Variable Domain-specific Software Languages with DjDSL

Stefan Sobernig

Variable Domain-specific Software Languages with DjDSL

Design and Implementation

Springer

Stefan Sobernig
Institute for Information Systems and New Media
Vienna University of Economics and Business
Vienna, Austria

ISBN 978-3-030-42151-9 ISBN 978-3-030-42152-6 (eBook)
https://doi.org/10.1007/978-3-030-42152-6

© Springer Nature Switzerland AG 2020
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

In software development, ideas of incorporating newcomers to programming as well as business and industry experts without software-development background into analyzing, designing, implementing, deploying, and operating software systems have been gaining importance rapidly. This is because software development is more and more about gluing together software from existing code artifacts including, for better or worse, open-source application frameworks while adequately trained developer workforce is scarce [14]. An inclusive and participatory software development must go beyond mere interactions via a graphical user interface (GUI) and via spreadsheet programming [17].

Language-oriented software development advocates the idea that architecting, designing, and implementing a software system should be driven by developing software languages [30]. Key to these ideas are special-purpose software languages: *domain-specific software languages* (DSL). For long, DSL have been provided as an alternative programming interface targeting developers in the application domain wrapped around application frameworks to ease framework integration [7, 24]. In the same way, a DSL empowers non-developer experts in interacting with a domain-specific software system. Take as an example of such a software system a scheduling system in the services domain of energy trading [26]. Other examples from a manufacturing domain are device simulators for automobile testbeds and sensor arrays [28].

A DSL-based approach aims at developing the entire software system around a DSL. At the same time, the objective is to empower domain experts (i.e., energy traders, test engineers) as *co*-developers of their supporting software systems (i.e., scheduling system, device simulator). This is achieved by providing expressive means and tailored tooling to specify, to adapt, and to configure their software systems. To pick up the above examples: By providing a DSL, energy traders are enabled to implement new types of trading reports or trading messages autonomously, that is, without the need for in-house or external software developers. Device testers can create variants of a device simulator on their own. In addition, testers can structure a suite of test cases on a given device in previously unforeseen ways.

As a software language, a DSL is based on a well-defined but domain-specific vocabulary in addition to well-defined notations (textual, graphical, and beyond), semantics, and language-specific tooling. Expectations towards DSL compare with those on general-purpose software languages (GPL) such as Java or C# and their industry-grade tooling (integrated development environments, IDE). DSL and DSL-based tooling (e.g., editors) must not only be robust but also as sophisticated as development environments for GPL. At the same time, they must be highly specialized for a given task and domain of application. This double objective presents a considerable challenge to DSL development and its adoption [22].

A DSL itself is implemented using a *DSL development system* [6]. A DSL development system can be provided by a tool spectrum ranging from extensible general-purpose software languages as DSL host languages to dedicated language workbenches. A DSL is developed using processes and techniques of domain engineering, including domain analysis, domain modeling (including semi-formal and formal modeling techniques), and domain implementation. This is to guarantee that a DSL is realized according to the stakeholders' requirements. Hence, a DSL represents a complex software system in its own right [2, 6].

As a consequence, developing a DSL represents a heavy upfront investment. A DSL adds to the overall costs of the overarching software development project [10, 31]. This is because a DSL requires a thorough and systematic analysis of the application domain [33]. Moreover, the resulting DSL design and implementation as a software language plus tooling has a considerable complexity. To realize benefits of a DSL and its tooling over time, its reuse within the original application domain and for similar application domains becomes necessary. Research has found a considerable potential for reuse. For example, many engineering domains require variants of state machines as an abstraction [13, 34]. A recent review [27] identified groups of multiple DSL (10+) having been developed for business or technical domains with a clear overlap (e.g., for modeling service-oriented architectures and web services). Even within the boundaries of a single business organization, one finds multiple DSL-based solutions targeting one and the same domain [28].

Despite such reuse potential at different scales, DSL development presents many technical barriers to reuse. These barriers add to more general inhibitors of technology reuse within and between organizations (e.g., ownership). To begin with, a DSL is developed over multiple development phases and involves multiple interconnected definition artifacts [29]. Examples include abstract syntax definitions, concrete-syntax definitions, structural and behavioural context conditions, implementation, test cases, and documentation. At the same time, acceptance and quality-in-use of a DSL are determined by its tailored tooling (e.g., editors, debuggers, and code generators). The more tailored a DSL-based tool, the better its uptake by practitioners in the one targeted domain but the less its reusability for others [22].

