When and How to Develop Domain-Specific Languages

Marjan Mernik[1], Jan Heering[2], Anthony M. Sloane[3]
[1] University of Maribor, Slovenia, marjan.mernik@uni-mb.si
[2] CWI Amsterdam, The Netherlands, Jan.Heering@cwi.nl
[3] Macquarie University, Australia, asloane@ics.mq.edu.au

Abstract

Domain-specific languages (DSLs) are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application. DSL development is hard, requiring both domain knowledge and language development expertise. Few people have both. Not surprisingly, the decision to develop a DSL is often postponed indefinitely, if considered at all, and most DSLs never get beyond the application library stage.

Although many articles have been written on the development of particular DSLs, there is very limited literature on DSL development methodologies and many questions remain regarding when and how to develop a DSL. To aid the DSL developer, we identify patterns in the *decision*, *analysis*, *design*, and *implementation* phases of DSL development. Our patterns improve and extend earlier work on DSL design patterns. We also discuss domain analysis tools and language development systems that may help to speed up DSL development. Finally, we state a number of open problems.

1 Introduction

1.1 General

Many computer languages are *domain-specific* rather than general-purpose. Domain-specific languages (DSLs) are also called *application-oriented* [118], *special-purpose* [148, p. xix], *task-specific* [108], *specialized* [15, p. 17], or *application* [99] languages. So-called *fourth-generation languages* (4GLs) [99] are usually DSLs for database applications. *Little languages* are small DSLs that do not include many features found in general-purpose programming language (GPLs) [14, p. 715].

DSLs trade generality for expressiveness in a limited domain. By providing notations and constructs tailored toward a particular application domain, they

Table 1: Some widely used domain-specific languages.

DSL	Application domain	Level	
BNF	Syntax specification	n.a.	
Excel	Spreadsheets	57	(version 5)
HTML	Hypertext web pages	22	(version 3.0)
LaTeX	Typesetting	n.a.	
Make	Software building	15	
MATLAB	Technical computing	n.a.	
SQL	Database queries	25	
VHDL	Hardware design	17	
Java	General-purpose	6	(comparison only)

Table 2: Language level vs. productivity as measured in function points (FP).

Level	Productivity average per staff month (FP)
1–3	5–10
4–8	10–20
9–15	16–23
16–23	15–30
24–55	30–50
> 55	40–100

offer substantial gains in expressiveness and ease of use compared with GPLs for the domain in question, with corresponding gains in productivity and reduced maintenance costs. Also, by reducing the amount of domain and programming expertise needed, DSLs open up their application domain to a larger group of software developers compared to GPLs. Some widely used DSLs with their application domains are listed in Table 1. The third column gives the language level of each DSL as given in [79]. Language level is related to productivity as shown in Table 2, also from [79]. Apart from these examples, the benefits of DSLs have often been observed in practice and are supported by quantitative results such as those reported in [72, 11, 79, 89, 66], but their quantitative validation in general as well as in particular cases is hard and an important open problem. Therefore, the treatment of DSL development in this article will be largely qualitative.

The use of DSLs is by no means new. APT, a DSL for programming numerically controlled machine tools, was developed in 1957–1958 [116]. BNF, the well-known syntax specification formalism, dates back to 1959 [6]. Domain-specific visual languages (DSVLs), such as visual languages for hardware description and protocol specification, are important, but beyond the scope of this survey.

We will not give a definition of what constitutes an application domain and what does not. Some consider Cobol to be a DSL for business applications, but others would argue this is pushing the notion of application domain too far. Leaving matters of definition aside, it is natural to think of DSLs in terms of a gradual scale with very specialized DSLs such as BNF on the left and GPLs such as C++ on the right. (The language level measure of [79] is one attempt to quantify this scale.) On this scale, Cobol would be somewhere between BNF and C++, but much closer to the latter. Similarly, it is hard to tell if command languages like the Unix shell or scripting languages like Tcl are DSLs. Clearly, *domain-specificity is a matter of degree.*

In combination with an *application library*, any GPL can act as a DSL. The library's Application Programmers Interface (API) constitutes a domain-specific vocabulary of class, method, and function names that becomes available by object creation and method invocation to any GPL program using the library. This being the case, why were DSLs developed in the first place? Simply because they can offer domain-specificity in better ways:

- Appropriate or established *domain-specific notations* are usually beyond the limited user-definable operator notation offered by GPLs. A DSL offers appropriate domain-specific notations from the start. Their importance should not be underestimated as they are directly related to the productivity improvement associated with the use of DSLs.

- Appropriate *domain-specific constructs and abstractions* cannot always be mapped in a straightforward way to functions or objects that can be put in a library. Traversals and error handling are typical examples [19, 66, 27]. A GPL in combination with an application library can only express these constructs indirectly or in an awkward way. Again, a DSL would incorporate domain-specific constructs from the start.

- Use of a DSL offers possibilities for *analysis*, *verification*, *optimization*, *parallelization*, and *transformation* in terms of DSL constructs that would be much harder or unfeasible if a GPL had been used, because the GPL source code patterns involved are too complex or not well-defined.

- Unlike GPLs, DSLs *need not be executable.* There is no agreement on this in the DSL literature. For instance, the importance of non-executable DSLs is emphasized in [150], but DSLs are required to be executable in [48]. We discuss DSL executability in Section 1.2.

Despite their shortcomings, application libraries are formidable competitors to DSLs. It is probably fair to say that most DSLs never get beyond the application library stage. These are sometimes called *domain-specific embedded languages* (DSELs) [76]. Even with improved DSL development tools, application libraries will remain the most cost-effective solution in many cases, the more so since the advent of component technologies such as COM and CORBA [133] has further complicated the relative merits of DSLs and application libraries. For instance, Microsoft Excel's macro language is a DSL for spreadsheet applications which adds programmability to Excel's fundamental interactive mode.

Using COM, Excel's implementation has been restructured into an application library of COM components, thereby opening it up to GPLs such as C++, Java and Basic, which can access it through its COM interfaces. This process of componentization is called *automation* [31]. Unlike the Excel macro language, which by its very nature is limited to Excel functionality, GPLs are not. They can be used to write applications transcending Excel's boundaries by using components from other "automated" programs and COM libraries in addition to components from Excel itself.

In the remainder of this section we discuss DSL executability (Section 1.2), DSLs as enablers of reuse (Section 1.3), the scope of this article (Section 1.4), and DSL literature (Section 1.5).

1.2 Executability of DSLs

DSLs are executable in various ways and to various degrees, even to the point of being non-executable. Accordingly, depending on the character of the DSL in question, the corresponding programs are often more properly called *specifications*, *definitions*, or *descriptions*. We identify some points on the "DSL executability scale":

- DSL with well-defined execution semantics (Excel macro language, HTML).
- Input language of an *application generator* [33, 129]. Examples are ATMOL [51], a DSL for atmospheric modeling, and Hancock [19], a DSL for customer profiling. Such languages are also executable, but they usually have a more declarative character and a less well-defined execution semantics as far as the details of the generated applications are concerned. The application generator is a compiler for the DSL in question.
- DSL not primarily meant to be executable, but nevertheless useful for application generation. The syntax specification formalism BNF is an example of a DSL with a purely declarative character that can also act as an input language for a parser generator.
- DSL not meant to be executable. Examples are domain-specific data structure representations [150]. Just like their executable relatives, such non-executable DSLs may benefit from various kinds of tool support such as specialized editors, prettyprinters, consistency checkers, analyzers, and visualizers.

1.3 DSLs as enablers of reuse

The importance of DSLs can also be appreciated from the wider perspective of the construction of large software systems. In this context the primary contribution of DSLs is to enable reuse of software artifacts [17]. Among the types of artifacts that can be reused via DSLs are language grammars, source code, software designs, and domain abstractions. Later sections provide many examples of DSLs; here we mention a few from the perspective of reuse.

In his definitive survey of reuse [92], Krueger categorizes reuse approaches along the following dimensions: abstracting, selecting, specializing, and integrating. In particular, he identifies application generators as an important reuse category. As already noted, application generators often use a DSL as their input language, thereby enabling programmers to reuse semantic notions embodied in the DSL without having to perform a detailed domain analysis themselves. Examples include BDL [16] that generates software to control concurrent objects and Teapot [30] that produces implementations of cache coherence protocols. Krueger identifies definition of domain coverage and concepts as a difficult challenge for implementors of application generators. We identify patterns for domain analysis in this article.

DSLs also play a role in other reuse categories identified by Krueger. For example, software architectures are commonly reused when DSLs are employed because the application generator or compiler follows a standard design when producing code from a DSL input. For example, GAL [139] enables reuse of a standard architecture for video device drivers. DSLs implemented as application libraries clearly enable reuse of source code. Prominent examples are Haskell-based DSLs such as Fran [50]. DSLs can also be used for formal specification of software schemas. For example, Nowra [126] specifies software manufacturing processes and SSC [28] deals with subsystem composition.

Reuse may involve exploitation of an existing language grammar. For example, Hancock [19] piggybacks on C while SWUL [25] extends Java. Moreover, the success of XML for DSLs is largely based on reuse of its grammar for specific domains. Less formal language grammars may also be reused when notations used by domain experts, but not yet available in a computer language, are realized in a DSL. For example, Hawk [96] uses a textual form of an existing visual notation.

1.4 Scope of this article

There are no easy answers to the “when and how” question in the title of this article. The previously mentioned benefits of DSLs do not come for free:

- DSL development is hard, requiring both domain and language development expertise. Few people have both.
- DSL development techniques are more varied than those for GPLs, requiring careful consideration of the factors involved.
- Depending on the size of the user community, development of training material, language support, standardization, and maintenance may become serious and time-consuming issues.

These are not the only factors complicating the decision to develop a new DSL. Initially, it is often far from evident that a DSL might be useful or that developing a new one might be worthwhile. This may become clear only after a sizable investment in domain-specific software development using a GPL has been made. The concepts underlying a suitable DSL may emerge only after a lot of GPL programming has been done. In such cases, DSL development may be a key step in *software reengineering* or *software evolution* [13].

To aid the DSL developer, we provide a systematic survey of the many factors involved by identifying patterns in the *decision*, *analysis*, *design*, and *implementation* phases of DSL development (Section 2). Our patterns improve and extend earlier work on DSL design patterns, in particular [131]. This is discussed in Section 2.6. The DSL development process can be facilitated by using domain analysis tools and language development systems. These are surveyed in Section 3. Finally, conclusions and open problems are given in Section 4.

