Domain-Specific Languages

Arie van Deursen Paul Klint Joost Visser

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
`http://www.cwi.nl/~{arie,paulk,jvisser}/`

ABSTRACT
Domain-Specific Languages are used in software engineering in order to enhance quality, flexibility, and timely delivery of software systems, by taking advantage of specific properties of a particular application domain. This survey covers terminology, risks and benefits, examples, design methodologies, and implementation techniques of domain-specific languages as used for the construction and maintenance of software systems. Moreover, it covers an annotated selection of 75 key publications in the area of domain-specific languages.

1998 ACM Computing Classification System: D.3

Keywords and Phrases: Example DSLs, DSL design, DSL implementation, survey.

Note: Work carried out under project SEN 1.5, *Domain-Specific Languages*, sponsored by the *Telematica Instituut*.

1 Introduction

In all branches of science and engineering one can distinguish between approaches that are *generic* and those that are *specific*. A generic approach provides a general solution for many problems in a certain area, but such a solution may be suboptimal. A specific approach provides a much better solution for a smaller set of problems. One of the incarnations of this dichotomy in computer science is the topic of this survey: *domain-specific languages* versus generic programming languages.

Of course, this is not a new topic. The older programming languages (Cobol, Fortran, Lisp) all came into existence as dedicated languages for solving problems in a certain area (respectively business processing, numeric computation and symbolic processing). Gradually they have evolved into general purpose languages and over and over again the need for more specialized language support to solve problems in well-defined application *domains* has resurfaced. Over time, the following solutions have been tried:

- *Subroutine libraries* contain subroutines that perform related tasks in well-defined domains like, for instance, differential equations, graphics, user-interfaces and databases. The subroutine library is *the* classical method for packaging reusable domain-knowledge.
- *Object-oriented frameworks* and *component frameworks* continue the idea of subroutine libraries. Classical libraries have a flat structure, and the application invokes the library. In object-oriented frameworks it is often the case that the framework is in control, and invokes methods provided by the application-specific code [42, 32].
- A *domain-specific language* (DSL) is a small, usually declarative, language that offers expressive power focused on a particular problem domain. In many cases, DSL programs are translated to calls to a common subroutine library and the DSL can be viewed as a means to hide the details of that library.

Although many domain-specific languages have been designed and used over the years, the systematic study of domain-specific languages has only started more recently. This survey provides an inventory of the field and covers references to research that deals with the following topics: terminology (Section 2), risks and opportunities (Section 3), example DSLs (Section 4), DSL design methodology (Section 5), and DSL implementation strategies (Section 6). The papers listed are annotated with summaries, which in turn are cross-referenced to related papers.

2 Terminology

The question what exactly is a domain-specific language is subject to debate. We propose the following definition:

> 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.

The key characteristic of DSLs according to this definition is their *focussed* expressive power.

Our definition inherits the vagueness of one of its defining terms: *problem domain*. Rather than attempting to define this volatile notion as well, we list and categorize a number of domains for which DSLs have actually been built in Section 4. Moreover, we refer to [70], which contains an interesting discussion contrasting a "domain as the real world" point of view as adopted in the artificial intelligence community, with a "domain as a set of systems" approach, as used in the systematic software reuse research community.

DSLs are usually *small*, offering only a restricted suite of notations and abstractions. In the literature they are also called *micro-languages* and *little languages* [7]. Sometimes, however, they contain an entire general-purpose language (GPL) as a sublanguage, thus offering domain-specific expressive power *in addition to* the expressive power of the GPL. This situation occurs when DSLs are implemented as *embedded languages* (see Section 6). Languages such as Cobol or Fortran, which could be viewed as languages tailored towards the domain of business and scientific programming, respectively, are generally not regarded as DSLs, because they are not small and because their expressive power is not restricted to these domains.

Domain-specific languages are usually *declarative*. Consequently, they can be viewed as specification languages, as well as programming languages. Many DSLs are supported by a DSL compiler which generates applications from DSL programs. In this case, the DSL compiler is referred to as *application generator* in the literature [17], and the DSL as *application-specific language*. Other DSLs, such as YACC [7] or ASDL [77], are not aimed at programming (specifying) complete applications, but rather at generating libraries or components. Also, DSLs exist for which execution consists in generating documents (TEX), or pictures (PIC [7]). A common term for DSLs geared towards building business data processing systems is 4th Generation Language (4GL).

Related to domain-specific programming is *end-user programming*, which happens when end-users perform simple programming tasks using a macro or scripting language. A typical example is spreadsheet programming using the Excel macro-language.

3 Risks and Opportunities

Adopting a DSL approach to software engineering involves both risks and opportunities. The well-designed DSL manages to find the proper balance between these two. The *benefits* of DSLs include:

- DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs.
- DSL programs are concise, self-documenting to a large extent, and can be reused for different purposes [50].
- DSLs enhance productivity, reliability, maintainability [24, 47], and portability [38].
- DSLs embody domain knowledge, and thus enable the conservation and reuse of this knowledge.
- DSLs allow validation and optimization at the domain level [6, 13, 55].
- DSLs improve testability following approaches such as [71].