Therefore, methods, techniques, and tools to develop a *DSL for reuse* systematically—as a DSL family—have attracted much attention. The methods, techniques, and tools are borrowed from *software product line engineering* [1] and combined with those of *software language engineering* [11]. Software product line

engineering is inspired by the notion of mass customization based on composite products using product lines in business and manufacturing.[1] The ambition is to provide technical solutions to the following problems:

Problem Statements
How can a DSL be developed in a way that fosters its reuse and, at the same time, that maintains the consistency of the various design and development artifacts? How can this be achieved without (a) incurring excessive pre-planning or (b) without requiring ad hoc intervention by specifically skilled software developers?

A DSL, or DSL family more precisely, matching these expectations is said to have the *quality attribute of variability*, that is, the ability to derive different variants of a DSL systematically from a common set of implementation and configuration assets.

The main contributions documented in this book are the conceptual foundations, the design, and the implementation of the language-based DSL development system DjDSL. DjDSL facilitates design-decision making on and implementation of reusable DSL and DSL product lines. DjDSL embodies the state of the art in language-based and composition-based DSL development. DjDSL complements existing multi-DSL development approaches (e.g., Melange, MontiCore, Neverlang/AiDE) by contributing the following:

- DjDSL as a DSL development system allows a DSL developer to develop families of different DSL types (internal, external, and mixed).
- DjDSL provides for a variable design and implementation of a DSL family across the different definition artifacts in an integrated manner (*collaboration-based designs*): abstract syntax, context conditions, and concrete syntaxes.
- DSL variability can be represented and analyzed as a first-class development artifact, by using the variability modeling language of DjDSL.
- Variable language models can be developed in a platform-independent manner (using DjDSL's UML extensions) or a platform-specific manner (using DjDSL directly).
- DjDSL supports different kinds of DSL composition for different scopes (i.e., artifact types). The supported composition kinds include, among others, DSL extension and DSL unification. These can be applied for the scopes of abstract syntax, with or without context conditions, with or without concrete syntaxes.

[1] A general background on the sources of inspiration for software product line engineering (e.g., assembly lines, automotive platforms) is provided by Czarnecki and Eisenecker [3] and by Apel et al. [1].

To realize a *multi-DSL* development system, DjDSL makes a number of original technical contributions regarding the composability of DSL development artifacts. As for language models, DjDSL allows a DSL developer to structure an abstract syntax definition into composable collaborations. A collaboration can directly represent optional features of a language-model family. Under composition, DjDSL guarantees important properties, for example, a collaboration-aware method-resolution order. DjDSL introduces novel techniques of contract composition to render context conditions composable (e.g., trimming). At the level of variable textual syntaxes, DjDSL offers a novel infrastructure for composing grammars. Most notably, grammar composition in DjDSL extends to support object grammars [32]. External-syntax definition and processing in DjDSL is based on an extended variant of parsing expression grammars (PEG): *object parsing-expression grammars* (OPEG).

This book concentrates, as its title indicates, on my work at the crossroads between software language engineering, model-driven software engineering, and feature-oriented software engineering. The book was prepared as part of my long-term habilitation project on designing and on implementing *variable* domain-specific software languages. This research effort was, however, not limited to the topics and to the content now presented in this book. Regarding domain-specific languages (DSL) and software language engineering, I have been active in empirical research on the external quality attributes of DSL-based software systems. In this context, over the last 2 years, I have been overseeing and contributing to industry projects with emphasis on usability and quality-in-use of DSL and DSL-based tooling (HybriDLUX, DLUX). As for software architecture, I work on architectural design decisions and reusable design rationale for UML-based DSL. As for variability and feature-oriented software development, I am involved in research on feature-oriented software composition, static analyses of feature-oriented programs, and the search-based analysis of variability models. Clearly, many of these works helped shape the contributions of this book.

Book Audience and Assumed Background

The book aims at readers interested in language-oriented as well as model-driven software development, in their roles as software-engineering researchers and software developers alike. For these readers, the book is primarily designed as an instructional text on how to design and to develop DSL families (a.k.a. DSL product lines) using the DSL development system DjDSL. In doing so, the book also serves those that aim at understanding, in more general, the systematic procedures, fundamental design decisions, and diverse implementation choices for variable DSL. Packed with textbook-level examples on standard problems for every elaborated concept, the book offers complementary materials such as tutorials for master's- and doctorate-level courses in Computer Science and Software Engineering with focus on software language engineering.