1.5 Literature

We give some general pointers to the DSL literature. More specific references are given at appropriate points throughout this article rather than in this section. Until recently, DSLs received relatively little attention in the computer science research community and there are few books on the subject. We mention [99], an exhaustive account of 4GLs, [18], a two-volume collection of articles on software reuse including DSL development and program generation, [108], which focuses on the role of DSLs in end-user programming, [117], a collection of articles on little languages (not all of them DSLs), and [9], which treats scripting languages (again, not all of them DSLs). Domain analysis, program generators, generative programming techniques, and intentional programming (IP) are treated in [43]. Domain analysis and the use of XML, DOM, XSLT, and related languages and tools to generate programs are discussed in [34]. Domain-specific language development is an important element of the software factories method [68].

Proceedings of recent workshops and conferences partly or exclusively devoted to DSLs are [83, 113, 49, 73, 74, 75, 97]. Several journals have published special issues on DSLs [152, 101, 102]. Many of the DSLs used as examples in this article were taken from these sources. A special issue on end-user development is [132]. A special issue on program generation, optimization, and platform adaptation is [106]. There are many workshops and conferences at least partly devoted to DSLs for a particular domain, for example, description of features of telecommunications and other software systems [61]. The annotated DSL bibliography [48] (78 items) has limited overlap with the references in this article because of our emphasis on general DSL development issues.

2 DSL Patterns

2.1 Pattern classification

The following DSL development phases can be distinguished: *decision*, *analysis*, *design*, *implementation*, and *deployment*. In practice, DSL development is not a simple sequential process, however. The decision process may be influenced by preliminary analysis, which in turn may have to supply answers to unforeseen questions arising during design, and design is often influenced by implementation considerations.

We associate classes of patterns with each of the above development phases except deployment, which is beyond the scope of this article. The decision phase corresponds to the "when"-part of DSL development, the other phases to the "how"-part. *Decision patterns* are common situations that potential developers may find themselves in for which successful DSLs have been developed in the past. In such situations, use of an existing DSL or development of a new one is a serious option. Similarly, *analysis patterns*, *design patterns*, and *implementation patterns* are common approaches to, respectively, domain analysis, DSL design, and DSL implementation. Patterns corresponding to different DSL development phases are independent. For a particular decision pattern virtually any analysis or design pattern can be chosen, and the same is true for design and implementation patterns. Patterns in the same class, on the other hand, need not be independent, but may have some overlap.

We discuss each development phase and the associated patterns in a separate section. Inevitably, there may be some patterns we have missed.

2.2 Decision

Deciding in favor of a new DSL is usually not easy. The investment in DSL development (including deployment) has to pay for itself by more economical software development and/or maintenance later on. As mentioned in Section 1.1, a quantitative treatment of the tradeoffs involved is difficult. In practice, short-term considerations and lack of expertise may easily cause indefinite postponement of the decision. Obviously, adopting an existing DSL is much less expensive and requires much less expertise than developing a new one. Finding out about available DSLs may be hard, since DSL information is scattered widely and often buried in obscure documents. Adopting DSLs that are not well-publicized might be considered too risky, anyway.

To aid in the decision process, we identify the decision patterns shown in Table 3. Underlying them are general, interrelated concerns such as:

- improved software economics,
- enabling of software development by users with less domain and programming expertise, or even by end-users with some domain but virtually no programming expertise [108, 132].

The patterns in Table 3 may be viewed as more concrete and specific subpatterns of these general concerns. We briefly discuss each decision pattern in turn. Examples for each pattern are given in Table 4.

Notation The availability of appropriate (new or existing) domain-specific notations is the decisive factor in this case. Two important subpatterns are:

Table 3: Decision patterns.

Pattern	Description
Notation	Add new or existing domain notation Important subpatterns: • Transform visual to textual notation • Add user-friendly notation to existing API
AVOPT	Domain-specific Analysis, Verification, Optimization, Parallelization, and Transformation
Task automation	Eliminate repetitive tasks
Product line	Specify member of software product line
Data structure representation	Facilitate data description
Data structure traversal	Facilitate complicated traversals
System front-end	Facilitate system configuration
Interaction	Make interaction programmable
GUI construction	Facilitate GUI construction

- **Transform visual to textual notation** There are many benefits to making an existing visual notation available in textual form, such as easier composition of large programs or specifications, and enabling of the AVOPT decision pattern discussed next.

- **Add user-friendly notation to an existing API** or "turn an API into a DSL".

AVOPT Domain-specific Analysis, Verification, Optimization, Parallelization, and Transformation of application programs written in a GPL are usually not feasible, because the source code patterns involved are too complex or not well-defined. Use of an appropriate DSL makes these operations possible. With continuing developments in chip-level multiprocessing (CMP), domain-specific parallelization will become steadily more important [93]. This pattern overlaps with most of the others.

Task automation Programmers often spend time on GPL programming tasks that are tedious and follow the same pattern. In such cases, the required code can be generated automatically by an application generator (compiler) for an appropriate DSL.

Product line Members of a software product line [147] share a common architecture and are developed from a common set of basic elements. Use of a DSL may often facilitate their specification. This pattern has considerable overlap with both the task automation and system front-end patterns.

Table 4: Examples for the decision patterns in Table 3.

Pattern	DSL	Application domain
Notation	MSC [123]	Telecom system specification
• Visual-to-textual	Hawk [96]	Microarchitecture design
	MSF [66]	Tool integration
	Verischemelog [78]	Hardware design
• API-to-DSL	SPL [153]	Digital signal processing
	SWUL [25]	GUI construction
AVOPT	AL [69]	Software optimization
	ATMOL [51]	Atmospheric modeling
	BDL [16]	Coordination
	ESP [94]	Programmable devices
	OWL-Light [44]	Web ontology
	PCSL [27]	Parameter checking
	PLAN-P [138]	Network programming
	Teapot [30]	Cache coherence protocols
Task automation	Facile [120]	Computer architecture
	JAMOOS [60]	Language processing
	lava [125]	Software testing
	PSL-DA [55]	Database applications
	RoTL [100]	Traffic control
	SHIFT [2]	Hybrid system design
	SODL [104]	Network applications
Product line	GAL [139]	Video device drivers
Data structure representation	ACML [63]	CASE tools
	ASDL [146]	Language processing
	DiSTiL [128]	Container data structures
	FIDO [91]	Tree automata
Data structure traversal	ASTLOG [42]	Language processing
	Hancock [19]	Customer profiling
	S-XML [35, 54]	XML processing
	TVL [66]	Tool integration
System front-end	Nowra [126]	Software configuration
	SSC [28]	Software composition
Interaction	CHEM [14]	Drawing chemical structures
	FPIC [85]	Picture drawing
	Fran [50]	Computer animation
	Mawl [3]	Web computing
	Service Combinators [29]	Web computing
GUI construction	AUI [121]	User interface construction
	HyCom [115]	Hypermedia applications

Table 5: Analysis patterns.

Pattern	Description
Informal	The domain is analyzed in an informal way.
Formal	A domain analysis methodology is used.
Extract from code	"Mining" of domain knowledge from legacy GPL code by inspection or by using software tools, or a combination of both.

Data structure representation Data-driven code relies on initialized data structures whose complexity may make them difficult to write and maintain. Such structures are often more easily expressed using a DSL.

Data structure traversal Traversals over complicated data structures can often be expressed better and more reliably in a suitable DSL.

System front-end A DSL based front-end may often be used for handling a system's configuration and adaptation.

Interaction Text or menu based interaction with application software often has to be supplemented with an appropriate DSL for the specification of complicated or repetitive input. For example, Excel's interactive mode is supplemented with the Excel macro language to make Excel "programmable".

GUI construction This is often done using a DSL.

2.3 Analysis

In the analysis phase of DSL development, the problem domain is identified and domain knowledge is gathered. Inputs are various sources of explicit or implicit domain knowledge, such as technical documents, knowledge provided by domain experts, existing GPL code, and customer surveys. The output of domain analysis varies widely, but consists basically of domain-specific terminology and semantics in more or less abstract form. There is a close link between domain analysis and *knowledge engineering*, which is only beginning to be explored. Knowledge capture, knowledge representation, and ontology development [45] are potentially useful in the analysis phase.

The analysis patterns we have identified are shown in Table 5. Examples are given in Table 6. Most of the time, domain analysis is done informally, but sometimes domain analysis methodologies are used. Examples of such methodologies are DARE (Domain Analysis and Reuse Environment) [57], DSSA (Domain-Specific Software Architectures) [134], FAST (Family-Oriented Abstractions,

Table 6: Examples for the analysis patterns in Table 5. References and application domains are given in Table 4. The FODA and FAST domain analysis methodologies are discussed in the text.

Pattern	DSL	Analysis methodology
Informal	All DSLs in Table 4 except:	
Formal	GAL	FAST commonality analysis
	Hancock	FAST
	RoTL	Variability analysis (close to FODA's)
	Service Combinators	FODA (only in this article—see text)
Extract from code	FPIC	Extracted by inspection from PIC implementation
	Nowra	Extracted by inspection from Odin implementation
	PCSL	Extracted by clone detection from proprietary C code
	Verischemelog	Extracted by inspection from Verilog implementation

Specification, and Translation) [147], FODA (Feature-Oriented Domain Analysis) [86], ODE (Ontology-based Domain Engineering) [53], or ODM (Organization Domain Modeling) [124]. To give an idea of the scope of these methods, we explain the FODA and FAST methodologies in somewhat greater detail. Tool support for formal domain analysis is discussed in Section 3.2.

The output of formal domain analysis is a *domain model* consisting of

- a domain definition defining the scope of the domain,
- domain terminology (vocabulary, ontology),
- descriptions of domain concepts,
- feature models describing the commonalities and variabilities of domain concepts and their interdependencies.

How can a DSL be developed from the information gathered in the analysis phase? No clear guidelines exist, but some are presented in [139, 137]. Variabilities indicate precisely what information is required to specify an instance of a system. This information must be specified directly in, or be derivable from, a DSL program. Terminology and concepts are used to guide the development of the actual DSL constructs corresponding to the variabilities. Commonalities are used to define the execution model (by a set of common operations) and primitives of the language. Note that the execution model of a DSL is usually much richer than that for a GPL. On the basis of a single domain analysis many different DSLs can be developed, but all share important characteristics found in the feature model.

