On Language Interfaces

Thomas Degueule, Benoit Combemale, Jean-Marc Jézéquel

On Language Interfaces

Thomas Degueule, Benoit Combemale, and Jean-Marc Jézéquel

Abstract Complex systems are developed by teams of experts from multiple domains, who can be liberated from becoming programming experts through domain-specific languages (DSLs). The implementation of the different concerns of DSLs (including syntaxes and semantics) is now well-established and supported by various languages workbenches. However, the various services associated to a DSL (e.g., editors, model checker, debugger or composition operators) are still directly based on its implementation. Moreover, while most of the services crosscut the different DSL concerns, they only require specific information on each. Consequently, this prevents the reuse of services among related DSLs, and increases the complexity of service implementation. Leveraging the time-honored concept of *interface* in software engineering, we discuss the benefits of *language interfaces* in the context of software language engineering. In particular, we elaborate on particular usages that address current challenges in language development.

1 Introduction

As far back as 1972, Edsger W. Dijkstra said in his ACM Turing Lecture:

> *Another lesson we should have learned from the recent past is that the development of 'richer' or 'more powerful' programming languages was a mistake in the sense that these baroque monstrosities, these conglomerations of idiosyncrasies, are really unmanageable, both mechanically and mentally.*
>
> *I see a great future for very systematic and very modest programming languages.*

Thomas Degueule
INRIA – IRISA e-mail: thomas.degueule@inria.fr

Benoit Combemale
University of Rennes 1 – IRISA e-mail: benoit.combemale@irisa.fr

Jean-Marc Jézéquel
University of Rennes 1 – IRISA e-mail: jezequel@irisa.fr

This quote is often cited by proponents of Domain Specific Languages (DSL), which are indeed modest languages specifically designed for a single purpose.

Until now however, the vision of building large systems with the help of a set of DSLs, each caring for a specific aspect of the system, did not really turn into reality. Two main reasons why it has always been hard to work with DSLs, are:

1. Weaving together the various aspects of a system expressed in different DSLs is still mostly ad hoc and artisanal [21]. However recent approaches such as the GEMOC initiative [9] have made significant progresses towards that goal.
2. A DSL needs specific tool support: editors, parsers, checkers, interpreters, compilers, analysers, refactoring tools, etc. All of this software is subject to standard software engineering issues: successive versions, simultaneous variants, quality control (with e.g. tests). While general purpose languages used by millions of people can justify a high level of investment on building these supporting tools, the return on investment is more problematic for DSL used by definition by a much smaller number of people.

Regarding the second issue, the implementation of the different concerns of DSLs (including syntaxes and semantics) is now well-established and supported by various languages workbenches. However, the implementation of the services associated to a DSL (e.g., editors, model checker, debugger and composition operators) are still directly based on its implementation. Consequently, as the implementation of a DSL evolves, all services must be updated to target the new implementation. It is also not possible to reuse services for different, yet similar DSLs that target the same domain of application (e.g. two variants of state machine DSLs developed by different companies, or two versions of the same DSL). Moreover, most services require information that is scattered in the different concerns that compose a DSL, expressed in various and usually complex formalisms. Overall, the lack of abstraction mechanisms on top of DSL implementations complexifies the definition of services and hampers their reuse.

In this paper, we reflect on the uses and benefits of interfaces in programming and software engineering (Section 2). We then study how the concept of interface can be adapted to software language engineering to improve the current practice of language development. Specifically, we show how the concept of *language interface* can help to address various challenges that arise from current language development practices (Section 3). Finally, we conclude in Section 4.

