages engineering for improving i) the reuse, ii) the variability management and iii) the verification and validation. Specifically, we highlight the main challenges that remain to be addressed in each case.

2.1 Reuse

Reusability of software artifacts is a central notion that has been thoroughly studied and used by both academics and industrials since the early days of soft-

ware construction. Essentially, designing reusable artifacts allows the construction of large systems from smaller parts that have been separately developed and validated, thus reducing the development costs by capitalizing on previous engineering efforts.

It is however still hardly possible for language designers to design typical language artifacts (e.g. language constructs, grammars, editors or compilers) in a reusable way. The current state of the practice most of the time prevents the reusability of language artifacts from one language to another, or from one system to another, consequently hindering the emergence of real engineering techniques around software languages.

Conversely, concepts and mechanisms that enable artifacts reusability abound in the software engineering community. In this section, we present the time-honored concepts of substitutability, inheritance and components, show their relevance for language designers and draw some perspectives and challenges for their inclusion in software language engineering.

Substitutability In its broadest sense, substitutability is the mechanism that allows the replacement of one software artifact (e.g. code, object, module) with another one under certain conditions. In the context of software language engineering, the considered artifacts (languages, models, abstractions, tools, etc.) are all candidates for substitutability mechanisms, allowing reusing them in different contexts. We propose the notion of types as interfaces that express the constraints that different artifacts must verify in order to be substituted one another.

The substitution principle has been thoroughly investigated in the object-oriented programming community. It states whether given objects in a program can safely be substituted to other objects based on the subtyping relation that stands between their types [14]. Most object-oriented programming languages feature a mechanism of subtype polymorphism that allows considering the same object through different interfaces (i.e. types), provided they are subtypes one another, enabling facilities such as code reuse or dynamic binding.

In the context of software language engineering, the definition of such types and subtyping relations enables model (i.e. graph of objects) polymorphism, namely the possibility to manipulate a model or program created using a specific language through different tools initially designed for similar yet different languages [16]. Model polymorphism allows tackling a wide range of scenarios that are commonly faced by system engineers. As a concrete example, consider the management of evolution on complex languages such as UML. It is difficult for engineers to deal with this evolution, as all efforts concentrated around a language are lost with subsequent versions; e.g. a transformation defined for UML2.1 cannot be reused for models created using UML2.2 since these are, although semantically close, different languages. Specifying the parameter of such a transformation in terms of model interface allows reusing it for any model that matches this interface: if a subtyping relation can be established between

the two versions, model polymorphism allows the reuse of all the tooling across them, even benefiting from dynamic binding for prospective specialization.

However, the currently prevalent modeling frameworks do not provide such type of substitutability mechanisms, and only a few recent research works address them (e.g. [5,9,17]). Challenges such as complete language semantics substitutability or concrete syntax replacement still need to be addressed. More generally, using interfaces for specifying the expected features and properties of a language paves the way for language-agnostic, generic manipulation of models and programs. This is particularly relevant in system engineering as engineers need to deal with many different domains and stakeholders, each using his own domain-specific, independently-evolving language.

Extension The need for language extension arises in many scenarios. DSLs are initially designed and implemented for a restricted set of users with a finite set of features that support their requirements but, most of the time, new requirements will emerge once the language gets effectively used: they tend to grow with the users' needs. Moreover, DSLs are now more and more scattered among different set of users that tackles the same domain, but with their own specificities. In this case, language designers should be able to reuse an existing DSL that contains the basic constructs and features for a particular domain, and extend it with their own business distinctive features.

To support such scenarios where extensions are most of the time unforeseen, DSLs must be designed in a way that facilitates their reuse and extensibility. Conversely, a language designer that extends an existing language should be able to concentrate on its business specificities, seamlessly reusing the base language along with all its tooling. In this regard, real extensibility mechanisms should support the introduction of new constructs, abstractions, or tools without having to understand or recompile the base language source code.