For the sake of concreteness, we apply the FODA domain analysis methodology [86] to the service combinator DSL discussed in [29]. The latter's goal is to reproduce human behavior while accessing and manipulating web resources such as reaction to slow transmission, failures, and many simultaneous links. FODA requires construction of a feature model capturing commonalities (mandatory features) and variabilities (variable features). More specifically, such a model consists of

- a feature diagram representing a hierarchical decomposition of features and their character, that is, whether they are mandatory, alternative, or optional,
- definitions of the semantics of features,
- feature composition rules describing which combinations of features are valid or invalid,
- reasons for choosing a feature.

A common feature of a concept is a feature present in all instances of the concept. All mandatory features whose parent is the concept are common features. Also, all mandatory features whose parents are common are themselves common. A variable feature is either optional or alternative (one-of, more-of). Nodes in the feature diagram to which those features are attached are called variation points.

In the case of our example DSL, the domain consists of resources, browsing behavior, and services (type, status and rate). Resources can be atomic or compound, access to the resource (service) can be through a URL pointer or a gateway, and browsing behavior can be sequential, concurrent, repetitive, limited by accessing time, or rate. Service has a rate and status (succeeded, failed, or nonterminating). A corresponding feature diagram is shown in Figure 1. The first step in designing the DSL is to look into variabilities and commonalities in the feature diagram. Variable parts must be specified directly in or be derivable from DSL programs. It is clear that type of service (URL pointer or gateway) and browsing behavior have to be specified in DSL programs. Service status and service rate will be examined and computed while running a DSL program. Therefore, both will be built into the execution model. Type of resource (atomic or compound) are actually types of values that exist during the execution of a DSL program. The basic syntax proposed in [29]

```
S ::=  url(String)            // basic services
   |    gateway get (String+)
   |    gateway post (String+)
   |    index(String, String)
   |    S1  ?  S2             // sequential execution
   |    S1 '|' S2             // concurrent execution
```

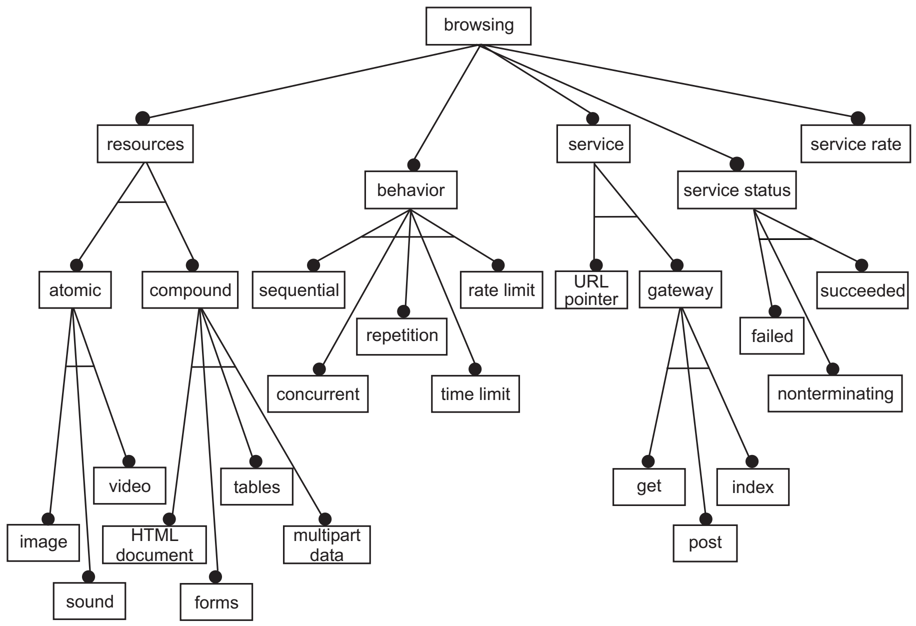

Figure 1: Feature diagram for web browsing.

```
|    timeout(Real, S)       // timeout combinator
|    limit(Real, Real, S)   // rate limit combinator
|    repeat(S)              // repetition
|    stall                  // nontermination
|    fail                   // failure
```

closely resembles our feature diagram. The syntax can later be extended with abstractions and binding.

Another domain analysis methodology is FAST (Family-Oriented Abstractions, Specification, and Translation) [37]. FAST is a software development process applying product line architecture principles, so it relates directly to the product line decision pattern. A common platform is specified for a family of software products. It is based on the similarities and differences between products. The FAST method consists of the following activities: domain qualification, domain engineering, application engineering, project management, and family change.

During domain engineering, the domain is analyzed and then implemented as a set of domain-specific reusable components. The purpose of domain analysis in FAST is to capture common knowledge about the domain and guide reuse of the implemented components. Domain analysis involves the following steps: decision model definition, commonality analysis, domain design, application modeling language design, create standard application engineering process

Table 7: Design patterns.

Pattern	Description
Language exploitation	DSL uses (part of) existing GPL or DSL Important subpatterns: • Piggyback: Existing language is partially used • Specialization: Existing language is restricted • Extension: Existing language is extended
Language invention	A DSL is designed from scratch with no commonality with existing languages
Informal	DSL is described informally
Formal	DSL is described formally using an existing semantics definition method such as attribute grammars, rewrite rules, or abstract state machines

design, and develop application engineering design environment. An important task of domain analysis is commonality analysis, which identifies useful abstractions that are common to all family members. Commonalities are the main source of reuse, thus the emphasis is on finding common parts. Besides the commonalities, variabilities are also discovered during commonality analysis. Variabilities indicate potential sources of change over the lifetime of the family. Commonalities and variabilities in FAST are specified as a structured list. For every variable property the range of variability as well as binding time are specified. Commonality analysis is later used in designing an application modeling language (AML), which is used to generate a family member from specifications.

2.4 Design

Approaches to DSL design can be characterized along two orthogonal dimensions: the relationship between the DSL and existing languages, and the formal nature of the design description. This dichotomy is reflected in the design patterns in Table 7 and the corresponding examples in Table 8.

The easiest way to design a DSL is to base it on an existing language. Possible benefits are easier implementation (see Section 2.5) and familiarity for users, but the latter only applies if users are also programmers in the existing language, which need not be the case. We identify three patterns of design based on an existing language. First, we can *piggyback* domain-specific features on part of an existing language. A related approach restricts the existing language to provide a *specialization* targeted at the problem domain. The difference between these two patterns is really a matter of how rigid the barrier is between the DSL and the rest of the existing language. Both of these approaches are often used when a notation is already widely known. For example, many DSLs contain arithmetic expressions which are usually written in the infix-operator style of

Table 8: Examples for the design patterns in Table 7. References and application domains are given in Table 4.

Pattern	DSL
Language exploitation	
• Piggyback	ACML, ASDL, BDL, ESP, Facile, Hancock, JAMOOS, lava, Mawl, PSL-DA, SPL, SSC, Teapot
• Specialization	OWL-Light
• Extension	AUI, DiSTiL, FPIC, Fran, Hawk, HyCom, Nowra, PLAN-P, SWUL, S-XML, Verischemelog
Language invention	AL, ASTLOG, ATMOL, CHEM, GAL, FIDO, MSF, RoTL, Service Combinators, SHIFT, SODL, TVL
Informal	All DSLs in Table 4 except:
Formal	ATMOL, ASTLOG, BDL, FIDO, GAL, OWL-Light, PLAN-P, RoTL, Service Combinators, SHIFT, SODL, SSC

mathematics.

Another approach is to take an existing language and extend it with new features that address domain concepts. In most applications of this pattern the existing language features remain available. The challenge is to integrate the domain-specific features with the rest of the language in a seamless fashion.

At the other end of the spectrum is a DSL whose design bears no relationship to any existing language. In practice, development of this kind of DSL can be extremely difficult and is hard to characterize. Well-known GPL design criteria such as readability, simplicity, orthogonality, the design principles listed by Brooks [26], and Tennent's design principles [135] retain some validity for DSLs. However, the DSL designer has to keep in mind both the special character of DSLs as well as the fact that users need not be programmers. Since ideally the DSL adopts established notations of the domain, the designer should suppress a tendency to improve them. As stated in [151], one of the lessons learned from real DSL experiments is:

> **Lesson T2:** You are almost never designing a programming language.
> Most DSL designers come from language design backgrounds. There the admirable principles of orthogonality and economy of form are not necessarily well-applied to DSL design. Especially in catering to the pre-existing jargon and notations of the domain, one must be careful not to embellish or over-generalize the language.
> **Lesson T2 Corollary:** Design only what is necessary. Learn to recognize your tendency to over-design.

Once the relationship to existing languages has been determined, a DSL designer must turn to specifying the design before implementation. We distinguish

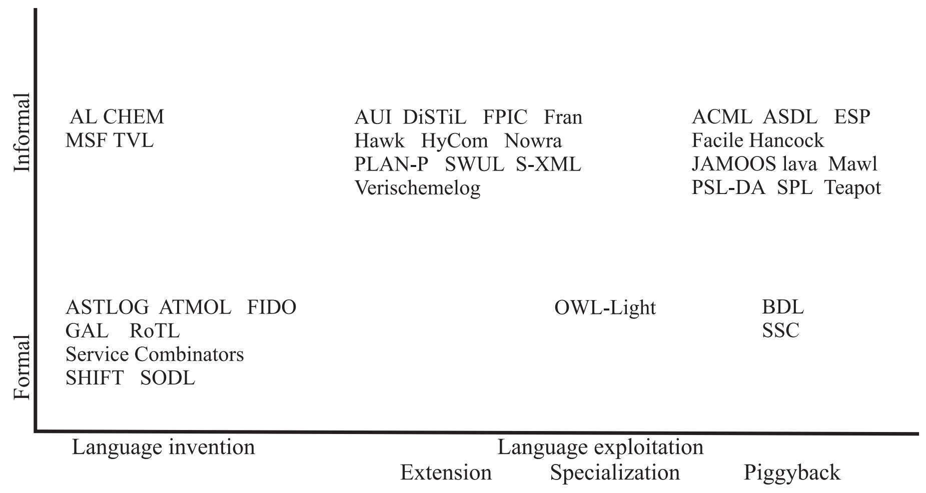

Figure 2: The DSLs from Table 8 in the design pattern plane.

between *informal* and *formal* designs. In an informal design the specification is usually in some form of natural language probably including a set of illustrative DSL programs. A formal design would consist of a specification written using one of the available semantic definition methods [127]. The most widely used formal notations include regular expressions and grammars for syntax specifications, and attribute grammars, rewrite systems and abstract state machines for semantic specification.