2 Interfaces in Programming and Software Engineering

The time-honored concept of interface has been studied since the early days of computer science in many areas of programming and software engineering. Despite variability in their exact realization, interfaces invariably rely on common fundamental concepts and provide similar benefits. Historically, the notion of interface is intrinsically linked to the need for *abstraction*, one of the fundamental concepts of computer science. As stated by Parnas et al, “an abstraction is a concept that can have more than one possible realization” and “by solving a problem in terms of the abstraction one can solve many problems at once” [30]. One can *refer* to an abstraction, leaving out the details of a concrete realization and the details that differ from one realization to another [23]. Originally, in programming, the key idea was to encapsulate the parts of a program that are more prone to change into so-called modules, and to design a more stable *interface* around these change-prone parts. This concept, known as *information hiding*, eliminates hard-wired dependencies between change-prone regions, thereby protecting other modules of the program from unexpected evolution [29].

As different realizations may be used in place of the same abstraction, interfaces also foster reuse: one module can substitute another one in a given context provided that they realize the same interface, as expected by a *client*. The choice of a concrete module is transparent from the point of view of the client of the interface. Because interfaces expose only a portion or an aspect of a realization, leaving out some details, the nature of an interface is highly dependent on the nature of the concrete realization it abstracts. Following the evolution of programming paradigms, authors have thus defined various kind of interfaces for various concrete realizations: modules (*module interfaces* in Modula-2 [37]), packages (*package specifications* in Ada [20]), objects (*protocols* in Smalltalk [17]), or components in Component-Based Software Engineering (CBSE) [19], to name a few.

The expressiveness in which one can specify an interface for a given kind of realization also varies: interfaces over classes in standard Java merely consists of a set of method signatures, while languages supporting design-by-contract enable the expression of behavioral specifications, e.g. in the form of pre- and post-conditions on those signatures [27]. The expressiveness of contracts themselves range from purely syntactical levels to extra-functional (e.g. quality of service) levels [2]. Interfaces are also closely linked to the notion of *data type* in programming languages [5]. Types abstract over the concrete values or objects manipulated by a program along its execution. Type systems use these types to check their compatibility, reduce the possibilities of bugs, and foster reuse and genericity through polymorphism and substitutability [6].

While in programming languages the realizations hidden behind a given kind of interface are most often homogeneous, this is usually not the case in CBSE. As an illustration, component models enable communication between components written in different programming languages and deployed on heterogeneous platforms [31]. In such a case, interfaces abstract away from implementation technologies to enable interoperability between heterogeneous environments. The associated runtime model is most of the time unaware of the functional aspect of components and uses generic interfaces to manage their life cycle (e.g. deploying or reloading a component). Most component models also provide the notion of required interface as a means to make explicit the dependencies between components and to reason about how different components interact together and must be composed.

In summary, interfaces are used in many ways and vary according to the purpose they serve. While “interfaces as types” mainly target the safe reuse and substitutability of modules and objects in different contexts and focus on functional aspects, the interfaces used in CBSE allow components to be independently developed and validated, and focus on extra-functional aspects. From these observations, we explore in the next section possible applications of the concept of interfaces at the language level, to improve the current practice of software language development.

3 Language Interfaces for Software Language Engineering

"Software languages are software too" [14] and, consequently, they inherit all the complexity of software development in terms of maintenance, evolution, user experience, etc. Not only do languages require traditional software development skills, but they also require specialized knowledge for conducting the development of complex artifacts such as grammars, metamodels, interpreters, or type systems. The need for proper tools and methods in the development of software languages recently led to the emergence of the Software Language Engineering (SLE) research field which is defined as "the application of systematic, disciplined, and measurable approaches to the development, use, deployment, and maintenance of software languages" [24]. While the notions presented in this paper are applicable to both general-purpose and domain-specific languages (DSLs), we put a particular emphasis on the specificities of DSLs, which are small languages targeted to a particular domain of application.

In the SLE community, new DSLs are usually developed using a language workbench, a "one-stop shop" for the definition of languages and their environments [34]. The notion of language workbench originates from the seminal work of Martin Fowler [15]. The main intent of language workbenches is to provide a unified environment to assist both language designers and users in, respectively, creating new DSLs and using them. Modern language workbenches typically offer a set of meta-languages that language designers use to express each of the implementation concerns of a DSL, along with tools and methods for manipulating their specifications. Examples of modern language workbenches are manifold: Xtext [13], Monticore [25], Spoofax [22] – to name just a few.

