Language-Driven Software Development

José-Luis Sierra

Fac. Informática, Universidad Complutense de Madrid
C/ Prof. José García Santesmases 9, 28040 Madrid, Spain
`jlsierra@fdi.ucm.es`

Abstract

Language-driven software development consists in applying computer language design and implementation techniques to build conventional software. The keynote reviews two different language-driven development approaches: *domain-specific languages* (DLSs), and *language-oriented architectures* (LOAs). The DSL approach focuses on the provision of languages specialized in different application aspects, which are used by developers, and even by domain experts, during application construction and maintenance. The LOA strategy, in its turn, conceives applications themselves as coordinated collections of language processors, which can be developed using language implementation tools (parser generators, attribute grammar-based systems, etc.). The presentation of the approaches is supported by case studies from the fields of knowledge-based systems, e-Learning, semi-structured data processing, and digital humanities.

1998 ACM Subject Classification D.3.4 Translator writing systems and compiler generators, D.3.2 Specialized Application Languages, D.2 Software Engineering, D.2.11 Software Architectures

Keywords and phrases domain-specific languages; language-oriented architectures; parser generators; attribute grammars; application domains

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.3

Category Invited Talk

1 Introduction

Nowadays design and implementation of computer languages is a mature and well-understood field, which comprises a wide spectrum of precise and well-founded methods, techniques and tools grounded in strong theoretical and mathematical principles [2]. Regardless of their initial limited applicability to the specialized compiler construction arena, recently these approaches have been recognized as very valuable instruments in many mainstream software development scenarios [43, 10, 19, 23, 24], which leads to a distinguished paradigm of software construction: *language-driven software development*. In these scenarios it is meaningful to recognize the linguistic nature of different aspects of software development, and therefore to undertake these aspects as ones concerning the explicit conception, design and implementation of special-purpose computer languages. The paradigm is particularly suited to address complex development situations involving sophisticated and highly customizable architectures, interdisciplinary teams of developers and domain experts, clearly defined stacks of abstraction levels, etc., in which the language development effort can pay off. Quoting to [1] " *. . . seen from this perspective, the technology for coping with large-scale computer systems merges with the technology for building new computer languages, and computer science itself becomes no more (and no less) than the discipline of constructing appropriate descriptive languages*".

3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 3–12

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This document summarizes the contents addressed in the keynote *Language-Driven Software Development* given in SLATE'14. The keynote is focused on two different practices concerning this development approach: *domain-specific languages* and *language-oriented architectures*. While the first one is well-established in the research community, and in recent years also among practitioners, the second one is more speculative and inspired by the Speaker's own research at Complutense University of Madrid, Spain (UCM).

2 Software Development based on Domain-Specific Languages

Quoting to [43] a domain-specific language (DSL) is "*a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain*". In this way, the concept of DSL is a well-established one in software development scenarios, where software developers have used several sorts of DSLs for decades (e.g., SQL for updating and querying relational databases, *make* or *ant* for tracking the dependencies among the files of a software systems and for automating the production of these software systems, YACC-like tools for generating parsers from grammar-based specifications, etc.) [31]. In addition, many DSLs have been also provided to facilitate application development in concrete domains [43, 24].

DSLs tailored to concrete application domains are particularly attractive from a software development perspective. Indeed, being more expressive and easy-to-use than a general-purpose programming language, these DSLs make possible, to some extent, the active participation of domain-experts in the development process. In this way, and in an idealized world, the aim of DSLs is to upgrade software developers to DSL designers and implementers, as well as to promote domain-experts to application developers [10, 12]. Of course, in the real world this idealized scenario (sometimes referred as *end-user programming* [16]) is hardly reachable due the fuzziness and dynamic nature of application domains, as well as to the difficulty of capturing some aspects of a system in terms of suitable domain abstractions. Regardless this difficulty, it is a matter of fact that this kind of DSLs can facilitate the active participation of domain-experts during the development process (e.g., they can understand specifications prepared in DSLs with suitable notations, suggest modifications, or even take the responsibility of producing and maintaining specific parts of the applications using suitable DSLs) [10, 12, 44].