Clearly, an informal approach is likely to be easiest for most people. A formal approach should not be discounted, however. Development of a formal description of both syntax and semantics can bring problems to light before the DSL is actually implemented. Furthermore, formal designs can be implemented automatically by language development systems and tools, thereby significantly reducing implementation effort (Section 3).

As mentioned in the beginning of this section, design patterns can be characterized in terms of two orthogonal dimensions: language invention or language exploitation (extension, specialization, or piggyback), and informal or formal description. Figure 2 indicates the position of the DSLs from Table 8 in the design pattern plane. We note that formal description is used more often than informal description when a DSL is designed using the language invention pattern. The opposite is true when a DSL is designed using language exploitation.

2.5 Implementation

2.5.1 Patterns

When an (executable) DSL is designed, the most suitable implementation approach should be chosen. This may be obvious, but in practice it is not, mainly

Table 9: Implementation patterns for executable DSLs.

Pattern	Description
Interpreter	DSL constructs are recognized and interpreted using a standard fetch-decode-execute cycle. This approach is appropriate for languages having a dynamic character or if execution speed is not an issue. The advantages of interpretation over compilation are greater simplicity, greater control over the execution environment, and easier extension.
Compiler/application generator	DSL constructs are translated to base language constructs and library calls. A complete static analysis can be done on the DSL program/specification. DSL compilers are often called application generators.
Preprocessor	DSL constructs are translated to constructs in an existing language (the *base language*). Static analysis is limited to that done by the base language processor. Important subpatterns: • Macro processing: Expansion of macro definitions. • Source-to-source transformation: DSL source code is transformed (translated) into base language source code. • Pipeline: Processors successively handling sublanguages of a DSL and translating them to the input language of the next stage. • Lexical processing: Only simple lexical scanning is required, without complicated tree-based syntax analysis.
Embedding	DSL constructs are embedded in an existing GPL (the *host language*) by defining new abstract data types and operators. Application libraries are the basic form of embedding.
Extensible compiler/ interpreter	A GPL compiler/interpreter is extended with domain-specific optimization rules and/or domain-specific code generation. While interpreters are usually relatively easy to extend, extending compilers is hard unless they were designed with extension in mind.
Commercial Off-The-Shelf (COTS)	Existing tools and/or notations are applied to a specific domain.
Hybrid	A combination of the above approaches.

Table 10: Examples for the implementation patterns in Table 9. References and application domains are given in Table 4.

Pattern	DSL
Interpreter	ASTLOG, Service Combinators
Compiler/application generator	AL, ATMOL, BDL, ESP, Facile, FIDO, Hancock, JAMOOS, lava, Mawl, PSL-DA, RoTL, SHIFT, SODL, SPL, Teapot
Preprocessor	
• Macro processing	S-XML
• Source-to-source transformation	ADSL, AUI, MSF, SWUL, TVL
• Pipeline	CHEM
• Lexical processing	SSC
Embedding	FPIC, Fran, Hawk, HyCom, Nowra, Verischemelog
Extensible compiler/interpreter	DiSTiL
Commercial Off-The-Shelf (COTS)	ACML, OWL-Light
Hybrid	GAL, PLAN-P

because of the many DSL implementation techniques that have no useful counterpart for GPLs. These DSL-specific techniques are less well-known, but can make a big difference in the total effort that has to be invested in DSL development. The implementation patterns we have identified are shown in Table 9. We discuss some of them in more detail. Examples are given in Table 10.

Interpretation and compilation are as relevant for DSLs as for GPLs, even though the special character of DSLs often makes them amenable to other, more efficient, implementation methods, such as preprocessing and embedding. This viewpoint is at variance with [131], where it is argued that DSL development is radically different from GPL development, since the former is usually just a small part of a project and hence DSL development costs have to be modest. This is not always the case, however, and interpreters and compilers/application generators are widely used in practice.

Macros and subroutines are the classical language extension mechanisms used for DSL implementation. Subroutines have given rise to implementation by embedding, while macros are handled by preprocessing. A recent survey of macros is given in [22]. Macro expansion is often independent of the syntax of the base language and the syntactical correctness of the expanded result is not guaranteed, but is checked at a later stage by the interpreter or compiler. This situation is typical for preprocessors.

C++ supports a language-specific preprocessing approach: *template metaprogramming* [143, 142]. It uses template expansion to achieve compile-time generation of domain-specific code. Significant mileage has been made out of template metaprogramming to develop mathematical libraries for C++ which have familiar domain notation using C++ user-definable operator notation and overloading, but also achieve good performance. An example is Blitz++ [144].

In the embedding approach, a DSL is implemented by extending an existing GPL (the *host language*) by defining specific abstract data types and operators. A domain-specific problem then can be described with these new constructs. Therefore, the new language has all the power of the host language, but an application engineer can become a programmer without learning too much of it. To approximate domain-specific notations as closely as possible, the embedding approach can use any features for user-definable operator syntax the host language has to offer. For example, it is common to develop C++ class libraries where the existing operators are overloaded with domain-specific semantics. Although this technique is quite powerful, pitfalls exist in overloading familiar operators to have unfamiliar semantics. Although the host language in the embedding approach can be any general-purpose language, functional languages are often appropriate, as shown by many researchers [77, 84]. This is due to functional language features such as lazy evaluation, higher-order functions, and strong typing with polymorphism and overloading.

Extending an existing language implementation can also be seen as a form of embedding. The difference is usually a matter of degree. In an interpreter or compiler approach the implementation would usually only be extended with a few features, such as new data types and operators for them. For a proper embedding, the extensions might encompass full-blown domain-specific language features. In both settings, however, extending implementations is often very difficult. Techniques for doing so in a safe and modular fashion are still the subject of much research. Since compilers are particularly hard to extend, much of this work is aimed at preprocessors and extensible compilers allowing addition of domain-specific optimization rules and/or domain-specific code generation. We mention user-definable optimization rules in the CodeBoost C++ preprocessor [8] and in the Simplicissimus GCC compiler plug-in [122], the IBM Montana extensible C++ programming environment [130], user-definable optimization rules in the GHC Haskell compiler [111], and exploitation of domain-specific semantics of application libraries in the Broadway compiler [70]. Some extensible compilers, such as OpenC++ [32], support a *metaobject protocol.* This is an object-oriented interface for specifying language extensions and transformations [88].

The COTS-based approach builds a DSL around existing tools and notations. Typically this approach involves applying existing functionality in a restricted way, according to domain rules. For example, the general-purpose Powerpoint tool has been applied in a domain-specific setting for diagram editing [150]. The current prominence of XML-based DSLs is another instance of this approach [63, 110]. For an XML-based DSL, grammar is described using a DTD or XML scheme, where nonterminals are analogous to elements and terminals to data content. Productions are like element definitions, where the element name is the left-hand side and the content model is the right-hand side. The start symbol is analogous to the document element in a DTD. Using a DOM parser or SAX (Simple API for XML) tool, parsing comes for free. Since the parse tree can be encoded in XML as well, XSLT transformations can be used for code generation. Therefore, XML and XML tools can be used to implement a programming language compiler [59].

Many DSL endeavors apply a number of these approaches in a hybrid fashion. Thus the advantages of different approaches can be exploited. For instance, embedding can be combined with user-defined domain-specific optimization in an extensible compiler, and the interpreter and compiler approach become indistinguishable in some settings (see next section).

2.5.2 Implementation trade-offs

Advantages of the interpreter and compiler/application generator approaches are:

- DSL syntax can be close to the notations used by domain experts,
- good error reporting is possible,
- domain-specific analysis, verification, optimization, parallelization, and transformation (AVOPT) is possible,

while some of its disadvantages are:

- the development effort is high because a complex language processor must be implemented,
- the DSL is more likely to be designed from scratch, often leading to incoherent designs compared with exploitation of an existing language,
- language extension is hard to realize because most language processors are not designed with extension in mind.

However, these disadvantages can be minimized or eliminated altogether when a language development system or toolkit is used, so that much of the work of language processor construction is automated. This presupposes a formal approach to DSL design and implementation. Automation support is discussed further in Section 3.

We now turn to the embedded approach. Its advantages are:

- development effort is modest because an existing implementation can be reused,
- it often produces a more powerful language than other methods since many features come for free,
- host language infrastructure can be reused (development and debugging environments: editors, debuggers, tracers, profilers etc.),
- user training costs might be lower since many users may already know the host language.

Disadvantages of the embedded approach are:

- syntax is far from optimal because most languages do not allow arbitrary syntax extension,
- overloading existing operators can be confusing if the new semantics does not have the same properties as the old,
- bad error reporting because messages are in terms of host language concepts instead of DSL concepts,
- domain-specific optimizations and transformations are hard to achieve, so efficiency may be affected, particularly when embedding in functional languages [84, 126].

Advocates of the embedded approach often criticize DSLs implemented by the interpreter or compiler approach in that too much effort is put into syntax design, whereas the language semantics tends to be poorly designed and cannot be easily extended with new features [84]. However, the syntax of a DSL is extremely important and should not be underestimated. It should be as close as possible to the notation used in a domain.

In the functional setting, and in particular if Haskell is used, some of these shortcomings can be reduced by using *monads* to modularize the language implementation [77]. Domain-specific optimizations can be achieved using approaches such as user-defined transformation rules in the GHC compiler [111] or a form of whole-program transformation called *partial evaluation* [80, 36]. In C++, template metaprogramming can be used and user-defined domain-specific optimization is supported by various preprocessors and compilers. See the references in Section 2.5.1.

The decision diagram on how to proceed with DSL implementation (Figure 3) shows when a particular implementation approach is more appropriate. If the DSL is designed from scratch with no commonality with existing languages (invention pattern), the recommended approach is to implement it by embedding, unless domain-specific analysis, verification, optimization, parallelization, or transformation (AVOPT) is required, a domain-specific notation must be strictly obeyed, or the user community is expected to be large.