One of the current trend in SLE is to consider more and more languages as first-class entities that can be extended, composed, and manipulated as a whole. However, to the best of our knowledge, there exists no previous work dealing with the explicitation of *language interfaces*, that is, interfaces *at the language level*, explicitly separated from language implementations, that provide the appropriate abstraction to ease the manipulation of languages as first-class entities. To motivate the need for language interfaces, we explore in this section some of the current challenges faced in SLE and highlight how they match the challenges that have been addressed with the use of interfaces in programming and software engineering.

3.1 *Ingredients of a Domain-Specific Language*

To clarify what language interfaces abstract over, we must first understand what languages are made of. Domain-specific languages are typically defined by their abstract syntax, concrete syntax(es) and semantics. Various approaches may be employed to specify each of those, usually using dedicated meta-languages provided by language workbenches [34]. The abstract syntax specifies the domain concepts and their relations defined by a metamodel or a grammar – the latter also defining the concrete syntax. This choice often depends on the language designer's background and culture. Examples of meta-languages for specifying the abstract syntax of a DSL include MOF [1] and SDF [18]. The semantics of a DSL can be defined using various approaches including axiomatic semantics, denotational semantics, operational semantics, and their variants [28]. Concrete syntaxes are usually specified as a mapping from the abstract syntax to textual or graphical representations, e.g. through the definition of a projectional editor [35]. DSLs usually benefit from dedicated environments that assist language users in the creation, analysis, and management of models and programs throughout their lifetime. Typically, these environments embed dedicated services such as code and documentation generators, checkers, editors, or debuggers.

Overall, the ingredients composing a language are manifold and usually complex to manipulate. Because there is no universally accepted way of implementing a language, it is difficult to directly relate languages created using different language workbenches. Moreover, the slightest variation in the syntax or semantics of a language leads to a conceptually new implementation, and all the artifacts referring to it (e.g. services) must be updated. Leveraging the concept of information hiding, the definition of explicit language interfaces on top of language implementations can help to abstract away from the change-prone implementation and switch the focus on a more stable and purpose-oriented view of languages. As we shall see in the next sections, such interfaces would ease the definition and reuse of services, the composition of languages, and the engineering of language families.

3.2 *Easing the Definition and Reuse of Services*

The development of services (e.g. code and documentation generators, static analyses) is an essential part of the development of software languages. Defining a new service requires to gather the appropriate information, which is often scattered in the various constituents of given language (abstract syntax, concrete syntax, semantics), in different formalisms. Naturally, the information to be extracted varies according to the *purpose* of the service. Some services, such as simple static analyses, only require access to a subset of the syntax specification of a language. More complex services may require to aggregate information from different sources. Additionally, services may be defined both at the language specification level (e.g. analysis of the completeness of a formal semantics specification), and at the instance level (e.g. dead code elimination for programs written in a particular language).

Rather than searching for the right information in the various constituents of a language, one can first design a language interface that aggregates the appropriate information in an easily manipulable form for a specific purpose. Using the appropriate interfaces eases the cognitive effort and abbreviates the definition of services. Moreover, since different languages can match the same interface, the services written on an interface can be applied to all matching languages.

One recent illustration in the programming community is the work of Brown et al, who use micro-grammars to ease the definition of static analysis tools for different programming languages [4]. Micro-grammars are partial grammars that abstract over the full-blown specification of a language using *wildcards*. Wildcards are non-terminals used to ignore parts of a language implementation. The authors show that various languages (C++, Java, JavaScript, etc.) can match the same micro-grammars and that static analysis tools written on micro-grammars are easier to develop and can be reused or adapted to new languages with little effort.