Finally, before going into the details of DSL-based development, it is worthwhile to highlight the relationships among this approach and *model-driven engineering* [40, 44]. Indeed, model-driven engineering can be understood as a particular incarnation of DSL-based development, in which DSLs take the form of domain-specific meta-models, and DSL sentences take the form of models resulting of instantiating these meta-models. Therefore, many of the reflections made in the following presentation can be also applied to model-driven engineering without substantial change.

2.1 DSL-oriented Process Models

From a process model perspective, DSL-oriented software development shares many features with generative approaches to software development [21, 6]. Common activities undertook during DSL-oriented software development are *Domain Engineering* Activities, *Language Design and Implementation* Activities and *Application Development* Activities.

2.1.1 Domain Engineering Activities

These activities are oriented to determine the commonalities and the variability of applications in the target application domain [6], and those can be carried out by adopting well-established domain engineering approaches [17, 39, 41]. In DSL-oriented software development variability, which is concerned with the differences among concrete applications, is usually captured in terms of *feature models* [6] that set the conceptual basis for subsequent DSL design. On the other hand, commonality (the core part shared by all the applications in a domain) is essential for providing DSL runtime support (e.g., as a specific object-oriented framework).

2.1.2 Language Design and Implementation Activities

These activities deal with usual aspects concerning the design and implementation of computer languages (lexical and syntactical specification, specification of the static and dynamic semantics, etc.) [11]. For this purpose, a common practice in DSL design is to invert the conventional workflow in computer language design (i.e., going from *concrete* to *abstract* syntax [2]). Indeed, modern tendencies in DSL design promote to start by an abstract syntax. Following the jargon used by the DSL community, abstract syntax is formalized in object-oriented terms, as a set of interrelated classes that makes up the *semantic model* of the DSL [10, 19, 44]. This model will be based on the variability analysis performed during the domain engineering activities.

Once a suitable semantic model is available, it is possible to provide one or several alternative concrete syntaxes. Depending of the intended use of the DSL, concrete syntaxes can be embedded in general-purpose programming languages (*internal* syntaxes) or those can be externally provided (*external* syntaxes). Still, each alternative can be accomplished using a wide range of techniques:

- Concerning internal syntaxes, their provision strongly depends on the features of the host programming language. For instance, LISP-like homogeneous syntaxes have proven to be specially amenable for supporting internal syntaxes for many embedded DSLs [1], while extensible syntaxes enabled by user-defined operators like the supported by Prolog-like languages or by modern functional languages can be particularly useful for better fitting domain notations[15]. Recently, dynamic languages with very expressive grammars have been also adopted as host languages for DSLs [7, 12, 29, 30]. In object-oriented languages, two usual design patterns for internal syntax design are *method nesting expressions* and *fluent* APIs [10, 12].
- In its turn, external syntaxes can be accomplished by using a general-purpose semantic agnostic notation (like XML), by defining a DSL-specific textual syntax (the classic approach promoted by compiler construction textbooks), or even by defining a visual syntax amenable for implementations based on DSL workbenches [4, 14].

Concerning semantics, it is worthwhile to notice that the term *semantic model* is somewhat confusing, since the model has little to do with semantic processing, but it is an explicit formalization of the language abstract syntax. Semantics themselves must be added as processes that operate on the instances of the semantic model. For this purpose:

- As a representation of the abstract syntax, the semantic model usually addresses the structure of the sentences of the language. *Static semantics* deals with additional constraints beyond these structural aspects. While in classic language design static semantics lead to type systems [35], which are subsequently implemented as type checking algorithms on the abstract syntax trees / graphs, in the DSL world it is usual to find

more pragmatic approaches based on constraint languages for object-oriented models, like OCL [19], or in ontology-aware semantic technologies [46].