The *disadvantages* of the use of a DSL are:

- The costs of designing, implementing and maintaining a DSL.
- The costs of education for DSL users.
- The limited availability of DSLs [49].
- The difficulty of finding the proper scope for a DSL.

- The difficulty of balancing between domain-specificity and general-purpose programming language constructs.
- The potential loss of efficiency when compared with hand-coded software.

Comparisons of the DSL approach to other approaches to software generation are made in [20, 22, 47]. In [24] the costs and benefits of DSLs are analyzed from the perspective of software maintenance. In [49], DSLs are categorized as one of the main approaches to software reuse, and a detailed comparison is made to other reuse techniques.

4 Example DSLs

Literally hundreds of DSLs are in existence today. Of these, only a subset is actually described in the software engineering or programming language literature. Best-known are classical examples like PIC, SCATTER, CHEM, LEX, YACC, and Make, which are described in [7]. Other well-known examples are SQL, BNF, and HTML. We have included references to various example domain-specific languages. Their domains can be grouped into the following areas:

- **Software Engineering**
  Financial products [12, 22, 24], behavior control and coordination [9, 10], software architectures [54], and databases [39].
- **Systems Software**
  Description and analysis of abstract syntax trees [77, 19, 51], video device driver specifications [76], cache coherence protocols [15], data structures in C [72], and operating system specialization [63].
- **Multi-Media**
  Web computing [14, 35, 4, 33], image manipulation [73], 3D animation [29], and drawing [44].
- **Telecommunications**
  String and tree languages for model checking [48], communication protocols [6], telecommunication switches [50], and signature computing [11].
- **Miscellaneous**
  Simulation [2, 13], mobile agents [36], robot control [61], solving partial differential equations [26], and digital hardware design [41].

A collection of several papers on DSLs can be found in [67].

5 DSL Design Methodology

The development of a domain-specific language typically involves the following steps (see [17, 24]):

- **Analysis**
  (1) Identify the problem domain. (2) Gather all relevant knowledge in this domain. (3) Cluster this knowledge in a handful of semantic notions and operations on them. (4) Design a DSL that concisely describes applications in the domain.
- **Implementation**
  (5) Construct a library that implements the semantic notions. (6) Design and implement a compiler that translates DSL programs to a sequence of library calls.
- **Use**
  (7) Write DSL programs for all desired applications and compile them.

The aim of the analysis steps (1) through (4) is to build up a thorough understanding of the underlying application domain. Guidelines for acquiring such an understanding are provided by the research area of *domain analysis* which investigates ways of modeling domains. Following [58], a *domain analyst* is a person who examines the needs and requirements of a collection of systems which seem "similar". Neighbors emphasizes that this is work that only can be done by a person who has built many systems for different customers in the same problem area. The domain analyst is like a systems analyst, except that the goal is to support the development of families of related systems, not just one-of-a-kind productions [75].

Domain engineering [3] refers to the activity of systematically modeling domains. Domain engineering originates from research in the area of *software reuse*, and can be used when constructing domain-specific reusable libraries, frameworks or languages. A recent domain engineering survey is provided by [20, Chapter 3]. Several domain engineering methodologies exists, of which ODM (Organizational Domain Modeling [69, 70]), FODA (Feature-Oriented Domain Analysis [45]), and DSSA (Domain-Specific Software Architectures [75]) are best known.

Strongly related to domain engineering is the notion of *program families* which are sets of similar programs [52, 18]. At Lucent, a systematic approach to the development of families is in use, the Family-Oriented Abstraction, Specification and Translation (FAST) approach, which has been successfully applied to over 25 different domains [18]. Program families are in turn related to *software product lines*. These emphasize features shared by all products, and are focused on the needs of a selected *market* [21, 53, 78].

A prerequisite to developing a DSL is mature domain knowledge. For that reason, a DSL is viewed as the final and most mature phase of the evolution of an object-oriented application framework [66, 22]. For the same reason, the existence of *legacy systems* implementing domain concepts will be of use when developing a DSL for that domain [70]. Reverse engineering techniques may be used to distill domain knowledge from such legacy systems — an overview of such techniques is provided by [16, 25].

6 DSL Implementation

The implementation steps (5) and (6) of the previous section can be carried out using several approaches:

Interpretation or compilation

This is the classical approach to implementing a new language. Standard compiler tools [1, 7] can be used, or tools dedicated to the implementation of DSLs like Draco [58], ASF+SDF [23], Kephera [31], Kodiyak [38], design by selection [62], or InfoWiz [56].

The main advantage of building a compiler or interpreter is that the implementation is completely tailored towards the DSL and no concessions are necessary regarding notation, primitives and the like. Also, error detection, static analysis, and optimizations can be done at the domain level, for example using an *effect system* as in [13].

Clearly, an important problem is the cost of building such a compiler or interpreter from scratch, and the lack of reuse from other (DSL) implementations, although some DSL tool sets (for example InfoWiz [56]) are particularly designed to overcome such problems.

As an alternative to implementing a DSL from scratch, a DSL can be implemented by extending a given base language. For instance, [6] describes an extension of (a restricted version of) a general-purpose language with domain-specific constructs. The main advantage of this approach is that all features of the base language remain available and need not be re-implemented.