If the DSL incorporates (part of) an existing language, one would like to reuse (the corresponding part of) the existing language's implementation as well. Apart from this, various implementation patterns may apply, depending on the language exploitation subpattern used. A piggyback or specialization design can be implemented using an interpreter, compiler/application generator, or preprocessor, but embedding or use of an extensible compiler/interpreter are not suitable, although specialization can be done using an extensible compiler/interpreter in some languages (Smalltalk, for instance). In the case of piggyback, a preprocessor transforming the DSL to the language it piggybacks on is best from the viewpoint of implementation reuse, but preprocessing has serious shortcomings in other respects. A language extension design can be implemented using all of the above-mentioned implementation patterns. From

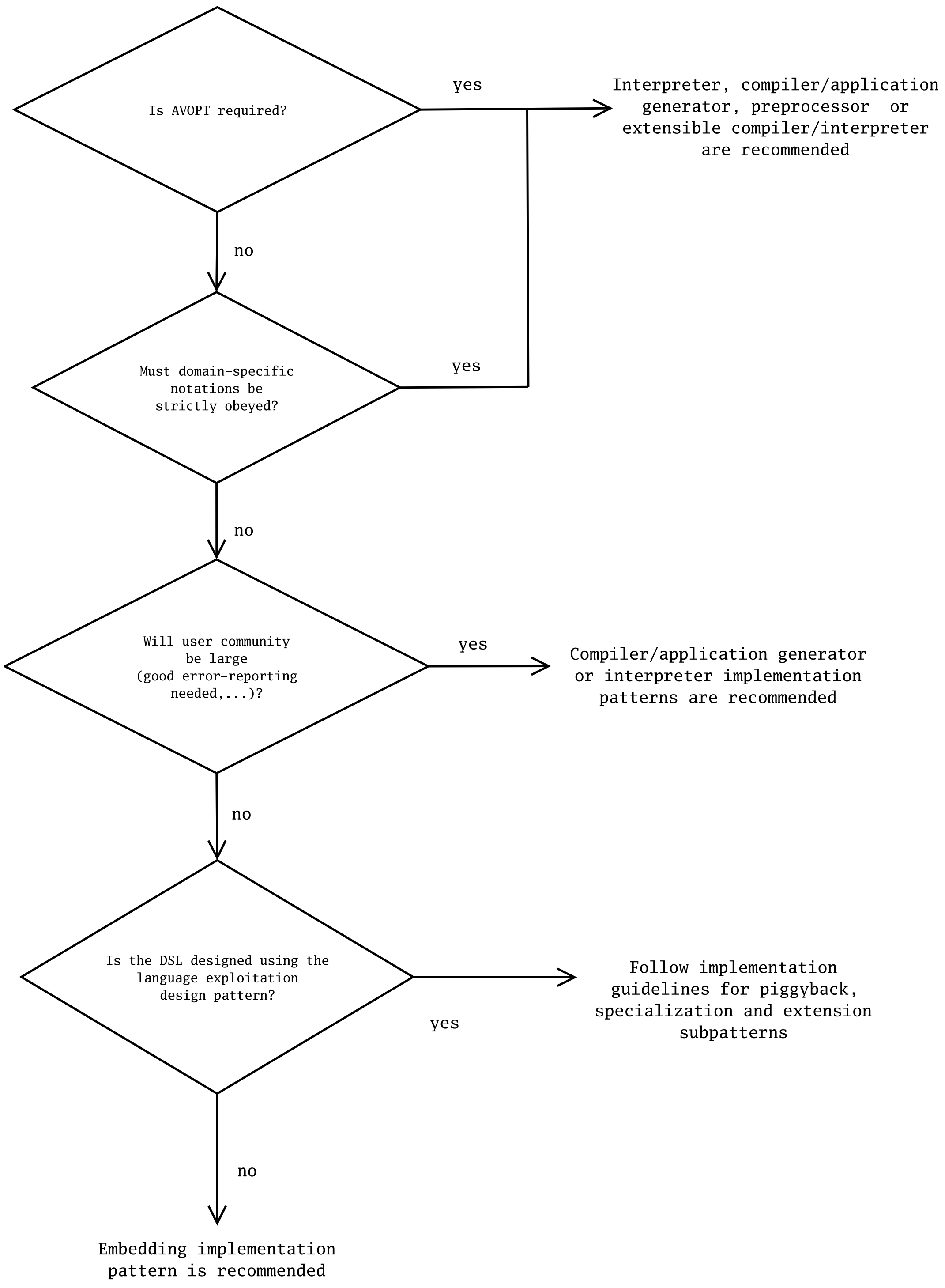
Is AVOPT required?
yes
Interpreter, compiler/application generator, preprocessor or extensible compiler/interpreter are recommended
no
Must domain-specific notations be strictly obeyed?
yes
no
Will user community be large (good error-reporting needed,...)?
yes
Compiler/application generator or interpreter implementation patterns are recommended
no
Is the DSL designed using the language exploitation design pattern?
yes
Follow implementation guidelines for piggyback, specialization and extension subpatterns
no
Embedding implementation pattern is recommended

Figure 3: Implementation guidelines.

the viewpoint of implementation reuse, embedding and use of an extensible compiler/interpreter are particularly attractive in this case.

If more than one implementation pattern applies, the one having the highest ratio of benefit (see discussion in this section) to implementation effort is optimal, unless, as in the language invention case, AVOPT is required, a domain-specific notation must be strictly obeyed, or the user community is expected to be large. As already mentioned, a compiler/application generator scores worst in terms of implementation effort. Less costly are (in descending order): interpreter, preprocessing, extensible compiler/interpreter, and embedding. On the other hand, a compiler/application generator and interpreter score best as far as benefit to DSL users is concerned. Less benefit is obtained from (in descending order): extensible compiler/interpreter, embedding, and preprocessing. In practice, such a cost-benefit analysis is rarely performed, and the decision is driven only by implementor experience. Of course, the latter should be taken into account, but it is not the only relevant factor.

2.6 Comparison with other classifications

We start by comparing our patterns with those proposed in [131]. Closely following [58], Spinellis distinguishes three classes of DSL patterns as shown in Table 11. The specific patterns for each class are summarized in Tables 12, 13, and 14. Most patterns are creational. The piggyback pattern might be classified as creational as well, since it is very similar to language extension. This would leave only a single pattern in each of the other two categories.

First, it should be noted that Spinellis's patterns do not include traditional GPL design and implementation techniques, while ours do, since we consider them to be as relevant for DSLs as for GPLs. Second, Spinellis's classification does not correspond in an obvious way to our classification in decision, analysis, design, and implementation patterns. The latter are all basically creational, but covering a wider range of creation-related activities than Spinellis's patterns.

The correspondence of Spinellis's patterns with ours is shown in Table 15. Since our patterns have a wider scope, many of them have no counterpart in Spinellis's classification. These are not shown in the right-hand column. We have retained the terminology used by Spinellis whenever appropriate.

Another classification of DSL development approaches is given in [150], namely, full language design, language extension, and COTS-based approaches. Since each approach has its own pros and cons, the author discusses them with respect to three kinds of issues: DSL specific, GPL support, and pragmatic support issues. Finally, the author shows how a hybrid development approach can be used.

Table 11: Pattern classification proposed by Spinellis.

Pattern class	Description
Creational pattern	DSL creation
Structural pattern	Structure of system involving a DSL
Behavioral pattern	DSL interactions

Table 12: Creational patterns.

Pattern	Description
Language extension	DSL extends existing language with new datatypes, new semantic elements, and/or new syntax.
Language specialization	DSL restricts existing language for purposes of safety, static checking, and/or optimization.
Source-to-source transformation	DSL source code is transformed (translated) into source code of existing language (the base language).
Data structure representation	Data-driven code relies on initialized data structures whose complexity may make them difficult to write and maintain. These structures are often more easily expressed using a DSL.
Lexical processing	Many DSLs may be designed in a form suitable for recognition by simple lexical scanning.

Table 13: Structural patterns.

Pattern	Description
Piggyback	DSL has elements, for instance, expressions in common with existing language. DSL processor passes those elements to existing language processor.
System front-end	A DSL based front-end may often be used for handling a system's configuration and adaptation.

Table 14: Behavioral patterns.

Pattern	Description
Pipeline	Pipelined processors successively handling sublanguages of a DSL and translating them to input language of next stage.

Table 15: Correspondence of Spinellis's patterns with ours. Since our patterns have a wider scope, many of them have no counterpart in Spinellis's classification. These are not shown in the right-hand column.

Spinellis's pattern	Our pattern
Creational: language extension	Design: language exploitation (extension)
Creational: language specialization	Design: language exploitation (specialization)
Creational: source-to-source transformation	Implementation: preprocessing (source-to-source transformation)
Creational: data structure representation	Decision: data structure representation
Creational: lexical processing	Implementation: preprocessing
Structural: piggyback	Design: language exploitation (piggyback)
Structural: system front-end	Decision: system front-end
Behavioral: pipeline	Implementation: preprocessing (pipeline)

3 DSL Development Support

3.1 Design and implementation support

As we have seen, DSL development is hard, requiring both domain knowledge and language development expertise. The development process can be facilitated by using a *language development system* or *toolkit*. Some systems and toolkits that have actually been used for DSL development are listed in Table 16. They have widely different capabilities and are in widely different stages of development, but are based on the same general principle: *they generate tools from language descriptions* [71]. The tools generated may vary from a consistency checker and interpreter to an integrated development environment (IDE) consisting of a syntax-directed editor, a prettyprinter, an (incremental) consistency checker, analysis tools, an interpreter or compiler/application generator, and a debugger for the DSL in question (assuming it is executable). As noted in Section 1.2, non-executable DSLs may also benefit from various kinds of tool support such as syntax-directed editors, prettyprinters, consistency checkers, and analyzers. These can be generated in the same way.

Some of these systems support a specific DSL design methodology, while others have a largely methodology-independent character. For instance, Sprint [36] assumes an interpreter for the DSL to be given and then uses partial evaluation to remove the interpretation overhead by automatically transforming a DSL program into a compiled program. Other systems, such as ASF+SDF [23],

Table 16: Some language development systems and toolkits that have been used for DSL development.