- *Dynamic semantics*, in their turn, take either the form of translations to target programming languages, or operational semantics specifications. The first scenario leads to the subsequent provision of code generators, while the second one leads to the provision of interpreters operating on instances of the semantic model [10].

In this way, the final implementation of the DSL typically consists of:

- A way of *editing* DSL programs. It can be as simple as using an existing text editor or an existing IDE (e.g., in the case of internal syntaxes or XML-based syntaxes), or as complex as using a dedicated IDE for the DSL. In order to cope with the later scenario it is possible to base the implementation of the DSL in a language workbench [4, 10, 14].
- A *binding* component, which maps concrete syntax sentences in semantic model instances. This component is analogous to the parser of a classic language processing architecture [2]. The exact nature of the component will depend of the nature of the concrete syntax and the semantic model.
- A *static semantic analyzer*. This component will be in charge of ensuring the additional semantic constraints on the semantic model instances.
- A *dynamic semantic infrastructure*. This infrastructure will vary on whether DSL execution is supported by translation or by interpretation. However, in both cases it is common to find a runtime support in terms of the domain-specific library or framework that results of the commonality analysis performed during the domain engineering activities. In this way, translators generate code that makes use of this domain-specific framework, while interpreters directly perform the operations on this framework required to carry out the execution (e.g., on-the-fly object instantiation and assembling, method invocation on the instantiated objects. etc.)

2.1.3 Application Development Activities

Once a suitable DSL is available, it can be used by developers and by domain experts to develop applications in the domain. As indicated earlier, in an idealized situation a DSL could free developers of application construction and maintenance in favor of domain-experts. However, a more realistic approach promotes the tight collaboration or both types of stakeholders in interdisciplinary development teams.

2.2 Development Process Dynamics

In a realistic DSL-based development process, DSLs must evolve according the expressive needs of domain experts. In this way, new expressive needs that are made apparent during application development imply the extension of the DSL infrastructure to accommodate these needs. As a consequence, DSL-based development processes are iterative and incremental in nature, promoting the iterative enhancement and the incremental extension of the DSL as a consequence of application construction.

The iterative and incremental nature of DSL construction shifts the recurrent software maintenance and evolution concerns to the language design and implementation level. Indeed, DSL maintenance and evolution is a keystone aspect of the DSL approach [42]. In particular, maintenance and evolution of dynamic semantics related components (translators and interpreters) are particularly critical due to the semantic modularity problem: local changes in a language can imply global changes in the associated processors [25, 45]. In this way, since DSLs are exposed to constant evolution and enhancement, the construction of their

processors (translators, interpreters) can take benefit of modularization techniques used in semantic specification and language processor construction [8, 15, 18, 22].

2.3 Some DSL-based Experiences

The Speaker of this keynote has been involved in DSL-based development in several fields, including knowledge-based systems and e-learning:

- During the early nineties of the past century, the Speaker had the opportunity of working at the Intelligent Systems Research Group, led by Prof. José Cuena, one of the pioneers of knowledge-based systems and artificial intelligence in Spain. Instead of using general-purpose knowledge representation formalisms (e.g., rule-based systems) to build intelligent systems, Cuena promoted the provision of formalisms specially tailored to each application domain, adopting in this way the concept of DSL several years before to its popularization and applying it to the development of knowledge-based systems for real-time decision-making support (in particular, Cuena developed several decision-making intelligent systems in the fields of traffic management and watershed management) [5]. Cuena's systems usually included specialized knowledge editors, which supported specialized knowledge-representation languages, and which were used directly by domain experts to provide the knowledge required by the system, as well as inference engines (interpreters of the aforementioned languages) able to execute the provided knowledge models. As a consequence of these experiences, Cuena's team developed an environment called KSM (*Knowledge Structured Management*), which was used to build this kind of knowledge-based systems [26]. In this sense, KSM could be understood as a sort of language workbench specialized in the field of knowledge-based systems.
- In 1998 the Speaker moved to UCM, where he was involved in several research projects concerning information management in e-learning. Indeed, e-learning is a field rich in examples concerning special-purpose languages (e.g., *educational modeling languages* intended to be used by instructors to describe the design of their courses [20]). At UCM, the Speaker took contact with the works done by the team of Prof. Fernández-Valmayor in the production and maintenance of complex educational hypermedia applications for second language learning [9]. In order to facilitate application maintenance, contents and other critical structures of the application were provided by domain experts (experts in philology) as structured documents marked with a SGML-based notation specific for the applications being constructed. These documents were subsequently processed in order to automatically update the application. Building on this idea, during the first decade of the present century, the Speaker worked on an approach for the development of educational (and other content intensive) systems based on the explicit formulation of XML-based DSLs, as well as in the construction of application generators for the resulting DSLs [37, 38].