When implementing domain-specific extensions of a base language, the implementation of the base language can be reused in three different ways:

Embedded languages / domain-specific libraries

In this approach, existing mechanisms such as definitions for functions or operators with user-defined syntax are used to build a library of domain-specific operations. The syntactic mechanisms of the base language are used to express the idiom of the domain.

An advantage of this approach is that the compiler or interpreter of the base language is reused *as is* for the DSL. The main limitation is in the expressiveness of the syntactic mechanisms in the base language. In many cases, the optimal domain-specific notation has to be compromised to fit the limitations of the base language. Typical examples of this approach are [61] (a robot control language embedded in Haskell) and [44] (a PIC-like drawing language embedded in ML). The concept of *domain-specific embedded language* was coined by Hudak [40].

Preprocessing or macro processing

In this approach the new constructs are translated to statements in the base language by a preprocessor. The main advantage of this approach is simplicity. Its main disadvantage is that static checking and optimization are not done at the domain level.

Consequently, generated code is error prone, and the user is provided with feedback on these errors at the level of the base language, or only at run-time.

Extensible compiler or interpreter

This approach is similar to the previous one, but the preprocessing phase is now integrated in the compiler. The advantage is that more type checking and better optimization is possible. This approach is taken by [30, 74]. The Tcl [59] interpreter is also a prime example: it has been extended for dozens of domains.

Apart from building a dedicated DSL compiler or interpreter, or reusing the implementation of an underlying base language, other implementation techniques may be used. For instance, in *aspect-oriented programming* [46] a DSL is used to describe an aspect of a system's behavior that is orthogonal to its main functionality. An *aspect weaver* is then used to generate domain-specific code and merge it with the main code.

7 Concluding Remarks

This survey on domain-specific languages covered covered terminology, risks and opportunities, example DSLs, and design and implementation issues, listing relevant references for each of these topics. The references themselves are annotated with a summary of the most important results discussed in each paper.

For up to date information on the topic of domain-specific languages, we refer to the series of DSL conferences organized by USENIX [64, 27], which most likely will have successors in the years to come.

Another valuable source of up to date information may be the web. A searchable domain engineering bibliography, with abstracts, is available at `http://www.iese.fhg.de/pubs_and_links/spl/bibliography/`. An online bibliography on the topic of generative programming can be found at `http://home.t-online.de/home/Ulrich.Eisenecker/gpref.htm`. Finally, `http://www.irisa.fr/compose/dsl/` provides a survey of domain-specific languages in general.

Acknowledgments

This research was sponsored by the Dutch *Telematica Instituut*, project DSL (see also `http://www.cwi.nl/projects/dsl/`).

An earlier version of this paper appeared as *Domain-Specific Languages: An Annotated Bibliography* in ACM SIGPLAN Notices, 2000.

We would like to thank Jan Heering from CWI for many useful remarks.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compiler: Principles, Techniques and Tools*. Addison-Wesley, 1986.

Standard text on compiler construction.

[2] M. Antoniotti and A. Göllü. SHIFT and SMART-AHS: A language for hybrid system engineering modeling and simulation. In Ramming [64], pages 171–182.

Describes the language SHIFT for hybrid system simulation. Main application area is traffic simulation. Implemented by translation to C and a run-time library with solvers for various kinds of differential equations.

[3] G. Arango. Domain analysis: From art form to engineering discipline. In *Fifth International Workshop on Software Specification and Design*, pages 152–159, May 1989. Appeared as ACM SIGSOFT Engineering Notes 14(3).

Outlines a framework to synthesize domain analysis methods, and to compare between different methods. The paper advocates an incremental, evolving approach towards developing domain models.

[4] D. Atkins, T. Ball, G. Bruns, and K. Cox. Mawl: A domain-specific language for form-based services. In DSL-IEEE [28], pages 334–346. An earlier version appeared in [64].

Describes the language Mawl that is intended for implementing form-based information services for different devices (web browser, interactive voice response service). The main contributions of this language are: (1) separation of user-interface *code and service logic, (2) static type checking, (3) device-independence, (4) automatic generation of low-level CGI code, (5) automatic generation of HTML templates, and (6) automatic generation of usage statistics.*

[5] D. R. Barstow. Domain-specific automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1321–36, November 1985.

Envisions a framework for stepwise synthesis of domain-specific applications from informal specifications. The framework applies search techniques to explore possible refinements of an initial specification, given a base of domain and programming knowledge (facts and heuristics).

[6] A. Basu, M. Hayden, G. Morrisett, and T. von Eicken. A language-based approach to protocol construction. In Kamin [43], pages 1–15.

Reports on the design and implementation of Promela++, a DSL for protocol construction and validation. Promela++ adds domain-specific constructs to restricted C, and supports validation and optimization on the domain-level.

[7] J. L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, August 1986.

Demonstrates and advocates the use of "little languages". Takes PIC as an example, as well as a number of little languages from which PIC input is generated (SCATTER, CHEM), and little languages that were used to implement PIC (LEX, YACC, Make). Contrasts three approaches: interactive systems, subroutine libraries, and little languages. Discusses DSL design principles.

[8] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.

Introduces the Syntax Definition Formalism SDF, the Algebraic Specification Formalism ASF, and their combination, ASF+SDF, which can be used to describe the syntax and semantics of (domain-specific) languages. Contains several language definition case studies. See also [12, 23].