System	Developer
ASF+SDF [23]	CWI/University of Amsterdam
AsmL [62]	Microsoft Research, Redmond
DMS [12]	Semantic Designs, Inc.
Draco [109]	University of California, Irvine
Eli [67]	University of Colorado, University of Paderborn, Macquarie University
Gem-Mex [1]	University of L'Aquila
InfoWiz [107]	Bell Labs/AT&T Labs
JTS [10]	University of Texas at Austin
Khepera [52]	University of North Carolina
Kodiyak [72]	University of Minnesota
LaCon [87]	University of Paderborn (LaCon uses Eli as back-end — see above)
LISA [105]	University of Maribor
`metafront` [21]	University of Aarhus
Metatool [33]	Bell Labs
POPART [149]	USC/Information Sciences Institute
SmartTools [4]	INRIA Sophia Antipolis
`smgn` [90]	Intel Compiler Lab/University of Victoria
SPARK [5]	University of Calgary
Sprint [36]	LaBRI/INRIA
Stratego [145]	University of Utrecht
TXL [38]	University of Toronto/Queen's University at Kingston

Table 17: Development support provided by current language development systems and toolkits for DSL development phases/pattern classes.

Development phase/ Pattern class	Support provided
Decision	None
Analysis	Not yet integrated — see Section 3.2
Design	Weak
Implementation	Strong

Table 18: Examples of DSL development using the systems in Table 16.

System used	DSL	Application domain
ASF+SDF	Box [24]	Prettyprinting
	Risla [46]	Financial products
AsmL	UPnP [141]	Networked device protocol
	XLANG [136]	Business protocols
DMS	(Various) [12]	Program transformation
	(Various) [12]	Factory control
Eli	Maptool [82]	Grammar mapping
	(Various) [112]	Class generation
Gem-Mex	Cubix [95]	Virtual data warehousing
JTS	Jak [10]	Syntactic transformation
LaCon	(Various) [87]	Data model translation
LISA	SODL [104]	Network applications
SmartTools	LML [110]	GUI programming
	BPEL [39]	Business process description
`smgn`	Hoof [90]	Compiler IR specification
	IMDL [90]	Software reengineering
SPARK	Guide [98]	Web programming
	CML2 [114]	Linux kernel configuration
Sprint	GAL [139]	Video device drivers
	PLAN-P [138]	Network programming
Stratego	Autobundle [81]	Software building
	CodeBoost [8]	Domain-specific C++ optimization

DMS [12], and Stratego [145], would not only allow an interpretive definition of the DSL, but would also accept a transformational or translational one. On the other hand, they might not support partial evaluation of a DSL interpreter given a specific program.

The input to these systems is a description of various aspects of the DSL to be developed in terms of specialized metalanguages. Depending on the type of DSL, some important language aspects are *syntax*, *prettyprinting*, *consistency checking*, *analysis*, *execution*, *translation*, *transformation*, and *debugging*. It so happens that the metalanguages used for describing these aspects are themselves DSLs for the particular aspect in question. For instance, DSL syntax is usually described in something close to BNF, the *de facto* standard for syntax specification (Table 1). The corresponding tool generated by the language development system is a parser.

Although the various specialized metalanguages used for describing language aspects differ from system to system, they are often (but not always) *rule based*. For instance, depending on the system, the consistency of programs or scripts may have to be checked in terms of *attributed syntax rules* (an extension of BNF), *conditional rewrite rules*, or *transition rules*. See, for instance, [127] for further details.

The level of support provided by these systems in various phases of DSL development is summarized in Table 17. Their main strength lies in the implementation phase. Support of DSL design tends to be weak. Their main assets are the metalanguages they support, and in some cases a meta-environment to aid in constructing and debugging language descriptions, but they have little built-in knowledge of language concepts or design rules. Furthermore, to the best of our knowledge, none of them provides any support in the analysis or decision phase. Analysis support tools are discussed in Section 3.2.

Examples of DSL development using the systems in Table 16 are given in Table 18. They cover a wide range of application domains and implementation patterns. The Box prettyprinting metalanguage is an example of a DSL developed with a language development system (in this case ASF+SDF) for later use as one of the metalanguages of the system itself. This is common practice. The metalanguages for syntax, prettyprinting, attribute evaluation, and program transformation used by DMS were all implemented using DMS, and the Jak transformational metalanguage for specifying the semantics of a DSL or domain-specific language extension in the Jakarta Tool Suite (JTS) was also developed using JTS.

3.2 Analysis support

The language development toolkits and systems discussed in the previous section do not provide support in the analysis phase of DSL development. Separate frameworks and tools for this have been or are being developed, however. Some of them are listed in Table 19. We have included a short description of each entry, largely taken from the reference given for it. The fact that a framework or tool is listed does not necessarily mean it is in use or even exists.

Table 19: Some domain analysis frameworks and tools.

Analysis framework or tool	Description
Ariadne [124]	ODM support framework enabling domain practitioners to collaboratively develop and evolve their own semantic models, and to compose and customize applications incorporating these models as first-class architectural elements.
DARE [57]	Supports the capture of domain information from experts, documents, and code in a domain. Captured domain information is stored in a domain book that will typically contain a generic architecture for the domain and domain-specific reusable components.
DOMAIN [140]	DSSA [134] support framework consisting of a collection of structured editors and a hypertext/media engine that allows the user to capture, represent, and manipulate various types of domain knowledge in a hyper-web. DOMAIN supports a "scenario-based" approach to domain analysis. Users enter scenarios describing the functions performed by applications in the domain of interest. The text in these scenarios can then be used (in a semi-automated manner) to develop a domain dictionary, reference requirements, and domain model, each of which are supported by their own editor.
FDL [47]	The Feature Description Language (FDL) is a textual representation of feature diagrams, which are a graphical notation for expressing assertions (propositions, predicates) about systems in a particular application domain. These were introduced in the FODA [86] domain analysis methodology. (FDL is an example of the visual-to-textual transformation subpattern in Table 3.)
ODE editor [53]	Ontology editor supporting ODE — see also [45].

Table 20: Summary of DSL development phases and corresponding patterns.

Development phase	Pattern
Decision	Notation
(Section 2.2)	AVOPT
	Task automation
	Product line
	Data structure representation
	Data structure traversal
	System front-end
	Interaction
	GUI construction
Analysis	Informal
(Section 2.3)	Formal
	Extract from code
Design	Language exploitation
(Section 2.4)	Language invention
	Informal
	Formal
Implementation	Interpreter
(Section 2.5)	Compiler/application generator
	Preprocessor
	Embedding
	Extensible compiler/interpreter
	COTS
	Hybrid

As noted in Section 2.3, the output of domain analysis consists basically of domain-specific terminology and semantics in more or less abstract form. It may range from a feature diagram (see FDL entry in Table 19) to a domain implementation consisting of a set of domain-specific reusable components (see DARE entry in Table 19), or a theory in the case of scientific domains. An important issue is how to link formal domain analysis with DSL design and implementation. The possibility of linking DARE directly to the Metatool meta-generator (that is, application generator generator) [33] is mentioned in [56].

4 Conclusions and Open Problems

DSLs will never be a solution to all software engineering problems, but their application is currently unduly limited by a lack of reliable knowledge available to (potential) DSL developers. To help remedy this situation, we distinguished five phases of DSL development and identified patterns in each phase, except deployment. These are summarized in Table 20. Furthermore, we discussed language development systems and toolkits that can be used to facilitate the development process, especially its later phases.

Our survey also showed many opportunities for further work. As indicated in Table 17, for instance, there are serious gaps in the DSL development support chain. More specifically, some of the issues needing further attention are:

Decision Can useful computer-aided decision support be provided? If so, its integration in existing language development systems or toolkits (Table 16) might yield additional advantages.

Analysis Further development and integration of domain analysis support tools. As noted in Section 2.3, there is a close link with knowledge engineering. Existing knowledge engineering tools and frameworks may be useful directly or act as inspiration for further developments in this area. An important issue is how to link formal domain analysis with DSL design and implementation.

Design and implementation How can DSL design and implementation be made easier for domain experts not versed in GPL development? Some approaches are (not mutually exclusive):

- Building DSLs in an incremental, modular, and extensible way from *parameterized language building blocks*. This is of particular importance for DSLs, since they change more frequently than GPLs [20, 150]. Progress in this direction is being made [1, 36, 77, 103].
- A related issue is how to combine different parts of existing GPLs and DSLs into a new DSL. For instance, in the Microsoft .NET framework many GPLs are compiled to the Common Language Runtime (CLR) [64]. Can this be helpful in including selected parts of GPLs into a new DSL?
- Provide *"pattern aware" development support*. The Sprint system [36], for instance, provides partial evaluation support for the interpreter pattern (see Section 3.1). Other patterns might benefit from specialized support as well. Embedding support is discussed separately in the next paragraph.
- Reduce the need for learning some of the specialized metalanguages of language development systems by supporting *description by example* (DBE) of selected language aspects like syntax or prettyprinting. The user-friendliness of DBE is due to the fact that examples of intended behavior do not require a specialized metalanguage, or only a small part of it. Grammar inference from example sentences, for instance, may be viable, especially since many DSLs are small. This is certainly no new idea [41, 108], but it remains to be realized. Some preliminary results are reported in [40].
- How can DSL development tools generated by language development systems and toolkits be integrated with other software development tools? Using a COTS-based approach, XML technologies such as DOM and

XML-parsers have great potential as a uniform data interchange format for CASE tools. See also [7, 34].

Embedding GPLs should provide more powerful support for embedding DSLs, both syntactically and semantically. Some issues are:

- Embedding suffers from the very limited user-definable syntax offered by GPLs. Perhaps surprisingly, there is no trend toward more powerful user-definable syntax in GPLs over the years. In fact, just the opposite has happened. Macros and user-definable operators have become less popular. Java has no user-definable operators at all. On the other hand, some of the language development systems in Table 16, such as ASF+SDF and to some extent Stratego, support metalanguages featuring fully general user-definable context-free syntax. Although these metalanguages cannot compete directly with GPLs as embedding hosts as far as expressiveness and efficiency are concerned, they can be used to express a source-to-source transformation to translate user-defined DSL syntax embedded in a GPL to appropriate API calls. See [25] for an extensive discussion of this approach.

- Improved embedding support is not only a matter of language features, but also of language implementation, and in particular of preprocessors or extensible compilers allowing addition of domain-specific optimization rules and/or domain-specific code generation. See the references given in Section 2.5.1 and [65, 119]. Alternatively, the GPL itself might feature domain-specific optimization rules as a special kind of compiler directive. Such compiler extension makes the embedding process significantly more complex, however, and its cost-benefit ratio needs further scrutiny.

Estimation

- Last but not least: In this article our approach toward DSL development has been qualitative. Can the costs and benefits of DSLs be reliably quantified?