3 Language-Oriented Architectures

The DSL approach promotes the use of language-driven techniques in the provision of domain-specific development tools. Indeed, a DSL is intended to describe different aspects of an application, but the internals of this application do not necessarily include language processing components. It can be even true when a DSL interpreter is used, since in this case the interpreter can bind the DSL description into an instantiation of an underlying runtime framework, and then to activate this instantiation by invoking suitable methods in the resulting objects. On the contrary, *Language-oriented Architectures* (LOAs), an approach

that the Speaker's team is experiencing at UCM, promotes to upgrade language processing techniques to the core of conventional applications.

3.1 Anatomy of a LOA

A LOA encourages the organization of an application as a coordinated set of language processors. Each processor operates on an information domain (e.g., a type of XML documents, an object-oriented class model, an even stream in an interactive application, etc.) and consists of:

- A *reader* that is able to read information instances in this information domain. As a consequence, it transforms these instances into sequences of tokens. Therefore, the reader plays the role of a scanner in a conventional language processor.
- A syntax-directed processor, which processes the sequence of tokens directed by its underlying syntactic structure. It is analogous to a parser extended with semantic actions.

3.2 Language Implementation Tools as General-Purpose Development Tools

There are not significant differences among the syntax-directed processor of a LOA and the corresponding component in a conventional language processor, since both components act on sequences of tokens. Indeed, readers in a LOA adapt information domains to the requirements of classic syntax-directed language processing models [2]. As a consequence, language implementation tools (like parse generators or attribute grammar-based tools) [2, 28], traditionally used in specialized fields like compiler construction, adopt a new and unexpected role as general-purpose development tools for applications architected according to the LOA principle. Indeed, a LOA-conforming application can be developed in terms of:

- A set of readers, one for each language processor that integrate the application. Although the provision of these readers can require conventional programming, in many information domains it will be possible to take advantage of the information structure (e.g., the markup in an XML document, the structure of an object model) to easily produce these readers by customizing generic ones using high-level customization specifications.
- A set of syntax-directed specifications (e.g., YACC, JavaCC or ANTLR translation schemes, LISA attribute grammars, etc.). These specifications are keystone assets in the development of the application, since they will serve to automatically generate the syntax-directed processors by using suitable language implementation tools (YACC, ANTLR, LISA ...)
- Additional conventional software components used to support the semantic actions invoked by the syntax-directed translators.

The resulting development approach to LOA applications has been called *grammatical* approach by the Speaker's team at UCM, since it strongly relies on the use of grammarware as primary development support. In addition, contrarily to other approaches that promote the application of grammatical formalisms specially tailored to each application domain (e.g., tree grammars in the XML field, graph grammars in model-driven engineering scenarios), the grammatical approach promotes the use of classic string-oriented grammars. The adaptation to each information domain is, in turn, delegated to suitable readers (in other words, to apply the grammatical approach to a new information domain, the first thing to do is to decide how to read information elements in this domain). Once it is done, it is possible to facilitate the application of the grammatical approach by devising specific grammar-based

notations for each particular domain, as well as ways of transforming these notations into the basic model.