[9] J.A. Bergstra and P. Klint. The discrete time TOOLBUS—a software coordination architecture. *Science of Computer Programming*, 31:205–229, 1998.

Describes how a language based on process algebra is used in the TOOLBUS *coordination architecture for building heterogeneous, distributed software systems. See also [10].*

[10] F. Bertrand and M. Augeraud. BDL: A specialized language for per-object reactive control. In DSL-IEEE [28], pages 347–362. An earlier version appeared in [64].

Many object-oriented languages contain only implicit constraints on the order of application of the methods in a class. This paper introduces the Behaviour Description Language (BDL) which uses a process-oriented notation to describe this ordering. BDL is translated to C, with ESTEREL as intermediary. The resulting C code is linked with a C++ program and acts as controller for the execution of C++ classes. See also [9].

[11] D. Bonachea, K. Fisher, A. Rogers, and F. Smith. Hancock: A language for processing very large-scale data. In DSL-99 [27], pages 163–176.

Describes the language Hancock that is intended for signature computations *on the data collected from telephone calls. A signature is a user profile with applications ranging from fraude detection to marketing. Typical issues are the large volume of data, the complex traversal patterns of these data and the different levels of precision for signatures. Hancock is translated to C combined with several run-time libraries. The major benefit of this DSL is a separation of concerns (traversal patterns, efficiency, signature computations). As a result programmers can concentrate on the signature computation, since the other concerns are taken care of by the DSL compiler. The major reason to design a DSL (as opposed to using a library) were the traversal patterns that cannot be captured in a library. The paper concludes with a description of the design process used.*

[12] M. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 9–18. Springer-Verlag, 1996.

Provides an overview of some industrial applications of the language prototyping environment ASF+SDF. The RISLA case study, involving a language for describing financial products, is discussed in considerable detail, covering pure RISLA, modular RISLA, and RISQUEST, a language for generating questionnaires used when composing new products. From a modular RISLA product description, COBOL code is generated for accessing a library of COBOL functions providing operations on cash flows, balances, intervals, and the like. See also [22, 24].

[13] D. Bruce. What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation. In Kamin [43], pages 17–35.

Discusses the design of a DSL for parallel discrete event simulation. On the basis of this experience a number of observations are made regarding DSL design principles. Most notably, the use of a strong effect system *is advocated to do static checking on the domain level, and to determine applicability of optimizations.*

[14] L. Cardelli and R. Davies. Service combinators for web computing. In DSL-IEEE [28], pages 309–316. An earlier version appeared in [64].

Access to the resources of the World-Wide Web is usually obtained though manual browsers. Service combinators are intended for writing programs that reproduce human browsing behaviour, including reactions to slow transmission rates and various kinds of failure. Based on a concurrent programming model, the paper gives both an informal and formal treatment of a DSL for Web computing.

[15] S. Chandra, , B. Richards, and J. R. Larus. Teapot: A domain-specific language for writing cache coherence protocols. In DSL-IEEE [28], pages 317–333. An earlier version appeared in [64].

The problem of cache coherence occurs when local replica of shared data are made in a distributed system in order to improve its scalability and performance. Writing the code to support coherence protocols is error-prone. This paper describes experience with the language Teapot for describing these protocols. Teapot programs can be translated to (1) C code that implements the protocol, or (2) input for an automatic verifier. Two case studies and overall experience with this approach are discussed.

[16] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

Overview of reverse engineering techniques, which also can be used to distill domain knowledge from legacy system. See also [25, 70].

[17] J. C. Cleaveland. Building application generators. *IEEE Software*, pages 25–33, July 1988.

Uses the term "application generators" to refer to DSL compilers. Gives a compiler generator architecture diagram. Describes relationships between roles of customers, domain engineers and system engineers. Lists pros and cons of application generators. Describes "Stage", an application-generator development tool. Describes a methodology for building an application generator.

[18] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *IEEE Software*, pages 37–45, November/December 1998.

A software family *is a set of similar systems with possibly many different variations. Scope, commonality, and variability (SCV) analysis gives software engineers a systematic way of thinking about and identifying the product family they are creating. The paper describes the Family-Oriented Abstraction, Specification, and Translation (FAST) approach, which has been used with immediate payoff in over 25 domains at Lucent Technologies.*

[19] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In Ramming [64], pages 229–242.

Introduces a Prolog-based query language for analyzing abstract syntax trees of C/C++ programs.

[20] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley, 1999. To appear.

Gives a comprehensive discussion of a range of programming techniques that involve some sort of code generation step, such as aspect-oriented, subject-oriented, and adaptive programming, composition filters, and domain-specific languages. Chapter 3 of this book provides a survey of domain-engineering methods.

[21] J.-M. DeBaud and K. Schmid. A systematic approach to derive the scope of software product lines. In *21st International Conference on Software Engineering, ICSE-99*, pages 34–43. ACM, 1999.

Argues that economic motives should be used for scoping software product lines, rather than more traditional domain engineering methods. The paper proposes PuLSE, which iteratively refines business objectives towards more operational evaluation criteria.

[22] A. van Deursen. Domain-specific languages versus object-oriented frameworks: A financial engineering case study. In *Smalltalk and Java in Industry and Academia, STJA'97*, pages 35–39. Ilmenau Technical University, 1997.