Modularity and composability Modularization of software (*i.e.*, components-based software development) is considered an effective mechanism for achieving software reuse and, consequently, reducing costs and time to market. The main principle is to structure software applications as sets of interconnected building blocks that, in turn, are designed in such a way that allows later re-use in other applications. In this context, three of the most important challenges are (1) design and implement components with real potential re-use; (2) design the interfaces that enable crosscutting collaboration among components in a system; and (3) provide components models [12] that offer composition mechanisms for integrating a set of components.

All the aforementioned ideas apply also when the software under construction is a software language; there are benefits in terms of the reduction of construction effort [2]. Nevertheless, those general challenges gain some special connotations analyzed below:

i) Components design (how to breakdown a language?): Decomposing a language in several language modules (a.k.a., language units) is not only about

offering a languages benchmark that enables modular definition of metamodels, grammars and semantics; it is just the technical part. The other important challenge is to understand how the language units should be defined so they can be reused in other contexts. What is the correct level of granularity? What are the *"services"* that a language unit should offer for being considered reusable? What is the meaning of a *"service"* in the context of software languages? What is the meaning of a *"services composition"* in the context of software languages?

ii)Languages interfaces (how language units are specified?): The construction of a language unit is not only about implementing a subset of the language but also about specifying its *boundary* (i.e., the set of services it offers to other language units and the set of services it requires from other language units). This fact refers to the classical idea of required and provided interfaces introduced by components-based software engineering approaches. But... what is the meaning of *"provided and required services"* in the context of software languages? We argue that the answer to that question must consider at least two facts: composability and (one more time) substitutability. In the case of composability, required and provided interfaces should provide a mechanism for exposing providing services so they can be consumed by the required services of other language units. In other words, interfaces are a mechanism for interaction between language units. By the other hand, substitutability refers to the possibility of implementing provided services in different language units so the provided interface becomes also a set of constraints that ensure the safe replacement of the given implementations.

iii) Language units composition (how two language units do interact?): The nature of the interaction between two language units might be different depending on some architectural decisions; extending one language unit with another one is a different situation from a required service of a language unit consuming one or several provided services from other language units. In the case of extension, the base language unit is usually independent of the extensions whereas the extensions have little sense without the base language unit [6]. In the case of required and provided interactions, the requiring language unit usually cannot work without the provided one. We argue that there is a need of composition operators that explicitly define the role of each language unit in a composition.

2.2 Variability management and language families

One of the main limitations of components-based software development –and of course it is also true in the case of software languages modularization– is the difficulty of designing components that can be actually re-used in other contexts. Despite the design principles and patterns, reusing of software components is not guaranteed. In fact, in many cases the effort of building modularized software is not compensated with the re-usability opportunities.

One of the answers that the software community has found is the idea of variability management. Variability management is a mechanism for explicitly

representing the commonalities and differences among a family of software products. A family of products is defined as a set of software applications that have similar purposes and that share some functionality but that is specialized in a particular type of users or situation. The idea is to effectively reuse the implementation of such common functionality and having a repository of "common assets" that implement product features. The process of creating a product by using the family of products is called product derivation. To do so, it is necessary to select the desired product features and to offer a mechanism of composition for integrating the assets corresponding to each feature. This is the main principle of what we know as *Software Product Line Engineering* (SPLE) that is a software engineering approach that has demonstrated important benefits in the general case of software development.

As demonstrated in [18], variability management –and the ideas behind SPLC in general– can be applied in the context of software languages for increasing the re-usability and then increasing the productivity of software language engineers. In this context, a family of products actually is a family of languages where there are some commonalities and some differences (consider as an example the family of OCL variants presented in [20]).