Readers should be familiar with the basics of object-oriented modeling (UML, MOF, Ecore) and programming (Java). The object-oriented programming language Next Scripting Language (NX), an extension to the Tool Command Language (Tcl), dominates the book, while mainstream software languages (Java, C, grammar languages) are additionally used for illustration. Therefore, readers may also need to consult background resources on Tcl [15]. An understanding of software-engineering basics (architecture, design, implementation, testing) and software patterns (e.g., the Gang-of-Four patterns [8]) is required.

Ideally, readers should have previously obtained a basic understanding of and experience in software language engineering as covered by Lämmel [11] and software product lines as covered by Apel et al. [1].

Book Overview

Chapter 1: DSL as Variable Software The notion of a domain-specific language as variable software is presented in greater detail. In particular, this chapter introduces the reader to the idea of software product line engineering for DSL-based software systems (*DSL product lines*).

Chapter 2: Variability Support in DSL Development This chapter sheds light on the different interrelated dimensions of DSL variability: variable development processes, variable design-decision, and variability implementation techniques for DSL. The three subsequent chapters are devoted to the key conceptual and technical contributions of this book: DjDSL.

Chapter 3: Variable Language Models The reader is introduced to designing and to implementing the abstract syntax of a DSL in a variable manner (collaboration-based designs).

Chapter 4: Variable Context Conditions The means to refine an abstract syntax (language model) by using composable context conditions (invariants) are provided.

Chapter 5: Variable Textual Syntaxes Solutions to implement variable textual syntaxes for different DSL types (internal and external) are elaborated on. The theme common to all three chapters on DjDSL is to demonstrate the applicability of DjDSL to different types of DSL composition (e.g., extension and unification; [33]).

Chapter 6: A Story of a DSL Family A story is told about developing a mixed DSL in a stepwise manner. The story cuts across the previous chapters. This is to exercise the previously introduced but complementing techniques (variable language models, variable context conditions, and variable textual syntaxes) on an advanced example of developing a DSL family.

On Supplemental Materials

All development and auxiliary artifacts (i.e., models, source code, build scripts, tutorials) are available from the author's public repository [23]. The code listings in this book are extracted from *executable* how-tos and tutorials maintained in this repository. All code by the author reproduced in this book is licensed under the MIT license, copyright held by Stefan Sobernig (2018–2020). When appropriate, the exact location of a how-to or tutorial in the repository is given in the text as a navigable link.

On Notation

The main body of the text does not feature peculiar uses of notation. Names of methods, techniques, and tools are written as-is, for example. There are two minor exceptions: Words in small caps indicate names of software patterns such as STRATEGY [8]. This corresponds to the notational convention as established in the pattern community. Words typeset using a monotype font, such as `Class`, refer to keywords or entities defined in diagrams, models, or source code which are then referenced from within the main narrative.

The diagrammatic notation used for UML diagrams at all levels (architecture, design, implementation) defaults to the one of UML 2.5 [20] and of MOF 2.5 [19], respectively. The diagrams and underlying models were created using Papyrus 3.3 (Eclipse Modeling Tools 4.7.3a, Oxygen 3a) and MagicDraw 18.5. Model-level constraint expressions are implemented using the Object Constraint Language (OCL; [18]). More precisely, the implementation uses the Eclipse OCL (i.e., its CompleteOCL variant) as a reference implementation of OMG OCL 2.4.

The diagrammatic notation to visualize feature models is the Czarnecki–Eisenecker notation, as defined and documented in Czarnecki and Eisenecker [3, Chapter 4].

If not indicated otherwise, the source-code listings exhibit code fragments in the syntax of the Tool Command Language (Tcl; [15, 21]) and the Next Scripting Language (NX; [16, 25]), or derivatives therefrom.