Contrasts domain-specific languages with object-oriented frameworks by comparing two projects in the financial engineering domain: RISLA (DSL) and the ET++SwapsManager (OO framework). See also [12].

[23] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

Describes the use of ASF+SDF as a meta-language for the specification of syntax and semantics. After introducing ASF+SDF, a number of language specification case studies are presented, and various styles for writing language specifications are illustrated. Moreover, different techniques for generating tools from these are presented. See also [8].

[24] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

Domain-specific languages (DSLs) have the potential to make software maintenance simpler: domain-experts can directly use the DSL to make required routine modifications. At the negative side, however, more substantial changes may become more difficult: such changes may involve altering the domain-specific language. This will require compiler technology knowledge, which not every commercial enterprise has easily available. The paper describes and uses the experience of the RISLA language for interest rate products to discuss the role of DSLs in software maintenance, the opportunities introduced by using them, and techniques for controlling the risks involved. See also [12].

[25] A. van Deursen, P. Klint, and C. Verhoef. Research issues in software renovation. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering, FASE99*, volume 1577 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, 1999.

Overview of parsing, transformation, and program understanding techniques that can be used when searching for domain knowledge in legacy systems. See also [16, 70].

[26] T. B. Dinesh, M. Haveraaen, and J. Heering. An algebraic programming style for numerical software and its optimization. Technical Report SEN-R9844, CWI, 1998. ACM CoRR Preprint Server cs.SE/9903002 (March 1999). Submitted to *Scientific Programming*.

Discusses a domain-specific programming style for the domain of partial differential equations, using an expression style directly obtained from the underlying algebraic theory. The use of this style permits optimizations beyond the scope of current compiler optimizations.

[27] *Proceedings of the second USENIX Conference on Domain-Specific Languages*. USENIX Association, October 3–5 1999.

[28] Special issue on domain-specific languages. *IEEE Transactions on Software Engineering*, 25(3), May/June 1999.

[29] C. Elliott. An embedded modeling language approach to interactive 3D and multimedia animation. In DSL-IEEE [28], pages 291–308. An earlier version appeared in [64].

Describes a multi-media extension for Haskell and discusses the merits of Haskell as basis for domain-specific extensions.

[30] D. R. Engler. Interface compilation: Steps toward compiling program interfaces as languages. In DSL-IEEE [28], pages 387–400. An earlier version appeared in [64].

Describes the extensible ANSI C compiler framework MAGIK, which allows the dynamic incorporation of user-defined compiler extensions. The extensions can transform, optimize or inspect the generated intermediate representation. The approach gives safe access to compiler internals and supports full optimization of application-specific language extensions. Implemented on top of lcc. See also [74].

[31] R. E. Faith, L. S. Nyland, and J. F. Prins. Khepera: A system for rapid implementation of domain specific languages. In Ramming [64], pages 243–55.

Presents Khepera, a tool kit for rapid implementation and long-term maintenance of DSLs via source-to-source transformation separated into three phases: parsing, AST transformation, and pretty-printing.

[32] M. E. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

Introduction to a special issue on (domain-specific) object-oriented frameworks, which are defined as reusable, semi-complete applications that can be specialized to produce custom applications. Covers classification, strengths and weaknesses, and future trends. See also [66].

[33] M. Fernández, D. Suciu, and I. Tatarinov. Declarative specification of data-intensive web sites. In DSL-99 [27], pages 135–148.

Covers a query language to describe data-intensive web sites. Three programming tasks are distinguished to build such *sites: accessing and integrating the data available in the site, building the site's structure, and generating the HTML representation of the site. The solution proposed is a declarative query language (StruQL) to define the site's content and structure, a template language to define the HTML representation and an extension of the query language with functions to describe dynamic behaviour and to promote reusability of queries. Reengineering an existing AT&T web site using this approach has resulted in less, more maintainable, code with more functionality. The initial learning curve of the new language is more than compensated for by the advantages gained.*

[34] M. Fromherz, V. Gupta, and V. Saraswat. cc — A generic framework for domain-specific languages. In Kamin [43], pages 89–96.

Proposes cc*, a family of languages for concurrent constraint programming, as a framework for DSL construction. Two approaches are explained by example: building a DSL on top of* cc*, and extending* cc *with domain-specific constructs.*

[35] M. Fuchs. Domain specific languages for ad hoc distributed applications. In Ramming [64], pages 27–36.

The current architecture of the Web is based on a client/server model in which most of the computation is done at the server side, while the client side is a browser that only displays the results of server computations. SGML/XML is used as meta-language for describing the interactions between heterogeneous agents on the Web. Essentially, a grammar is defined of all possible interactions and this grammar steers the behaviour of each agent. See [9] for a fully process-based approach to this problem.

[36] R. Gray. Agent Tcl: A transportable agent system. In J. Mayfield and T. Finnin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM'95)*, December 1995.

Describes an extension of Tcl [59] for mobile agents.

[37] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In DSL-99 [27], pages 39–52.

A language is presented for annotating C libraries with information that is exploited by an optimizing compiler. Domain-specific information is conveyed by annotations that in effect define (i) a dataflow analysis problem on the various library procedures, and (ii) procedure specializations that are to be triggered by the outcome of the analysis. The approach aims at giving libraries some of the compiler support enjoyed by DSLs.