The modeling community has also shown interest in such interfaces, as illustrated by the notions of model types [32] and concepts [11]. Model types and concepts are both structural interfaces that specify a set of requirements over a metamodel – the standard way to define the abstract syntax of a language in the modeling community. A service, e.g. implemented as a model transformation, can be made generic by expressing its parameter in terms of a model type or concept, and any metamodel matching the model type or concept can benefit from this service. The duality of model types and concepts highlight the fact that there exist many ways to realize a given kind of interface. Model types are akin to subtype polymorphism while concepts are closer to parametric polymorphism but, overall, they are complementary and provide similar benefits. In addition, model types have also been used as unified interfaces that aggregate information from both the abstract syntax (the metamodel concepts) and the semantics (the transition steps of the operational semantics) of a language [12].

Some generic services do not even require any information on the syntax or semantics of a language. A generic debugger, for instance, only requires the ability to start, pause, or inspect the execution of a model or program [3]. This common set of operations can be captured in a generic interface, and any language implementing it can benefit from debugging facilities.

3.3 Facing the Multiplication of DSLs

In the last decade, the development of Model-Driven Engineering (MDE) and the advances in language workbenches have strengthen the proliferation of domain-specific languages (DSLs). MDE advocates the use of DSLs to address each concern separately in the development of complex systems with appropriate abstractions and tools [16]. As a result, the development of modern software-intensive systems often involves the use of multiple models expressed in different DSLs to capture different system aspects [9]. This trend is very similar to what was proposed by Ward in his early work on language-oriented programming [36]. Even in traditional software development, multiple languages are often used to describe different aspects of the system of interest. Java projects, for instance, typically consist of Java source files, XML files describing the structure of modules and the deployment scenarios, Gradle build files expressed in Groovy, scripts, etc.

When multiple languages are used, the need for relating “sentences” that describe the same underlying system in different languages arises. In this context, models are seldom manipulated independently: checking a given property on a system, for instance, requires to gather information that is scattered in various models written by various stakeholders in various languages. In addition, the set of languages used to describe a given system is likely to change over time. A new, more expressive language can replace an existing one. New languages may be added, merged or split. While language workbenches support for engineering isolated languages is becoming more and more mature, there is still little support for relating concepts expressed in different DSLs together. The necessity of coordinated use of languages used in the development of a given system has recently been recognized [9, 7]. One promising approach is to leverage language interfaces to expose the appropriate information that allows to relate concepts from one language to the other [8]. The type systems of two languages, for instance, may be related through the appropriate interface that would expose their respective type definitions to allow their integration. Overall, the challenges that must be tackled are very similar to the ones that were faced a few decades ago with the use of modules in software development.

The coordinated use of DSLs engineered with different language workbenches is even more challenging. Indeed, there is no abstraction mechanisms, at the language level, that allows to abstract from the concrete implementation techniques of a language workbench, e.g. a particular meta-language for defining the abstract syntax or a particular implementation technique for the semantics. The use of interfaces and connectors has been thoroughly studied in the context of component-based software engineering to alleviate the problem of technological space compatibility. Explicitly separating the implementation of a language from its interface can help to break the barriers between language workbenches if a common agreement on technology-agnostic interfaces is found.

3.4 Enabling Language Composition

Good practices from the component-based software engineering community can also be relevant in the context of software language engineering. The idea of building reusable and independently-validated language components to ease the definition of new languages has already been studied by several authors. A language component, such as a simple action language or a for-loop construct, can be defined and thoroughly validated independently and then reused as such in other languages that encompass the expression of actions or for-loops. In the same way languages are defined, the definition of such component encompasses both the definition of its abstract syntax, concrete syntax, and semantics. This leads to what is known as compositional language engineering: the ability to design new languages as assemblies of existing language components, thus lowering the development costs [25]. The actual realization of the composition may be done statically, i.e. the different modules are merged together to produce a new language specification with its associated implementation, or dynamically, i.e. the modules are kept separated but communicate to provide the composed behavior.