On Running Examples

Throughout the book, a number of running application examples[2] of variable DSL are provided and referred to. The aim was to select examples that clearly

[2]Note that the term "case study" is consciously avoided. This is because, despite the selected examples going beyond toy examples, they do not qualify as case studies by any definition.

demonstrate the challenges of developing variable DSL. The second objective was to adopt examples that represent—in one—*typical* examples from the perspective of readers having a background in research on software composition or software language engineering. The selected examples are:

- *Graph-Product Line* (GraphPL): Lopez-Herrejon and Batory [12] defined a library for graph representation and graph analysis with nontrivial variability as a standard problem for product-line and software-composition techniques. Graphs invite the use of DSL, e.g., for defining graph structures or for defining transformations on graphs.
- *State-Machine Definition Language* (SMDL): Both need and uses of domain-specific variants of state-machine modeling languages are omnipresent in research and practise. There are documented attempts to capture them as DSL families [4, 34]. Fowler [7] uses a state-machine example (Miss Grant's Controller) to contrast internal and external DSL techniques.
- *Expression Language* (a.k.a. Expression Product Line, EPL): This is the ubiquitous example of a small but immediately useful software language for defining and for evaluating expressions of all sorts. This small language family is ideal as a running example because it has already been used to highlight common software-composition problems (e.g., abstraction mismatches under separation and composition of concerns).

Chapter 6 presents an application example of DjDSL beyond that of a running example only. This application example (Pongo) has already served as a standard problem on advanced model-driven software development, in particular on co-evolution of model transformations [9].

Vienna, Austria Stefan Sobernig
January 2020

References

1. Apel S, Batory D, Kästner C, Saake G (2013) Feature-oriented software product lines, 1st edn. Springer, Berlin. https://doi.org/10.1007/978-3-642-37521-7
2. Butting A, Eikermann R, Kautz O, Rumpe B, Wortmann A (2018) Controlled and extensible variability of concrete and abstract syntax with independent language features. In: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'18). ACM, New York, pp 75–82. https://doi.org/10.1145/3168365.3168368
3. Czarnecki K, Eisenecker UW (2000) Generative programming — Methods, Tools, and Applications, 6th edn. Addison-Wesley, Boston
4. 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 (SLE'15). ACM, New York, pp 25–36. https://doi.org/10.1145/2814251.2814252

5. Erdweg S, Giarrusso PG, Rendel T (2012) Language composition untangled. In: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA'12). ACM, New York, pp 7:1–7:8. https://doi.org/10.1145/2427048.2427055
6. Erdweg S, van der Storm T, Völter M, Tratt L, Bosman R, Cook WR, Gerritsen A, Hulshout A, Kelly S, Loh A, Konat G, Molina PJ, Palatnik M, Pohjonen R, Schindler E, Schindler K, Solmi R, Vergu V, Visser E, van der Vlist K, Wachsmuth G, van der Woning J (2015) Evaluating and comparing language workbenches: existing results and benchmarks for the future. Comput Lang Syst Struct **44**(Part A), 24–47. https://doi.org/10.1016/j.cl.2015.08.007
7. Fowler M (2010) Domain specific languages, 1st edn. Addison-Wesley, Boston
8. Gamma E, Helm R, Johnson RE, Vlissides J (1995) Design Patterns - Elements of Reusable Object-Oriented Software. Addison Wesley Professional Computing Series. Addison-Wesley, Boston
9. Hoisl B, Sobernig S (2015) Towards benchmarking evolution support in model-to-text transformation systems. In: Proceedings of the 4th Workshop on the Analysis of Model Transformations (AMT@MoDELS 2015), CEUR-WS.org, CEUR Workshop Proceedings, vol 1500, pp 16–25. http://ceur-ws.org/Vol-1500/paper3.pdf
10. Krueger CW (1992) Software reuse. ACM Comput Surv 24(2):131–183. https://doi.org/10.1145/130844.130856
11. Lämmel R (2018) Software languages: Syntax, Semantics, and Metaprogramming, 1st edn. Springer, Berlin. https://doi.org/10.1007/978-3-319-90800-7
12. Lopez-Herrejon RE, Batory DS (2001) A standard problem for evaluating product-line methodologies. In: Proceedings of the 3rd International Conference on Generative and Component-based Software Engineering (GPCE'01). Springer, Berlin, pp 10–24
13. Méndez-Acuña D, Galindo JA, Combemale B, Blouin A, Baudry B (2017) Reverse engineering language product lines from existing DSL variants. J Syst Softw 133:145–158. https://doi.org/10.1016/j.jss.2017.05.042
14. Miller H (2019) The times they are a-changin'. https://speakerdeck.com/heathermiller/the-times-they-are-a-changin, talk given at the Scale By the Bay 2019 Conference. Accessed 4, Dec 2019
15. Nadkarni AP (2017) The Tcl programming language: A comprehensive guide. CreateSpace Independent Publishing Platform
16. Neumann G, Sobernig S (2009) XOTcl 2.0 – A ten-year retrospective and outlook. In: Flynt C, Fox R (eds) Proceedings of the Sixteenth Annual Tcl/Tk Conference. Tcl Association Publications, Portland, pp 179–204. http://nm.wu-wien.ac.at/research/publications/b806.pdf
17. Neumann G, Sobernig S, Aram M (2014) Evolutionary business information systems: Perspectives and challenges of an emerging class of information systems. Bus Inf Syst Eng 6(1):33–38. https://doi.org/10.1007/s12599-013-0305-1
18. Object Management Group (2014) Object Constraint Language specification, formal/14-02-03. http://www.omg.org/spec/OCL/2.4/
19. Object Management Group (2016) OMG Meta Object Facility (MOF) core specification. http://www.omg.org/spec/MOF, version 2.5.1, formal/16-11-01
20. Object Management Group (2017) OMG Unified Modeling Language (UML). http://www.omg.org/spec/UML, version 2.5.1, formal/17-12-05
21. Ousterhout JK, Jones K (2009) Tcl and the Tk toolkit, 2nd edn. Addison-Wesley, Boston
22. Selic B (2012) What will it take? A view on adoption of model-based methods in practice. Softw Syst Model 11(4):513–526. https://doi.org/10.1007/s10270-012-0261-0
23. Sobernig S (2019) DjDSL. https://github.com/mrcalvin/djdsl/
24. Sobernig S, Gaubatz P, Strembeck M, Zdun U (2011) Comparing complexity of API designs: An exploratory experiment on DSL-based framework integration. In: Proceedings of the 10th International Conference on Generative Programming and Component Engineering (GPCE'11). ACM, New York, pp 157–166. https://doi.org/10.1145/2047862.2047890