Some of the challenges that should be considered are:

Alignment with the modularization approach: It is worth noting that modularization is a prerequisite for addressing variability management. In fact, at the implementation level software modularization and variability management are strongly linked. Each concrete feature expressed in the variability model must correspond to a software component in the architecture so a given configuration can be derived in a concrete functional product. In the case of software languages each feature should be mapped to one (or more) language units that offers the corresponding services. Moreover, in [13] van der Linden *et. al.* present a set of three variability realization techniques at the level of the software modularization schema. Those techniques can be viewed as a set of requirements in terms of modularization and composition of the architecture and they are quite related with the concepts of extension, substitutability and adaptation, some of them discussed in the previous section. How to conjugate all those concepts for effectively define an approach that allows the construction of families of software languages?

Multi-stage orthogonal variability modeling: Typically, a software language specification is intended to define the abstract syntax, the concrete syntax and the semantics of a language. As a result, language units have to contribute to each of those dimensions. In other words, each language unit specification includes a partial definition of the abstract syntax, the concrete syntax, and the semantics. The whole language specification is obtained by putting all the language units together. In [8] the authors observed that there exists some variability between each of those dimensions. Thereby, one language construct (i.e., a concept in the abstract syntax) may be represented in several ways (i.e., several possible concrete syntaxes) and/or may have different meanings (several possible semantics). This analysis remains the same for both the whole language

specification and each segment defined in language units. Consequently, we have at least three different dimensions of variability each of them regarding one field of the tuple:

- **Abstract syntax variability or "functional variability":** This variability refers to the capability of selecting the desired language constructs for a particular product as long as the dependencies are respected. Consider for example a family of languages for state machines where concepts such as timed transitions, composite states, or history pseudo-states are optional and are only included if the user of the language needs them. This variability dimension is quite similar to the classical concept of functional variability of SPLC where each feature represents a piece of functionality that may be or not included depending on the specific requirements of a user.
- **Concrete syntax variability or "representation variability":** This variability refers to the capability of offering different representations for the same concept. Consider for example a language for state machines that can have textual or graphical representations. Note that this type of variability is especially relevant in the context of metamorphics DSLs that we will explain later in this paper.
- **Semantics variability or "interpretation variability":** This variability refers to capability of offering the different interpretations to the same concept. Consider for example the semantics differences that exist between state machines languages explored in [4]. In that work, we can see how, for example, the priorities between conflicting transitions in a state machine are resolved with different criteria. If we are able to manage such variability, the reuse opportunities are drastically increased since we are able to reuse the entire language infrastructure (e.g., editors, abstract syntax trees) for the implementation of different languages that are interpreted according to the needs of specific users.

Note that both representation variability and interpretation variability depend on the functional variability. It makes no sense to select a representation (or interpretation) for a language construct that has not been included as part of the language product. In other words, the configuration of representation and interpretation must be performed only for the construct selected in the functional variability resolution.

2.3 Verification & Validation

Just as any complex software artifact, software languages need to be thoroughly verified and validated. Their complex nature, the different aspects that compose them makes it particularly difficult: is a language really suited for the problems it tries to tackle? Can all programs relevant for a specific domain be expressed in a precise and concise manner? Are all valid programs correctly handled by the interpreter? Does the compiler always generate valid code?

Different techniques have been developed for the V&V of traditional software and are good candidates for adaptation to software languages: among them, we

focus on design-by-contract and software testing, and the challenges they need to address for the engineering of software languages.

Design-by-contract [15] advocates the definition of precise interfaces for software components, e.g. using preconditions, postconditions, invariants or types. Contracts can then be checked at different levels to assess the correct interaction of components. In the context of software languages, contracts may be defined on the abstract syntax (e.g. using invariants), on the semantics (e.g. using preconditions and postconditions), etc [17]. Design-by-contract is especially relevant for system engineering as it raises the level of abstraction in which the interaction between the different domains and languages is considered, and makes explicit some of the original requirements on the language. An integrated design-by-contract process for software languages engineering is expected to bring the same benefits as in traditional software development: precise – structural and behavioral – interfaces, improved error handling, specification-driven definition of artifacts, etc.