In this context, language interfaces serve as a support for the definition of provided and required interfaces of each component that are later used to reason about the composition of several language components and the correctness of the assembly. Interfaces are used to detect whether the constructs of different language components are in conflict, without having to dive into their intrinsic implementation. Language interfaces are also the concrete mean for several language components to communicate with each other at runtime. For instance, when two executable languages are composed together, their interpreter must also be coordinated. Language interfaces provide the appropriate information for relating together the two interpreters, for example through a delegation pattern.

3.5 Engineering Language Families

A language family is a set of languages that share meaningful commonalities but differ on some aspects. Finite-state machine languages, for instance, are all used to model some form of computation but expose syntactic variation points (e.g. nested states, orthogonal regions) and semantic variation points (e.g. inner or outer transition priority) [10]. Altogether, these different variants form a family of finite-state machine languages. Similarly, when a language evolves, the subsequent versions form a set of variants, which raises the question of backward and forward compatibility between them. Recently, different approaches have been proposed for automating the generation of such language variants based on earlier work from the software product line engineering community [26, 33].

In the presence of a language family, language designers must be given the possibility to reuse as much as possible the environment and services (e.g. editors, generators, checkers) from one variant to the other. Naturally, the opportunities of reuse must be framed, as not all services are compatible with all variants. Language interfaces can be employed to reason about the commonalities of various language variants and the applicability of a given service or environment. Model types [32], for instance, can be used to assign a type to a language and specify the safe substitutions between different artifacts based on typing relations. The definition of those types is precisely an example of language interface. Types abstract over the details of the implementation of different variants and are used to reason about substitutability between them. In this context, language interfaces can support the definition of common abstractions that are shared by all or a subset of the members of a family, abstracting from the details that vary from one member of the family to the other.

4 Conclusion

The lack of abstraction in the manipulation of DSL implementations complexifies the definition of services (e.g. debuggers, generators, composition operators) and hampers their reuse. In this paper, we have reflected on the use of interfaces in programming and software engineering, and advocated the definition of interfaces at the language level for software language engineering. We have shown that various challenges of today's language development can be addressed through the use of language interfaces. The concept of language interface presented here is purposely abstract. We leave to future work the definition of concrete interfaces for specific purposes.

References

[1] (2004) MOF, 2.0 core final adopted specification
[2] Beugnard A, Jézéquel JM, Plouzeau N, Watkins D (1999) Making components contract aware. Computer 32(7):38–45
[3] Bousse E, Corley J, Combemale B, Gray J, Baudry B (2015) Supporting efficient and advanced omniscient debugging for xdsmls. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, ACM, pp 137–148
[4] Brown F, Nötzli A, Engler D (2016) How to build static checking systems using orders of magnitude less code. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, pp 143–157
[5] Canning PS, Cook WR, Hill WL, Olthoff WG (1989) Interfaces for strongly-typed object-oriented programming. In: ACM SigPlan Notices, ACM, vol 24, pp 457–467
[6] Cardelli L, Wegner P (1985) On understanding types, data abstraction, and polymorphism. ACM Computing Surveys (CSUR) 17(4):471–523
[7] Cheng BH, Combemale B, France RB, Jézéquel JM, Rumpe B (2015) On the globalization of domain-specific languages. In: Globalizing Domain-Specific Languages, Springer, pp 1–6
[8] Clark T, den Brand M, Combemale B, Rumpe B (2015) Conceptual Model of the Globalization for Domain-Specific Languages. In: Globalizing Domain-Specific Languages, Springer, pp 7–20
[9] Combemale B, Deantoni J, Baudry B, France RB, Jézéquel JM, Gray J (2014) Globalizing modeling languages. Computer 47(6):68–71