25. Sobernig S, Neumann G, Adelsberger S (2012) Supporting multiple feature binding strategies in NX. In: Proceedings of the 4th International Workshop on Feature-Oriented Software Development (FOSD'12). ACM, New York, pp 45–53. https://doi.org/10.1145/2377816.2377823
26. Sobernig S, Strembeck M, Beck A (2013) Developing a domain-specific language for scheduling in the European energy sector. In: Proceedings of the 6th International Language on Software Language Engineering (SLE'13). Lecture Notes in Computer Science, vol 8225. Springer, Berlin, pp 19–35. https://doi.org/10.1007/978-3-319-02654-1_2
27. Sobernig S, Hoisl B, Strembeck M (2016) Extracting reusable design decisions for UML-based domain-specific languages: A multi-method study. J Syst Softw **113**, 140–172. https://doi.org/10.1016/j.jss.2015.11.037
28. Stieglbauer G, Burghard C, Sobernig S, Korosec R (2018) A daily dose of DSL - MDE micro injections in practice. In: Proceedings of the 6th international conference on model-driven engineering and software development (MODELSWARD'18), SciTePress, pp 642–651. https://doi.org/10.5220/0006754406420651
29. Strembeck M, Zdun U (2009) An approach for the systematic development of domain-specific languages. Softw. Pract. Exp. 39(15):1253–1292
30. Şutîi AM, van den Brand M, Verhoeff T (2017) Exploration of modularity and reusability of domain-specific languages: An expression DSL in MetaMod. Comput Lang Syst Struct. https://doi.org/10.1016/j.cl.2017.07.004
31. Tambe S, White J, Gray J, Hill JH, Gokhale AS, Schmidt DC (2009) Improving domain-specific language reuse with software product line techniques. IEEE Softw 26:47–53. https://doi.org/10.1109/MS.2009.95
32. van der Storm T, Cook WR, Loh A (2014) The design and implementation of object grammars. Sci Comput Program 96:460–487. https://doi.org/10.1016/j.scico.2014.02.023
33. van Deursen A, Klint P (2002) Domain-specific language design requires feature descriptions. J Comput Inf Technol 10(1):1–17
34. Wille D, Schulze S, Schaefer I (2016) Variability mining of state charts. In: Proceedings of the 7th International Workshop on Feature-oriented Software Development (FOSD'16). ACM, New York, pp 63–73. https://doi.org/10.1145/3001867.3001875

Acknowledgments

Obviously, no long-term effort of this kind takes place in a vacuum. Most importantly, I am much indebted to my family, especially: my wife, Marlis, kept me loved; my daughter, Teresa, kept me cheerful; my son, Sebastian, kept me smiling; my brother, Georg, backed me up; my mother, Susi, hosted us; Martina and Joe, my parents-in-law, let us rest; Alois, my father, kept encouraging me.