Software testing, on the other side, is the most prevalent V&V technique in software engineering. Testing software languages is a challenging activity since all their aspects must be checked: abstract syntax, grammar, semantics, tooling, etc. Furthermore, in this context, test data (i.e. models or programs) are themselves complex artifacts, thus complicating the coverage of representative inputs and the definition of oracles functions [1]. The extensive use of generative programming techniques also raises additional problems due to the gap between generation time and testing time; i.e. the to-be-tested generated artifacts are not known yet when the tests are written. Workarounds on this issue include the automatic generation of test cases together with generated artifacts, which in turns increases the testing activity complexity. Finally, the inherent nature of multi-languages engineering requires not only the different languages to be tested, but also their combination and interaction. Such integration tests should be dedicated to the verification and validation of the composition, reusing the testing effort spent on each of its part.

3 Challenges for SLE From the Language Users' Point of View

From the perspective of the users the emergence of several software languages is also challenging. Despite the overall purpose of constructing DSLs is to facilitate the daily work of systems engineers, dealing with several languages implies not only learning new syntaxes but also interacting with an increasing number of tools: editors, compilers, and code generators among others. The remainder of this section is dedicated to explore some of those challenges that must be addressed to serve this vision that the language design decisions must be represented as first-class entities in the software languages workbench.

3.1 Language viewpoint

Domain-Specific Languages (DSLs) are plain languages, in the sense that many difficult design decisions must be taken during their development and mainte-

nance, and that they can take different shapes: plain-old to more fluent APIs; internal or embedded DSLs written inside an existing host language; external DSLs with their own syntax and domain-specific tooling. All forms of DSLs have strengths and weaknesses – whether you are a developer or a user of a DSL. The basic trade-offs between internal and external DSLs have already been identified and are subject to extensive discussions and research for several years. A new trend though is observed. DSLs are now so widespread that very different users with separate roles and varied objectives use them. Depending on the kinds of users, roles or objectives, the same form of DSL (external or internal) might not be the best for everybody. Beyond the unification of the different approaches, it is worthwhile for DSLs to support the ability to be self-adaptable to the most appropriate form (including the corresponding IDE) according to a particular usage or task: we call such a DSL a *metamorphic DSL*.

From the same language description and different interface specifications, we envision the ability to derive various IDEs that can be used accordingly. This vision raises many challenges: systematic methods to evaluate when a form of a DSL meets the expected properties (e.g., learnability); artefact modularization; information sharing, while being able to visualize and manipulate an artefact in a particular representation and in a particular IDE; global mechanism to ensure consistency of the artefacts between these heterogeneous IDEs.

3.2 Language evolution

By definition, DSLs are bbounded to evolve with the domain they abstract. Consequently, DSLs users need to learn and understand the newly-created abstractions, syntaxes and tools. This raises new challenges in terms of change management and learnability of languages. In order to facilitate the transition, the migration of models from one version of a language to another, aka co-evolution, must be fully supported by the workbench: automatic migration when possible, *diff* computation with explicit user refinement when required, etc. The same situation arises when an older language is replaced with a completely new one, defined independently, but abstracting the same domain.

3.3 Language integration

The development of modern complex software-intensive systems often involves the use of multiple DSLs that capture different system aspects. In addition, models of the system aspects are seldom manipulated independently of each other. System engineers are thus faced with the difficult task of relating information presented in different models. For example, a system engineer may need to analyze a system property that requires information scattered in models expressed in different DSLs. Current DSL development workbenches provide good support for developing independent DSMLs, but provide little or no support for integrated use of multiple DSLs. The lack of support for explicitly relating concepts expressed in different DSMLs makes it very difficult for developers to reason about information spread across different models.

Supporting coordinated use of DSLs leads to what we call the globalization of modeling languages [3], that is, the use of multiple modeling languages to support coordinated development of diverse aspects of a system. The term "globalization" is used to highlight the desire that DSLs developed in an independent manner to meet the specific needs of domain experts, should also have an associated framework that regulates interactions needed to support collaboration and work coordination across different system domains.