[38] R. M. Herndon and V. A. Berzins. The realizable benefits of a language prototyping language. *IEEE Transactions on Software Engineering*, SE-14:803–809, 1988.

Discusses language prototyping tools (LPT) in general, as well as the specific LPT Kodiyak. Lists application areas of LPTs and benefits of applying them. Gives a brief description of Kodiyak and reports on experience with it.

[39] E. Horowitz, A. Kemper, and B. Narasimhan. A survey of application generators. *IEEE Software*, pages 40–54, January 1985.

Surveys a number of database query and update languages, as prime examples of application generators (DSL compilers), and hypothesizes a 'generic' database language. Discusses the possibilities of combining such a language with a general purpose language. Outlines AdaRel, an extension of Ada with relational database programming constructs.

[40] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es), December 1996.

Argues that a DSL is the "ultimate abstraction", capturing precisely the semantics of the application domain, but also that designing and implementing languages is difficult and resists evolution. Proposes the notion of embedded *DSLs, which inherit the infrastructure from some other language, and discusses the importance of modular monadic interpreters, instrumentation, and partial evaluation.*

[41] J. Jennings and E. Beuscher. Verischemelog: Verilog embedded in Scheme. In DSL-99 [27], pages 123–134.

Verilog, a digital hardware design language, is extended with facilities for generating and manipulating hardware descriptions by embedding it into the general purpose language Scheme. The extended language features early error detection and high customizability.

[42] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

Introduced the notion of object-oriented frameworks. A framework is defined as a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes. In a white-box *framework, application-specific behavior is obtained via method overriding or by adding new methods to the framework's* *classes. In a* black-box*, support for extensibility is provided by defining interfaces for components that can be plugged into the framework via object composition, thus better hiding the implementation details of the framework. See also [32, 66]*

[43] S. Kamin, editor. *DSL '97 – First ACM SIGPLAN Workshop on Domain-Specific Languages, in Association with POPL '97*, Paris, France, January 1997. University of Illinois Computer Science Report.

[44] S. Kamin and D. Hyatt. A special-purpose language for picture-drawing. In Ramming [64], pages 297–310.

Describes FPIC, a reconstruction of the original PIC embedded in ML.

[45] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

FODA is a domain engineering approach emphasizing feature analysis. A feature is defined as a prominent, user-visible characteristic of a software system. FODA aims at building up a feature model*, consisting of a features diagram (hierarchical decomposition of mandatory, alternative, or optional features), feature definitions, composition rules for features, and a rationale for features indicating the trade-offs. See also [20, 70]*

[46] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. Lopes, C. Maeda, and A. Mendhekar. Aspect oriented programming. In Kamin [43], pages 75–88.

Presents a novel programming technique, called aspect-oriented programming (AOP). This technique consists in describing each aspect (e.g. basic functionality, communication, coordination) of a system's behaviour in a (little) language that allows it to be expressed in its most natural form. An aspect weaver merges these separate aspect descriptions into a single, efficient program. An important benefit of AOP is that it allows high-level domain-specific programming for performance-critical domains. See also [20]

[47] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th International Conference on Software Engineering ICSE-18*, pages 542–553. IEEE, 1996.

Reports the results of an experiment in which a template-based approach and a DSL approach to software generation were compared. Several subjects were monitored while performing a number of development and maintenance tasks using alternatively template technology and DSL technology. Flexibility, productivity, reliability, and usability were measured. The DSL approach scored better on all counts.

[48] N. Klarlund and M. I. Schwartzbach. A domain-specific language for regular sets of strings and trees. In DSL-IEEE [28], pages 378–386. An earlier version appeared in [64].

Describes design and implementation of FIDO, a language to express large finite-state automata on large alphabets. Typical application is in verification and model checking.

[49] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

Categorizes, describes and compares existing approaches to software reuse, among which DSLs (or application generators). Compared to the other approaches DSLs reduce the intellectual effort required to obtain an executable system from its specification. Limited availability and difficulty of building DSLs of optimal specificity/generality are listed as disadvantages of DSLs.

[50] D. A. Ladd and J. C. Ramming. Two application languages in software production. In *USENIX Very High Level Languages Symposium Proceedings*, pages 169–178, October 1994.

Describes how PRL5, an application-oriented, declarative language used to maintain the integrity of databases in the AT&T 5ESS telecommunications switch, evolved from an earlier, imperative domain-specific language, PRL, which in turn replaced a combination of English and C. The constraint descriptions expressed in PRL5 can be used in more than one way, whereas a program to check constraints is useful only for performing that particular computation. A key lesson is that domain-specific languages should not be designed to describe computation, but to express useful facts from which one or more computations can be derived.

[51] D. Leijen and E. Meijer. Domain specific embedded compilers. In DSL-99 [27], pages 109–122.

Explains how a DSL (SQL is taken as example) can be embedded in Haskell by (i) coding an abstract syntax of the DSL as a Haskell datatype (ii) writing a code generator in Haskell that maps the abstract syntax to the concrete syntax, and (iii) making Haskell call an external server which compiles and executes the generated DSL code.