Many people gave inspiration, advise, aid, and encouragement. Some people deserve, however, more than general mention: Gustaf Neumann gave me the room and his support to work on my habilitation project at the Institute for Information Systems and New Media, WU Vienna. Gustaf Neumann maintains much of the "languageware" (NSF/NX) allowing for the contributions in this book. He is also coauthor on a paper that forms the basis for Chap. 3. The earlier works on DSL development by Mark Strembeck and Uwe Zdun inspired many contributions in this book. Noteworthy are their ideas on DSL design decisions and Uwe's DSL toolkit Frag. Joint work with Mark Strembeck and Bernhard Hoisl on reusable design decisions for UML-based DSL entered Chap. 2. Moreover, I would like to thank Sven Apel whose contributions to feature orientation have influenced this work.

Vienna, Austria
January 2020

Stefan Sobernig

Contents

1 DSL as Variable Software 1
1.1 Domain-Specific Languages 2
1.1.1 Modelware and Grammarware 3
1.1.2 Languages for Domain-Specific Modeling 4
1.1.3 Internal and External DSL 4
1.2 DSL Product Lines 6
1.2.1 Drivers Towards Variable DSL 6
1.2.2 DSL Product Lines 8
1.2.3 Trajectories to DSL Product Lines 11
1.3 Empirical Research and Technology Evaluation 13
1.3.1 Empirical Evidence on DSL 13
1.3.2 DSL-Aware Approaches to Technology Evaluation 13
1.4 Single-DSL Development Using NX 14
1.4.1 Implementing a Language Model 16
1.4.2 Textual Concrete Syntax 17
1.4.3 Integration and Execution 21
1.4.4 Testing 25
References 27

2 Variability Support in DSL Development 33
2.1 DSL Engineering Procedures 35
2.1.1 Problem Space 36
2.1.2 Solution Space 37
2.2 DSL Design Rationale 39
2.2.1 Reusable DSL Design Decisions 41
2.2.2 Excursus: Reusing Design Decisions for UML-Based DSLs 48
2.3 Variability Challenges 52
2.3.1 Types of DSL Composition 54
2.3.2 Abstraction Mismatches Under Composition 55
2.3.3 Development Processes 58

2.4 State of Variability Support 59
2.4.1 MetaMod 60
2.4.2 MPS 60
2.4.3 LISA 61
2.4.4 Melange 61
2.4.5 Gromp + Fusion 61
2.4.6 Neverlang + AiDE 62
2.4.7 MontiCore 62
2.4.8 Eco 63
2.4.9 MetaDepth 63
2.4.10 YAJCo 64
2.4.11 APEG 64
2.4.12 Helvetia + PetitParser 65
2.4.13 Silver + Copper 65
2.4.14 FeatureHouse + Spoofax 66
2.5 DjDSL: A Variability-Aware Development System for Mixed DSLs 66
References 68

3 Variable Language Models 73
3.1 Some Background on Language Models 73
3.1.1 Purpose 73
3.1.2 Definition Style 75
3.1.3 Variability and Variability Modeling 77
3.2 Feature-Oriented Decomposition and Composition of Language Models 78
3.2.1 Collaboration-Based Designs of Language Models 81
3.2.2 A UML Extension for Feature-Oriented Language Models 90
3.2.3 Feature-Oriented Language Models in DjDSL 100
3.3 Rendering DSL Variability Explicit 107
3.3.1 Abstract Syntax 110
3.3.2 Semantics 114
3.3.3 Concrete Syntax 117
3.3.4 Integration: Binary Decision Diagrams 121
3.4 Related Work 124
3.4.1 Model Composition and Model-Based Software Product Lines 124
3.4.2 Collaboration-Based and Feature-Oriented Object Orientation 127
3.4.3 (Textual) Variability Modeling 129
3.5 Summary and Discussion 131
References 132