3.3 Experiences with the LOA Approach

The Speaker's team at UCM is currently working with the characterization and generation of several kinds of processors to be integrated in applications organized according to LOAs:

- XML *syntax-directed processors*. The team has devised several models for processing XML documents based on the combination of XML stream-oriented processing frameworks (SAX and STaX) with parser generation tools (JavaCC and CUP) [34]. They also have defined a specific grammar-based notation for describing XML processing tasks based on attribute grammars (XLOP: *XML Language Oriented Processing*), together with its transformation into the processing framework integrated by STaX + CUP [33].
- JSON *syntax-directed processors*. The work concerning JSON processing is similar to the work concerning XML. In this case the parser generator tool was ANTLR [32]. Currently the team is working on JLOP (JSON Language Oriented Processing) a meta-tool similar to XLOP.
- Model transformation based on attribute grammars. Attribute grammars in this proposal operate on spanning trees of object networks serialized by suitable readers (a prototype implementation is described in [13]).
- Syntax-directed processors of event streams in interactive applications. The idea is similar to the described in [27], and oriented to generate controllers for interactive applications from attribute grammar-based specifications (see [36] for an alternative approach based on structural operational semantics specifications)

In addition, the team is also working on the definition of a LOA for @note, a RIA for the collaborative annotation of digitized literary text [3]

4 Closing

This keynote has reviewed two different approaches for bringing computer language design and implementation technologies to mainstream software development scenarios: DSLs, which are oriented to provide domain-specific development tools, and LOAs, which promotes to architect applications as coordinated sets of language processors. Both approaches are oriented to enhance the production and maintenance of applications by providing specification of components to higher levels than the enabled by general-purpose programming languages: domain-specific notations in the case of DSLs, grammar-oriented specifications in the case of LOAs.

Currently the Speaker's team is working on refining the concept of LOA and in applying it to real case-studies in the field of digital humanities. Concerning future work, the relationships and synergies of DSL and LOA-based approaches arise as a very promising field. Also a more in-depth insight concerning the relationships of these language-driven approaches with model-driven ones appears to be a promising concern to explore.

Acknowledgements. Work partially supported by project grant TIN2010-21288-C02-01.

References

1 Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

2 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2007.

3 Juan Cigarrán-Recuero, Joaquín Gayoso-Cabada, Miguel Rodríguez-Artacho, María-Dolores Romero-López, Antonio Sarasa-Cabezuelo, and José-Luis Sierra. Assessing semantic annotation activities with formal concept analysis. *Expert Syst. Appl.*, 41(11):5495–5508, 2014.

4 Steve Cook, Gareth Jones, Stuart Kent, and Alan Wills. *Domain-specific Development with Visual Studio Dsl Tools*. Addison-Wesley Professional, 2007.

5 José Cuena. Architectures for second generation knowledge based systems. In *Proceedings of the International Summer School on Advanced Topics in Artificial Intelligence*, pages 373–403, London, UK, UK, 1992. Springer-Verlag.

6 Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

7 Fergal Dearle. *Groovy for Domain-Specific Languages*. Packt Publishing, 1st edition, 2010.

8 Dominic Duggan. A mixin-based, semantics-based approach to reusing domain-specific programming languages. In Elisa Bertino, editor, *ECOOP*, volume 1850 of *Lecture Notes in Computer Science*, pages 179–200. Springer, 2000.

9 Baltasar Fernandez-Manjon and Alfredo Fernandez-Valmayor. Improving world wide web educational uses promoting hypertext and standard general markup language content-based features. *Education and Information Technologies*, 2(3):193–206, 1997.

10 Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

11 Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages, 3rd Edition*. The MIT Press, 3 edition, 2008.

12 Debasish Ghosh. *DSLs in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

13 Juan-Pablo Gracia. Marco para la transformacion de modelos basado en gramaticas de atributos. Master's thesis, Facultad de Informatica, UCM, 2010.

14 Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Eclipse Series. Pearson Education, 2009.

15 Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), December 1996.