[52] F. van der Linden, editor. *Development and Evolution of Software Architectures for Product Families*, volume 1429 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

Proceedings of a workshop originating from the ESPRIT ARES project, which investigates software architectures for families of embedded systems.

[53] R. R. Macala, L. D. Sutckey, and D. C. Gross. Managing domain-specific product-line development. *IEEE Software*, 13, May 1996.

Describes recommendations and lessons learned from managing a reusability project at Boeing in the area of real-time training systems for flight crews. Product-line development separates the software-development process into two separate life cycles: domain engineering*, which aims to create reusable assets, and* application engineering*, which fields systems using those assets. Lessons learned include that product-line development demands careful strategic planning, a mature development process, and the ability to overcome organizational resistance.*

[54] N. Medvidovic and D. S. Rosenblum. Domains of concern in software architectures and architecture description languages. In Ramming [64], pages 199–212.

Gives a categorization of DSLs for describing software architectures.

[55] V. Menon and K. Pingali. A case for source-level transformations in MATLAB. In DSL-99 [27], pages 53–66.

Three kinds of source-to-source transformations for optimizing MATLAB programs are proposed and shown to be effective. The transformations yield performance benefits additional to those obtained by (optimizing) compilation, and may be useful for other DSLs that are high-level, untyped, and interpreted.

[56] L. Nakatani and M. Jones. Jargons and infocentrism. In Kamin [43], pages 59–74.

Describes and advocates the development of DSLs as jargons*: domain-specific extensions of a tiny common base language. According to a new programming paradigm (*infocentrism*) the application semantics for these jargons can be programmed by providing actions for the constructs specific to the jargon only; the traversal semantics is inherited from the base language. Because all jargons share the base syntax and semantics, it is easy to combine and reuse their definitions as well as their tools. The* InfoWiz *technology which supports the development of jargons is discussed.*

[57] L. H. Nakatani, M. A. Ardis, R. G. Olsen, and P. M. Pontrelli. Jargons for domain engineering. In DSL-99 [27], pages 15–24.

Discusses the use of jargons *(see [56]) in the domain of configuration control.*

[58] J. M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, SE-10(5):564–74, September 1984.

The Draco approach starts by capturing domain analysis information in a DSL. The objects and operations of this DSL are refined into various DSLs of lower levels of abstraction, and finally into executable languages. These refinements capture design information (implementation decisions). The Draco system supports the development and reuse of constellations of DSLs and refinements. It offers tactics for refinement selection as well as automatic consistency checking of the resulting system specification.

[59] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, March 1998.

Discusses scripting *languages, such as Perl, Tcl, and Visual Basic, which are designed for gluing applications, assuming the existence of a set of components that just need to be connected together. Emphasizes that scripting languages should be typeless and interpreted.*

[60] J. Peterson and G. Hager. Monadic robotics. In DSL-99 [27], pages 95–108.

Discusses the importance of monads *in the implementation of tasks in Frob (see [61]), which help to achieve modularity and reusability.*

[61] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. In *PADL'99*, volume 1551 of *LNCS*, pages 91–105, 1999.

Describes two domain-specific extensions of Haskell: Frob *a language for robot control and* Fran *a language for reactive animations.*

[62] P. Pfahler and U. Kastens. Language design and implementation by selection. In Kamin [43], pages 97–108.

A language design system is presented which allows a user to design a DSL by selecting language features from menus. After selection, an implementation of the DSL can be generated. The system relies on domain designers to provide a definition of the design space, as well as specification components for all possible language features.

[63] C. Pu, A. Black, C. Cowan, J. Walpole, and C. Consel. Microlanguages for operating system specialization. In Kamin [43], pages 49–57.

Discusses the use of DSLs in the domain of operating system specialization. A high-level DSL is envisioned to describe application behavior, which will be compiled into a low-level DSL describing customized operating system behavior.

[64] J. C. Ramming, editor. *Proceedings of the USENIX Conference on Domain-Specific Languages*, Berkeley, CA, October 15–17 1997. USENIX Association.

[65] J. Reichwein, G. Rothermel, and M. Burnett. Slicing spreadsheets: An integrated methodology for spreadsheet testing and debugging. In DSL-99 [27], pages 25–38.

Building on techniques for dynamic program slicing and program dicing, a fault localization technique for incremental spreadsheet debugging is developed. Using various kinds of visual clues, the technique is integrated into a spreadsheet environment.

[66] D. Roberts and R. Johnson. Evolve frameworks into domain-specific languages. In *3rd International Conference on Pattern Languages*, Allerton Park, Ill., September 1996.

Discusses 9 stages of framework development. An object-oriented framework evolves gradually, starting from three examples, moving via a white-box framework, component library, pluggable objects, to a black-box framework. The final, and most mature, stage is when the domain knowledge is sufficiently stable to merit the development of a domain-specific language or visual builder to access the framework.

[67] P.H. Salus, editor. *Little Languages*, volume III of *Handbook of Programming Languages*. MacMillan, 1998.