[10] Crane ML, Dingel J (2005) Uml vs. classical vs. rhapsody statecharts: Not all models are created equal. In: Model Driven Engineering Languages and Systems, Springer, pp 97–112
[11] Cuadrado JS, Guerra E, De Lara J (2011) Generic model transformations: write once, reuse everywhere. In: Theory and practice of model transformations, Springer, pp 62–77
[12] Degueule T, Combemale B, Blouin A, Barais O, Jézéquel JM (2015) Melange: A meta-language for modular and reusable development of dsls. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, ACM, pp 25–36
[13] Eysholdt M, Behrens H (2010) Xtext: implement your language faster than the quick and dirty way. In: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, ACM, pp 307–309
[14] Favre JM (2005) Languages evolve too! changing the software time scale. In: Eighth International Workshop on Principles of Software Evolution (IWPSE'05), IEEE, pp 33–42
[15] Fowler M (2005) Language workbenches: The killer-app for domain specific languages
[16] France R, Rumpe B (2007) Model-driven development of complex software: A research roadmap. In: 2007 Future of Software Engineering, IEEE Computer Society, pp 37–54
[17] Goldberg A, Robson D (1983) Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc.
[18] Heering J, Hendriks PRH, Klint P, Rekers J (1989) The syntax definition formalism sdf—reference manual—. ACM Sigplan Notices 24(11):43–75
[19] Heineman GT, Councill WT (2001) Component-based software engineering. Putting the pieces together, addison-westley p 5
[20] Ichbiah JD, Firth R, Hilfinger PN, Roubine O, Woodger M, Barnes JG, Abrial JR, Gailly JL, Heliard JC, Ledgard HF, et al (1983) Reference manual for the Ada programming language. Castle House Publications Limited
[21] Jézéquel JM (2008) Model Driven Design and Aspect Weaving. Journal of Software and Systems Modeling (SoSyM) 7(2):209–218, URL https://hal.inria.fr/inria-00468233
[22] Kats LC, Visser E (2010) The spoofax language workbench: rules for declarative specification of languages and IDEs, vol 45. ACM
[23] Kell S (2014) In search of types. In: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, ACM, pp 227–241
[24] Kleppe A (2008) Software language engineering: creating domain-specific languages using metamodels. Pearson Education
[25] Krahn H, Rumpe B, Völkel S (2010) Monticore: a framework for compositional development of domain specific languages. International journal on software tools for technology transfer 12(5):353–372

[26] Kühn T, Cazzola W, Olivares DM (2015) Choosy and picky: configuration of language product lines. In: Proceedings of the 19th International Conference on Software Product Line, ACM, pp 71–80
[27] Meyer B (1992) Applying'design by contract'. Computer 25(10):40–51
[28] Mosses PD (2001) The varieties of programming language semantics and their uses. In: Perspectives of System Informatics, Springer, pp 165–190
[29] Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12):1053–1058
[30] Parnas DL, Shore JE, Weiss D (1976) Abstract types defined as classes of variables. ACM SIGPLAN Notices 11(SI):149–154
[31] Siegel J (2000) CORBA 3 fundamentals and programming, vol 2. John Wiley & Sons New York, NY, USA:
[32] Steel J, Jézéquel JM (2007) On model typing. SoSyM 6(4):401–413
[33] Vacchi E, Cazzola W, Pillay S, Combemale B (2013) Variability support in domain-specific language development. In: Software Language Engineering, Springer, pp 76–95
[34] Visser E, Wachsmuth G, Tolmach A, Neron P, Vergu V, Passalaqua A, Konat G (2014) A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In: Proc. SPLASH, pp 95–111
[35] Voelter M, Kolb B, Warmer J (2014) Projecting a modular future. IEEE Software 32(5)
[36] Ward MP (1994) Language-oriented programming. Software-Concepts and Tools 15(4):147–161
[37] Wirth N (1977) Modula: A language for modular multiprogramming. Software: Practice and Experience 7(1):1–35