Acknowledgments We would like to thank the anonymous reviewers for many useful comments. Arie van Deursen kindly gave us permission to use the source of the annotated DSL bibliography [48].

References

[1] M. Anlauff, P. W. Kutter, and A. Pierantonio. Tool support for language design and prototyping with Montages. In S. Jähnichen, editor, *Compiler Construction (CC '99)*, volume 1575 of *Lecture Notes in Computer Science*, pages 296–299. Springer-Verlag, 1999.

[2] M. Antoniotti and A. Göllü. SHIFT and SMART-AHS: A language for hybrid system engineering modeling and simulation. In Ramming [113], pages 171–182.

[3] D. Atkins, T. Ball, G. Bruns, and K. Cox. Mawl: A domain-specific language for form-based services. In TSE-DSL [152], pages 334–346.

[4] I. Attali, C. Courbis, P. Degenne, A. Fau, D. Parigot, and C. Pasquier. SmartTools: A generator of interactive environments tools. In R. Wilhelm, editor, *Compiler Construction: 10th International Conference (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 355–360. Springer-Verlag, 2001.

[5] J. Aycock. The design and implementation of SPARK, a toolkit for implementing domain-specific languages. In CIT-DSL-II [102], pages 55–66.

[6] J. W. Backus. The syntax and semantics of the proposed International Algebraic Language of the Zurich ACM-GAMM conference. In *Proceedings of the International Conference on Information Processing, UNESCO, Paris, 1959*, pages 125–132. Oldenbourg, Munich and Butterworth, London, 1960.

[7] G. Badros. JavaML: A markup language for Java source code. In *Proceedings of the Ninth International World Wide Web Conference*, 2000. `http://www9.org/w9cdrom/start.html`.

[8] O. S. Bagge and M. Haveraaen. Domain-specific optimisation with user-defined rules in CodeBoost. In J.-L. Giavitto and P.-E. Moreau, editors, *Proceedings of the 4th International Workshop on Rule-Based Programming (RULE 2003)*, volume 86(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003. `http://www.sciencedirect.com/`.

[9] D. W. Barron. *The World of Scripting Languages*. Wiley, 2000.

[10] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for implementing domain-specific languages. In P. Devanbu and J. Poulin, editors, *Proceedings of the Fifth International Conference on Software Reuse (JCSR '98)*, pages 143–153. IEEE Computer Society, 1998.

[11] D. Batory, J. Thomas, and M. Sirkin. Reengineering a complex application using a scalable data structure compiler. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 111–120, 1994.

[12] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformation for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 625–634. IEEE Computer Society, 2004.

[13] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: A roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 73–87. ACM Press, 2000.

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

[15] T. J. Bergin and R. G. Gibson, editors. *History of Programming Languages II.* ACM Press, 1996.

[16] F. Bertrand and M. Augeraud. BDL: A specialized language for per-object reactive control. In TSE-DSL [152], pages 347–362.

[17] T. J. Biggerstaff. A perspective of generative reuse. *Annals of Software Engineering*, 5:169–226, 1998.

[18] T. J. Biggerstaff and A. J. Perlis, editors. *Software Reusability.* ACM Press/Addison-Wesley, 1989. Vol. I: Concepts and Models, Vol. II: Applications and Experience.

[19] D. Bonachea, K. Fisher, A. Rogers, and F. Smith. Hancock: A language for processing very large-scale data. In DSL-99 [49], pages 163–176.

[20] J. Bosch and Y. Dittrich. Domain-specific languages for a changing world. `http://www.cs.rug.nl/~bosch/articles.html`, n.d.

[21] C. Braband, M. I. Schwartzbach, and M. Vanggaard. The `metafront` system: Extensible parsing and transformation. In B. R. Bryant and J. Saraiva, editors, *Proceedings of the Third Workshop on Language Descriptions, Tools, and Applications (LDTA '03)*, volume 82(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003. `http://www.sciencedirect.com/`.

[22] C. Braband and M.I. Schwartzbach. Growing languages with metamorphic syntax macros. *ACM SIGPLAN Notices*, 37(3):31–40, March 2002.

[23] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Oliver, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: A component-based language development environment. In R. Wilhelm, editor, *Compiler Construction (CC 2001)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001. `http://www.cwi.nl/projects/MetaEnv`.

[24] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.

[25] M. Bravenboer and E. Visser. Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pages 365–383. ACM, 2004.

[26] F. P. Brooks, Jr. Language design as design. In Bergin and Gibson [15], pages 4–15.

[27] M. Bruntink, A. van Deursen, and T. Tourwé. Isolating idiomatic crosscutting concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM'05)*. IEEE Computer Society, 2005. Submitted.

[28] J. Buffenbarger and K. Gruell. A language for software subsystem composition. In HICSS-34 [73].

[29] L. Cardelli and R. Davies. Service combinators for web computing. In TSE-DSL [152], pages 309–316.

[30] S. Chandra, B. Richards, and J. R. Larus. Teapot: A domain-specific language for writing cache coherence protocols. In TSE-DSL [152], pages 317–333.

[31] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[32] S. Chiba. A metaobject protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, pages 285–299. ACM, 1995.

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

[34] J. C. Cleaveland. *Program Generators Using Java and XML*. Prentice-Hall, 2001.

[35] J. Clements, M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi. Fostering little languages. *Dr. Dobb's Journal*, 29(3):16–24, March 2004.

[36] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming (PLILP '98/ALP '98)*, volume 1490 of *Lecture Notes in Computer Science*, pages 170–194. Springer-Verlag, 1998.

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

[38] J. R. Cordy. TXL — A language for programming language tools and applications. In G. Hedin and E. van Wyk, editors, *Proceedings of the Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA '04)*, volume 110 of *Electronic Notes in Theoretical Computer Science*, pages 3–31. Elsevier, 2004. `http://www.sciencedirect.com/`.

[39] C. Courbis and A. Finkelstein. Towards an aspect weaving BPEL engine. In Y. Coady and D. H. Lorenz, editors, *Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2004. Technical Report NU-CCIS-04-04, College of Computer and Information Science, Northeastern University, Boston, MA 02115.

[40] M. Črepinšek, M. Mernik, F. Javed, B. R. Bryant, and A. Sprague. Extracting grammar from programs: evolutionary approach. *ACM SIGPLAN Notices*, 40(4):39–46, April 2005.

[41] S. Crespi-Reghizzi, M. A. Melkanoff, and L. Lichten. The use of grammatical inference for designing programming languages. *Communications of the ACM*, 16:83–90, 1973.

[42] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In Ramming [113], pages 229–242.

[43] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley, 2000.

[44] M. Dean, G. Schreiber, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL Web Ontology Language Reference. Working draft, W3C, March 2003. `http://www.w3.org/TR/2003/WD-owl-ref-20030331/`.

[45] M. Denny. Ontology building: A survey of editing tools. Technical report, XML.com, 2003. `http://www.xml.com/lpt/a/2002/11/06/ontologies.html`.

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

[47] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. In CIT-DSL-II [102], pages 1–17.

[48] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[49] *Proceedings of the second USENIX Conference on Domain-Specific Languages (DSL '99)*. USENIX Association, 1999.

[50] C. Elliott. An embedded modeling language approach to interactive 3D and multimedia animation. In TSE-DSL [152], pages 291–308.

[51] R. van Engelen. ATMOL: A domain-specific language for atmospheric modeling. In CIT-DSL-I [101], pages 289–303.

[52] R. E. Faith, L. S. Nyland, and J. F. Prins. Khepera: A system for rapid implementation of domain specific languages. In Ramming [113], pages 243–255.

[53] R. A. Falbo, G. Guizzardi, and K. C. Duarte. An ontological approach to domain engineering. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 2002)*, pages 351–358. ACM, 2002.

[54] M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi. Building little languages with macros. *Dr. Dobb's Journal*, 29(4):45–49, April 2004.

[55] K. Fertalj, D. Kalpič, and V. Mornar. Source code generator based on a proprietary specification language. In HICSS-35 [74].

[56] W. Frakes. Panel: Linking domain analysis with domain implementation. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 348–349. IEEE Computer Society, 1998.

[57] W. Frakes, R. Prieto-Diaz, and C. Fox. DARE: Domain analysis and reuse environment. *Annals of Software Engineering*, 5:125–141, 1998.

[58] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[59] R. Germon. Using XML as an intermediate form for compiler development. In *XML 2001 Conference Proceedings*, 2001. `http://www.idealliance.org/papers/xml2001/index.html`.

[60] J. Gil and Y. Tsoglin. JAMOOS — A domain-specific language for language processing. In CIT-DSL-I [101], pages 305–321.

[61] S. Gilmore and M. Ryan, editors. *Language Constructs for Describing Features — Proceedings of the FIREworks Workshop*. Springer-Verlag, 2001.

[62] U. Glässer, Y. Gurevich, and M. Veanes. An abstract communication model. Technical Report MSR-TR-2002-55, Microsoft Research, Redmond, 2002.

[63] K. Gondow and H. Kawashima. Towards ANSI C program slicing using XML. In M. G. J. van den Brand and R. Lämmel, editors, *Proceedings of the Second Workshop on Language Descriptions, Tools, and Applications (LDTA '02)*, volume 65(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002. `http://www.sciencedirect.com/`.

[64] J. Gough. *Compiling for the .NET Common Language Runtime (CLR)*. Prentice Hall, 2002.

[65] A. Granicz and J. Hickey. Phobos: Extending compilers with executable language definitions. In HICSS-36 [75].

[66] J. Gray and G. Karsai. An examination of DSLs for concisely representing model traversals and transformations. In HICSS-36 [75].

[67] R. W. Gray, S. P. Levi, V. P. Heuring, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, February 1992.

[68] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[69] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In DSL-99 [49], pages 39–52.

[70] S. Z. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. In *Proceedings of the IEEE* [106], pages 342–357.

[71] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, 35(3):39–48, March 2000.

[72] 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.

[73] *Proceedings of the 34th Hawaii International Conference on System Sciences (HICSS-34)*. IEEE (CDROM), 2001.

[74] *Proceedings of the 35th Hawaii International Conference on System Sciences (HICSS-35)*. IEEE (CDROM), 2002.

[75] *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS-36)*. IEEE (CDROM), 2003.

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

[77] P. Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings of the Fifth International Conference on Software Reuse (JCSR '98)*, pages 134–142. IEEE Computer Society, 1998.