Globalized DSLs aim to support the following critical aspects of developing complex systems: communication across teams working on different aspects, coordination of work across the teams, and control of the teams to ensure product quality.

4 Conclusion and Perspectives

This paper claims that research conducted in SLE for systems engineering should consider that:

- The first phase of research and development in SLE has matured the technology to a level where industry adoption is wide-spread and few fundamental issues remain for efficiently designing any single (disposable) DSL.
- The traditional view on SLE suffers from a number of key problems that cannot be solved without changing our perspective on the notion of language, and especially of DSL. These problems include i) the lack of first-class representation of design decisions in DSL: since design decisions are cross-cutting and intertwined, they are easy to forget and hard to change, leading to high maintenance costs; ii) the lack of support for explicitly relating different DSLs that makes it very difficult for systems engineers to use multiple DSLs while enabling a coordinated development of the diverse system aspects, and to reason about information spread across artifacts built with different DSLs.
- As a community, we need to take the next step and adopt the perspective that a software language is, fundamentally, software too, that is, the result of a composition of design decisions. These design decisions should be represented as first-class entities in the software language workbench and it should, during the language lifecycle, be possible to add, remove and change language design decisions with limited effort to go from continuous design to continuous meta-design.

5 Acknowledgments

This work is partially supported by the ANR INS Project GEMOC (ANR-12-INSE-0011), the bilateral collaboration between INRIA and Thales Research & Technology VaryMDE, and the ITEA2 European project MERgE.

References

1. B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J-M. Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 53(6):139–143, 2010.
2. T. Cleenewerck. Component-based dsl development. In *Generative Programming and Component Engineering*, volume 2830 of *LNCS*, pages 245–264. Springer, 2003.
3. B. Combemale, J. DeAntoni, B. Baudry, R. B. France, J-M. Jezequel, and J. Gray. Globalizing modeling languages. *Computer*, 47(6):68–71, 2014.
4. M. Crane and J. Dingel. Uml vs. classical vs. rhapsody statecharts: not all models are created equal. *Software & Systems Modeling*, 6(4):415–435, 2007.
5. J. De Lara and E. Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *Model Driven Engineering Languages and Systems*, pages 16–30. Springer, 2010.
6. S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, pages 7:1–7:8. ACM, 2012.
7. J-M. Favre, D. Gasevic, R. Lämmel, and E. Pek. Empirical language analysis in software linguistics. In *Software Language Engineering*, volume 6563 of *LNCS*, pages 316–326. Springer, 2011.
8. H. Grönniger and B. Rumpe. Modeling language variability. In *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, volume 6662 of *LNCS*, pages 17–32. Springer, 2011.
9. C. Guy, B. Combemale, S. Derrien, J. Steel, and J-M. Jézéquel. On model subtyping. In *Modelling Foundations and Applications*, pages 400–415. Springer, 2012.
10. J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480. ACM, 2011.
11. A. Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1 edition, 2008.
12. K-K. Lau and Z. Wang. Software component models. *Transactions on Software Engineering*, 33(10):709–724, Oct 2007.
13. F. J. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.
14. B. H Liskov and J. M Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.
15. B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.
16. J. Steel and J-M. Jézéquel. On model typing. *Software & Systems Modeling*, 6(4):401–413, 2007.
17. W. Sun, B. Combemale, S. Derrien, and R. France. Using model types to support contract-aware model substitutability. In *Modelling Foundations and Applications*, pages 118–133. Springer, 2013.
18. E. Vacchi, W. Cazzola, S. Pillay, and B. Combemale. Variability support in domain-specific language development. In *Software Language Engineering*, volume 8225 of *LNCS*, pages 76–95. Springer, 2013.
19. M. P Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.
20. C. Wende, N. Thieme, and S. Zschaler. A role-based approach towards modular language engineering. In *Software Language Engineering*, volume 5969 of *LNCS*, pages 254–273. Springer, 2010.