4 Variable Context Conditions 137
4.1 Some Background on Context Conditions 138
4.1.1 Context Conditions under Feature Orientation 139
4.1.2 Context Conditions for DSL 140
4.2 Composition Strategies 143
4.2.1 Introduction-Only Composition 143
4.2.2 Chaining Composition 145
4.2.3 Combination Composition 146
4.2.4 Template Composition 148
4.3 Design and Implementation in DjDSL 150
4.3.1 Trimming (Including Introduction-Only) 152
4.3.2 Chaining 154
4.3.3 Combination 155
4.3.4 Templating 156
4.3.5 Intrinsic Context Conditions 159
4.3.6 Composition Order 161
4.3.7 Binding Times and Modes 161
4.4 Related Work 161
4.4.1 Composing Model-Based Context Conditions 162
4.4.2 Composing Contracts in FOP 163
4.5 Summary and Discussion 164
References 164

5 Variable Textual Syntaxes 167
5.1 Internal DSL: Pattern-Based Variability Implementation Techniques 168
5.1.1 DSL Extension 178
5.1.2 DSL Unification 185
5.1.3 DSL Extension Composition 191
5.2 External DSL: Grammar-Based Variability Implementation Techniques 197
5.2.1 Abstraction Mismatches of Parse Representations 197
5.2.2 Composable Parser Definitions 204
5.2.3 Parsing Expression Grammars and Packrat Parsing 218
5.2.4 Object Parsing-Expression Grammars (OPEG) 222
5.2.5 Design and Implementation 227
5.2.6 Composing Object Parsing Grammars 230
5.3 Related Work 251
5.4 Summary and Discussion 255
References 257

6 A Story of a DSL Family 261
6.1 Object-to-NoSQL Mapping: Pongo DSL 262
6.1.1 Domain of Application 262
6.1.2 Pongo 263
6.1.3 Problem Statement 265

6.2 Iteration 1: MVLP 266
6.2.1 Language Model (LM) 267
6.2.2 Context Conditions (CC) 268
6.2.3 Platform Integration (PI) 269
6.3 Iteration 2: Frontend Syntax 271
6.3.1 Concrete Syntax (CS) 271
6.4 Iteration 3: Collections 273
6.4.1 Language Model (LM) 274
6.4.2 Context Conditions (CC) 274
6.4.3 Platform Integration (PI) 275
6.4.4 Concrete Syntax (CS) 275
6.5 Iteration 4: Mixed Syntax 277
6.5.1 Concrete Syntax (CS) 278
6.6 Lessons Learned 280
6.6.1 Feature Traceability 281
6.6.2 Separation and Composition of Concerns 281
6.6.3 Unmodified Reuse 282
6.6.4 Preplanning 282
References 282

Postface 285
References 287

A A PEG for Object Parsing-Expression Grammars 289

Index 293

Acronyms

ADSL	Architecture domain-specific language
ANTLR	Another Tool for Language Recognition
API	Application programming interface
AST	Abstract syntax tree
BCEL	Boolean Comparison and Expression Language
BDD	Binary decision diagram
CFG	Context-free grammar
CNF	Conjunctive normal form
CP	Constraint programming
CRUD	Create, read, update, and delete
CSP	Constraint satisfaction problem
DAG	Directed acyclic graph
DNF	Disjunctive normal form
DR	Design rationale
DSL	Domain-specific (software) language
DSML	Domain-specific modeling language
E/BNF	Extended/Backus Naur Form
EMF	Eclipse Modeling Framework
E/MOF	Essential/Meta Object Facility
EPL	Expression Product Line
FFD	Free Feature Diagrams
FOP	Feature-oriented programming
GLL	Generalized left-to-right, leftmost
GLR	Generalized left-to-right, rightmost
GMF	Graphical Modeling Framework
GPL	General-purpose (software) language
GraphPL	Graph-Product Line
GUI	Graphical user interface
IDE	Integrated development environment
INF	If-then-else normal form
JML	Java Modeling Language

LALR	Lookahead left-to-right, rightmost
LEA	Literal, Expression, Addition
LHS	Left-hand side
LL	Left-to-right, leftmost
LOP	Language-oriented programming
LR	Left-to-right, rightmost
MRO	Message-resolution order
MTD	Model-transformation diagram
NX	Next Scripting Language
OCL	Object Constraint Language
OMG	Object Management Group
ONM	Object-to-NoSQL mapper
OOP	Object-oriented programming
OPEG	Object parsing-expression grammar
PEG	Parsing expression grammar
QOC	Questions-Options-Criteria
RAC	Runtime assertion checking
RHS	Right-hand side
SAT	Satisfiability
SDF	Syntax Definition Formalism
SMDL	State-Machine Definition Language
Tcl	Tool Command Language
UML	Unified Modeling Language
VFD	Varied Feature Diagrams