This book contains a collection of mostly reprints and only a few original papers describing DSLs. It contains, for instance, papers like Little Languages *(Bentley [7]),* A system for typesettting mathematics: EQN *(Kernighan and Cherry), and an overview of the Documenter's Workbench (Akkerhuis) covering TROFF and several DSLs for describing graphics, chemical formulae, and the like. Other chapters cover AWK, SED, SQL, TCL/TK, PERL and PYTHON. The most original papers are a survey of DSLs and domain-specific extension languages by Hudak and an elaborate description of* Little Music Languages *by Langston.*

[68] T. Sheard, Z. Benaissa, and E. Pasalic. DSL implementation using staging and monads. In DSL-99 [27], pages 81–94.

Discusses how the use of staging *(separating compile-time computations from run time ones) and* monads *(for capturing effects and actions of the target code) lead to a simple, reusable, controlable, and correct DSL methodology.*

[69] M. Simos. Organization domain modeling (ODM): Formalizing the core domain modeling life cycle. In M. Samadzeh and M. Zand, editors, *Proceedings of the Symposium on Software Reusability SSR'95*, pages 196–205, August 1995. ACM Software Engineering Notes.

Summarizes the key elements of the ODM domain engineering methodology. The full description is given in [70].

[70] M. Simos, D. Creps, C. Klinger, L. Levine, and D. Allemang. Organization domain modelling (ODM) guidebook version 2.0. Technical Report STARS-VC-A025/001/00, Synquiry Technologies, Inc, 1996.

A comprehensive description of the ODM approach to domain engineering. The three main ODM steps are: (1) plan *the domain, selecting objectives, stakeholders, and a set of* boundary decisions *to scope the domain. (2)* model *the domain, building a domain lexicon, and describing the concepts and features, as well as their commonalities and variabilities. (3) (optional)* engineer *an asset base of components by combining features and customers in novel ways. ODM emphasizes existing (legacy) software systems as valuable sources of domain knowledge. It takes the "domain as a set of systems" point of view, rather than the "domain as the real world" viewpoint.*

[71] E. G. Sirer and B. N. Bershad. Using production grammars in software testing. In DSL-99 [27], pages 1–14.

Describes lava*, a DSL for specifying* production grammars*. These are used to* generate *sentences over a language, for the purpose of testing tools implementing that language. Experience with* lava *demonstrates that a special purpose language for production grammars can bring high coverage, simplicity, manageability, and structure to the testing effort. Observe that the production grammar approach can also be used to for testing DSL-tools.*

[72] Y. Smaragdakis and D. Batory. DiSTiL: A transformation library for data structures. In Ramming [64], pages 257–270.

Describes DiSTiL, a DSL for describing container data structures in C, implemented on top of MicroSoft's Intentional Programming (IP) system.

[73] D. E. Stevenson and M. M. Fleck. Programming language support for digitized images or, the monsters in the closet. In Ramming [64], pages 271–284.

Describes the image manipulation language Envision, implemented as an extension of Scheme.

[74] J. M. Stichnoth and T. Gross. Code composition as an implementation language for compilers. In Ramming [64], pages 119–132.

Describes the ANSI C compiler framework Catacomb that supports code composition. By providing user-defined code templates (describing new language constructs such as parallel array assignment) and a fixed code composition mechanism inside the compiler, new constructs can be implemented in the same way as standard ones. See also [30].

[75] R. N. Taylor, W. Tracz, and L. Coglianese. Software development using domain-specific software architectures. *ACM SIGSOFT Software Engineering Notes*, 20(5):27–37, 1995.

Provides the material used for a course on DSSA, Domain-Specific Software Architectures, which aims at the reduction in time and cost of producing specific application systems within a supported domain. The paper covers key examples, architecture representation formalisms, domain engineering, and the DSSA process. See also [20, 70]

[76] S. A. Thibault, R. Marlet, and C. Consel. Domain-specific languages: From design to implementation application to video device drivers generation. In DSL-IEEE [28], pages 363–377. An earlier version appeared in [64].

A video card stores and displays images on a computer display. Each card is programmed by similar, but highly vendor-specific, instructions. The authors exploit this similarity by designing a DSL for specifying drivers for video cards in the context of the XFree86 implementation of X windows. This Graphic Adaptor Language is implemented in two stages: a C library provides a low level abstract machine that is used by an interpreter for the DSL. The Tempo partial evaluator for C is used to eliminate the overhead of interpretation and of the generality of the abstract machine. Includes a discussion of the merits of the DSL approach in this domain.

[77] D. C. Wang, A. W. Appel, J. L. Korn, and C. S. Serra. The Zephyr abstract syntax description language. In Ramming [64], pages 213–28.

Presents the Abstract Syntax Description Language (ASDL). Reports the implementation of a tool that converts ASDL descriptions into C, C++, Java, or ML code. The generated code defines data-structures corresponding to this abstract syntax as well as functions for reading and writing abstract terms to a standard flattened representation. ASDL has been used to respecify the compiler intermediate format SUIF.

[78] J. Withey. Investment analysis of software assets for product lines. Technical Report CMU/SEI-96-TR-010, Software Engineering Institute, 1996.

Presents a model for analyzing the expected benefits from investing in domain-specific software product lines. One of the key concepts is economy of scope*, which is a condition where fewer inputs (such as effort and time) are needed to produce a greater* variety *of outputs. By contrast, economy of* scale *is achieved where fewer inputs are needed to produce greater* quantities *of a single output.*