[78] J. Jennings and E. Beuscher. Verischemelog: Verilog embedded in Scheme. In DSL-99 [49], pages 123–134.

[79] C. Jones. SPR Programming Languages Table. `http://www.theadvisors.com/langcomparison.htm`, March 1996. Release 8.2, accessed April 21, 2005; later release n.a. to authors at time of writing.

[80] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[81] M. de Jonge. Source tree composition. In C. Gacek, editor, *Software Reuse: Methods, Techniques, and Tools: 7th International Conference (ICSR-7)*, volume 2319 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 2002.

[82] B. M. Kadhim and W. M. Waite. Maptool — Supporting modular syntax development. In T. Gyimóthy, editor, *Compiler Construction (CC '96)*, volume 1060 of *Lecture Notes in Computer Science*, pages 268–280. Springer-Verlag, 1996.

[83] S. Kamin, editor. *DSL '97 — First ACM SIGPLAN Workshop on Domain-Specific Languages, in Association with POPL '97.* University of Illinois Computer Science Report, 1997. `http://www-sal.cs.uiuc.edu/~kamin/dsl/`.

[84] S. Kamin. Research on domain-specific embedded languages and program generators. *Electronic Notes in Theoretical Computer Science*, 14, 1998. `http://www.sciencedirect.com/`.

[85] S. Kamin and D. Hyatt. A special-purpose language for picture-drawing. In Ramming [113], pages 297–310.

[86] 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.

[87] U. Kastens and P. Pfahler. Compositional design and implementation of domain-specific languages. In R. N. Horspool, editor, *IFIP TC2 WG 2.4 Working Conference on System Implementation 2000: Languages, Methods and Tools*, pages 152–165. Chapman and Hall, 1998.

[88] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[89] 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 '96)*, pages 542–552. IEEE, 1996.

[90] H. M. Kienle and D. L. Moore. `smgn`: Rapid prototyping of small domain-specific languages. In CIT-DSL-II [102], pages 37–53.

[91] N. Klarlund and M. Schwartzbach. A domain-specific language for regular sets of strings and trees. In TSE-DSL [152], pages 378–386.

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

[93] D. J. Kuck. Platform 2015 software: Enabling innovation in parallelism for the next decade. *Technology@Intel Magazine*, pages 1–9, April 2005.

[94] S. Kumar, Y. Mandelbaum, X. Yu, and K. Li. ESP: A language for programmable devices. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, pages 309–320. ACM, 2001.

[95] P. W. Kutter, D. Schweizer, and L. Thiele. Integrating domain specific language design in the software life cycle. In D. Hutter et al., editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 196–212. Springer-Verlag, 1998.

[96] J. Launchbury, J. R. Lewis, and B. Cook. On embedding a microarchitectural design language within Haskell. *ACM SIGPLAN Notices*, 34(9):60–69, September 1999.

[97] C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors. *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[98] M. R. Levy. Web programming in Guide. *Software — Practice and Experience*, 28:1581–1603, 1998.

[99] J. Martin. *Fourth-Generation Languages*. Prentice-Hall, 1985. Vol. I: Principles, Vol II: Representative 4GLs.

[100] S. Mauw, W. Wiersma, and T. Willemse. Language-driven system design. *International Journal of Software Engineering and Knowledge Engineering*, 14:1–39, 2004.

[101] M. Mernik and R. Lämmel (editors). Special issue on Domain-Specific Languages, Part I. *Journal for Computing and Information Technology*, 9(4), 2001.

[102] M. Mernik and R. Lämmel (editors). Special issue on Domain-Specific Languages, Part II. *Journal for Computing and Information Technology*, 10(1), 2002.

[103] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. Multiple attribute grammar inheritance. *Informatica*, 24(3):319–328, September 2000.

[104] M. Mernik, U. Novak, E. Avdičaušević, M. Lenič, and V. Žumer. Design and implementation of Simple Object Description Language. In *Proceedings of the 2001 ACM Symposium on Applied Computing (SAC 2001)*, pages 590–594. ACM, 2001.

[105] M. Mernik, V. Žumer, M. Lenič, and E. Avdičaušević. Implementation of multiple attribute grammar inheritance in the tool LISA. *ACM SIGPLAN Notices*, 34(6):68–75, June 1999.

[106] J. M. F. Moura, M. Püschel, D. Padua, and J. Dongarra (editors). Special issue on Program Generation, Optimization, and Platform Adaptation. *Proceedings of the IEEE*, 93(2), 2005.

[107] L. Nakatani and M. Jones. Jargons and infocentrism. In Kamin [83], pages 59–74. `http://www-sal.cs.uiuc.edu/~kamin/dsl/`.

[108] B. A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, 1993.

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

[110] D. Parigot. Towards domain-driven development: The SmartTools software factory. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented Programming Systems, Languages, and Applications*, pages 37–38. ACM, 2004.

[111] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Proceedings of the Haskell Workshop 2001*, 2001.

[112] P. Pfahler and U. Kastens. Configuring component-based specifications for domain-specific languages. In HICSS-34 [73].

[113] J. C. Ramming, editor. *Proceedings of the USENIX Conference on Domain-Specific Languages*. USENIX Association, 1997.

[114] E. S. Raymond. The CML2 language: Python implementation of a constraint-based interactive configurator. In *Proceeding of the 9th International Python Conference*, pages 135–142, 2001. `http://www.catb.org/~esr/cml2/cml2-paper.html`.

[115] W. Risi, P. Martinez-Lopez, and D. Marcos. Hycom: A domain specific langauge for hypermedia application development. In HICSS-34 [73].

[116] D. T. Ross. Origins of the APT language for automatically programmed tools. In Wexelblat [148], pages 279–338.

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

[118] J. E. Sammet. *Programming Languages: History and Fundamentals*. Prentice-Hall, 1969.

[119] J. Saraiva and S. Schneider. Embedding domain specific languages in the attribute grammar formalism. In HICSS-36 [75].

[120] E. Schnarr, M. D. Hill, and J. R. Larus. Facile: A language and compiler for high-performance processor simulators. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, pages 321–331. ACM, 2001.

[121] K. A. Schneider and J. R. Cordy. AUI: A programming language for developing plastic interactive software. In HICSS-35 [74].

[122] S. Schupp, D. P. Gregor, D. R. Musser, and S. Liu. User-extensible simplification — Type-based optimizer generators. In R. Wilhelm, editor, *Compiler Construction (CC 2001)*, volume 2027 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, 2001.

[123] SDL Forum. MSC-2000: Interaction for the new millenium. `http://www.sdl-forum.org/MSC2000present/index.htm`, May 2000.

[124] M. Simos and J. Anthony. Weaving the model web: A multi-modeling approach to concepts and features in domain engineering. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 94–102. IEEE Computer Society, 1998.

[125] E. G. Sirer and B. N. Bershad. Using production grammars in software testing. In DSL-99 [49], pages 1–14.

[126] A. M. Sloane. Post-design domain-specific language embedding: A case study in the software engineering domain. In HICSS-35 [74].

[127] K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach.* Addison-Wesley, 1995.

[128] Y. Smaragdakis and D. Batory. DiSTiL: A transformation library for data structures. In Ramming [113], pages 257–270.

[129] Y. Smaragdakis and D. Batory. Application generators. In J. Webster, editor, *Wiley Encyclopedia of Electrical and Electronics Engineering Online.* Wiley, 2000.

[130] D. Soroker, M. Karasick, J. Barton, and D. Streeter. Extension mechanisms in Montana. In *Proceedings of the 8th Israeli Conference on Computer-Based Systems and Software Engineering (ICCSSE '97)*, pages 119–128. IEEE Computer Society, 1997.

[131] D. Spinellis. Notable design patterns for domain-specific languages. *The Journal of Systems and Software*, 56:91–99, 2001.

[132] A. Sutcliffe and N. Mehandjiev (editors). Special issue on End-User Development. *Communications of the ACM*, 47(9), 2004.

[133] C. Szyperski. *Component Software — Beyond Object-Oriented Programming.* Addison-Wesley/ACM Press, second edition, 2002.

[134] 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.

[135] R. D. Tennent. Language design methods based on semantic principles. *Acta Informatica*, 8:97–112, 1977.

[136] S. Thatte. XLANG: Web services for business process design. Technical report, Microsoft, 2001. `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/`.

[137] S. A. Thibault. *Domain-Specific Languages: Conception, Implementation and Application.* PhD thesis, University of Rennes, 1998.

[138] S. A. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143. IEEE Computer Society, 1998.

[139] S. A. Thibault, R. Marlet, and C. Consel. Domain-specific languages: From design to implementation — Application to video device drivers generation. In TSE-DSL [152], pages 363–377.

[140] W. Tracz and L. Coglianese. DOMAIN (DOmain Model All INtegrated) — a DSSA domain analysis tool. Technical Report ADAGE-LOR-94-11, Loral Federal Systems, 1995.

[141] Universal Plug and Play Forum. `http://www.upnp.org/`, 2003.

[142] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[143] T. L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[144] T. L. Veldhuizen. Blitz++ User's Guide. `http://www.oonumerics.org/blitz/manual/blitz.ps`, February 2001. Version 1.2.

[145] E. Visser. Stratego — Strategies for program transformation. `http://www.stratego-language.org`, 2003.

[146] D. C. Wang, A. W. Appel, J. L. Korn, and C. S. Serra. The Zephyr abstract syntax description language. In Ramming [113], pages 213–28.

[147] D. Weiss and C. T. R. Lay. *Software Product Line Engineering.* Addison-Wesley, 1999.

[148] R. L. Wexelblat, editor. *History of Programming Languages.* Academic Press, 1981.

[149] D. S. Wile. *POPART: Producer of Parsers and Related Tools.* USC/Information Sciences Institute, November 1993. `http://mr.teknowledge.com/wile/popart.html`.

[150] D. S. Wile. Supporting the DSL spectrum. In CIT-DSL-I [101], pages 263–287.

[151] D. S. Wile. Lessons learned from real DSL experiments. *Science of Computer Programming*, 51:265–290, 2004.

[152] D. S. Wile and J. C. Ramming (editors). Special issue on domain-specific languages. *IEEE Transactions on Software Engineering*, SE-25(3), May/June 1999.

[153] J. Xiong, J. Johnson, R. W. Johnson, and D. A. Padua. SPL: A language and compiler for DSP algorithms. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, pages 298–308. ACM, 2001.