16 Capers Jones. End-user programming. *IEEE Computer*, 28(9):68–70, 1995.

17 Kio C. Kang, Sholom G. Cohen, Janes A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

18 Uwe Kastens and William M. Waite. Modularity and reusability in attribute grammars. *Acta Inf.*, 31(7):601–627, October 1994.

19 Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.

20 Rob Koper and Bill Olivier. Representing the learning design of units of learning. *Educational Technology & Society,*, 7(3):97–111, 2004.

21 Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.

22 Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM.

23 Sjouke Mauw, Wouter T. Wiersma, and Tim A. C. Willemse. Language-driven system design. *International Journal of Software Engineering and Knowledge Engineering*, 14(6):625–663, 2004.

24 Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

25 Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.

26 Martín Molina, José-Luis Sierra, and José Cuena. Reusable knowledge-based components for building software applications: A knowledge modelling approach. *International Journal of Software Engineering and Knowledge Engineering*, 9(3):297–317, 1999.

27 Albert Nymeyer. A grammatical specification of human-computer dialogue. *Comput. Lang.*, 21(1):1–16, 1995.

28 Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, June 1995.

29 Paolo Perrotta. *Metaprogramming Ruby: Program Like the Ruby Pros*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2010.

30 Ayende Rahien. *DSLs in Boo: Domain Specific Languages in .Net*. Manning Pubs Co Series. Manning Publications Company, 2010.

31 Peter H. Salus. *Little Languages and Tools*. Macmillan Technical Publishing, 1st edition, 1998.

32 Antonio Sarasa-Cabezuelo and José-Luis Sierra. Grammar-driven development of json processing applications. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *FedCSIS*, pages 1545–1552, 2013.

33 Antonio Sarasa-Cabezuelo and José-Luis Sierra. The grammatical approach: A syntax-directed declarative specification method for xml processing tasks. *Comput. Stand. Interfaces*, 35(1):114–131, January 2013.

34 Antonio Sarasa-Cabezuelo, Bryan Temprado-Battad, Daniel Rodriguez-Cerezo, and José-Luis Sierra. Building xml-driven application generators with compiler construction tools. *Comput. Sci. Inf. Syst.*, 9(2):485–504, 2012.

35 Michael L. Scott. *Programming Language Pragmatics, Third Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2009.

36 José-Luis Sierra, Baltasar Fernández-Manjón, and Alfredo Fernández-Valmayor. A language-driven approach for the design of interactive applications. *Interacting with Computers*, 20(1):112–127, 2008.

37 José-Luis Sierra, Alfredo Fernández-Valmayor, and Baltasar Fernández-Manjón. A document-oriented paradigm for the construction of content-intensive applications. *Comput. J.*, 49(5):562–584, 2006.

38 José-Luis Sierra, Alfredo Fernández-Valmayor, and Baltasar Fernández-Manjón. From documents to applications using markup languages. *IEEE Softw.*, 25(2):68–76, March 2008.

39 Mark A. Simos. Organization domain modeling (odm): Formalizing the core domain modeling life cycle. In *Proceedings of the 1995 Symposium on Software Reusability*, SSR '95, pages 196–205, New York, NY, USA, 1995. ACM.

40 Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

41 Richard N. Taylor, Will Tracz, and Lou Coglianese. Software development using domain-specific software architectures: Cdrl a011—a curriculum module in the sei style. *SIGSOFT Softw. Eng. Notes*, 20(5):27–38, December 1995.

42 Arie van Deursen and Paul Klint. Little languages: Little maintenance. *Journal of Software Maintenance*, 10(2):75–92, March 1998.

43 Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

44 Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

45 Philip Wadler. How to declare an imperative. *ACM Comput. Surv.*, 29(3):240–263, September 1997.

46 Tobias Walter, Fernando Silva Parreiras, and Steffen Staab. Ontodsl: An ontology-based framework for domain-specific languages. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 408–422, Berlin, Heidelberg, 2009. Springer-Verlag.