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@comp.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.

While 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 try to improve on and extend earlier work on DSL design patterns, in particular by Spinellis (2001). 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*, *special purpose*, *task-specific*, *problem-oriented*, *specialized*, or *application* languages.

DSLs trade generality for expressiveness in a limited domain. By providing notations and constructs tailored toward a particular application domain, they offer substantial gains in expressiveness and ease of use compared with general purpose programming languages (GPLs) for the domain in question. These in turn lead to gains in productivity and reduced maintenance costs. While GPLs yield an order of magnitude productivity improvement over assembly languages,

DSL	Application domain
BNF	Syntax specification
Excel macro language	Spreadsheets
HTML	Hypertext web pages
LaTeX	Typesetting
Make	Software building
SQL	Database queries
VHDL	Hardware design

Table 1: Some widely used domain-specific languages.

DSLs give another order of magnitude improvement over GPLs in their domain of application.

The use of DSLs is by no means new. APT, a DSL for programming numerically controlled machine tools, was developed in 1957–1958 [76]. BNF, the well-known syntax specification formalism, dates back to 1959 [4]. So-called *fourth-generation languages* (4GLs) are usually DSLs for database applications. *Little languages* are small DSLs that do not include many features found in GPLs [10, p. 715]. Some widely used DSLs with their application domains are listed in Table 1. Many others will be mentioned in later sections. Visual DSLs, such as UML, are important, but are beyond the scope of this article.

We will not try to give a definition of what constitutes an application domain and what does not. Some people consider Cobol to be a DSL for business applications, while 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 HTML on the left and GPLs such as C++ on the right. 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, so 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 on functions or objects that can be put in a library. This means a GPL in combination with an application library can only express these constructs indirectly or in a cumbersome, repetitious way. Again, a DSL would incorporate domain-specific constructs

from the start.

- Unlike GPLs, DSLs *need not be executable*. There is some confusion on this in the DSL literature. For instance, the importance of non-executable DSLs is emphasized in [96], while DSLs are required to be executable in [28]. We discuss DSL executablility 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). 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 frameworks has further complicated the relative merits of DSLs and application libraries.

Consider Microsoft Excel, for instance. Its 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 or toolbox of COM components. This has opened it up to general purpose programming languages such as C++, Java and Basic, which can access it through its COM interfaces. This is called *automation* [17]. Unlike the Excel macro language, which by its very nature is limited to Excel functionality, general purpose programming languages 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.

1.2 Executability of DSLs

DSLs are executable in various ways and to various degrees, even to the point of being non-executable. Accordingly, DSL 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* [19, 83]. These 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 input language for a parser generator.
- DSL not meant to be executable [96]. Just like their executable relatives, such non-executable DSLs may benefit from various kinds of tool sup-

port such as specialized editors, prettyprinters, consistency checkers, 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 [11]. Among the types of artifact that can be reused via DSLs are syntax, source code, software designs, and domain abstractions.

In his definitive survey of reuse [59], Krueger categorises 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 DSLs as their input language, thereby enabling the reuse of domain semantics. 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 paper.

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. DSLs implemented as application libraries clearly enable reuse of source code and DSLs can play a role in the formal specification of software schemas.

Reuse of syntax may take the form of reuse of (parts of) an actual grammar already available in an existing GPL or DSL processor or reuse of a notation already in use by domain experts, but perhaps not yet available in a computer language.

1.4 Scope of this article

There are no easy answers to the "when and how" question in the title of this article. The above-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

sizeable 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* [9].

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* phase of DSL development (Section 2). The DSL development process can be facilitated by using domain analysis tools and language development systems. These are surveyed in Section 3. Our patterns try to improve on and extend earlier work on DSL design patterns, in particular by Spinellis [85]. This is discussed in Section 2.6. Other related work is discussed at appropriate points throughout rather than in a separate section. Finally, conclusions and open problems are given in Section 4. As mentioned before, visual DSLs are beyond the scope of this article.

1.5 Literature

Until recently, DSLs received relatively little attention in the computer science research community and there are few books on the subject. [62] is an exhaustive account of the state of 4GLs at the time it was written, [12] is a two-volume collection of articles on software reuse including DSL development and program generation, [70] focuses on the role of DSLs in end-user programming, [77] is a collection of articles on little languages (not all of them DSLs), [6] treats scripting languages (again, not all of them DSLs), [24] discusses domain analysis, program generators, and generative programming techniques, and [20] discusses domain analysis and the use of XML, DOM, XSLT, and related languages and tools to generate programs.

Proceedings of recent workshops and conferences partly or exclusively devoted to DSLs are [53, 74, 29, 46, 47, 48]. Several journals have published special issues on DSLs [98, 64, 65]. 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 [36]. The annotated DSL bibliography [28] (78 items) has limited overlap with the references in this article because of our emphasis on general DSL development issues. Finally, articles on DSL patterns and DSL development methodologies are [21, 49, 83, 85, 96].

2 DSL Patterns

2.1 Pattern classification

The following DSL development phases can be distinguished:

- decision,
- analysis,

When/ How	Pattern class	Description
When	Decision pattern	Common situations suitable for designing a new DSL (or use of an existing one)
How	Analysis pattern	Common approaches to domain analysis
How	Design pattern	Common approaches to DSL design
How	Implementation pattern	Common approaches to DSL implementation

Table 2: Pattern classification.

- design,
- implementation,
- deployment.

DSL development is not a simple sequential process of (positive) decision followed by domain analysis, followed by DSL design, and so on. In practice, the decision process may be influenced by preliminary analysis, analysis in turn may have to supply answers to unforeseen questions arising during design, and design is often influenced by implementation considerations.

As shown in Table 2, we associate classes of patterns with each of the above development phases except deployment, which is beyond the scope of this article. Patterns in different classes 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

The decision phase corresponds to the "when"-part of DSL development. 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. In practice, short-term considerations and lack of expertise may easily cause indefinite postponement of the decision.

To aid in the decision process, we identify a number of *decision patterns*. These are common situations that potential developers find themselves in that might motivate the use of DSLs. Underlying these patterns are general, interrelated concerns such as

Pattern	Description
Notation	The availability of appropriate (new or existing) domain-specific notations is the decisive factor. Important special cases: • Make existing visual notation available in textual form. There are many benefits to such a visual-to-textual transformation, such as easier composition of large programs or specifications, etc. The pre-eminent example of this subpattern is probably VHDL [2]. Another one is MSF [41]. • Add domain-specific notation beyond the limited user-definable operator notation offered by GPLs to an existing application library.
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 for an appropriate DSL (e.g., SODL [67], proprietary specification language [32]).
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 (e.g., Fido [58]).
Data structure traversal	Traversals over complicated data structures can often be expressed better and more reliably in a suitable DSL (e.g., TVL [41]).
System front-end	A DSL based front-end may often be used for handling a system's configuration and adaptation (e.g., Nowra [81]).
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". Another example in the context of web computing is discussed in [16].
AVOT	Domain-specific analysis, verification, optimization, 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 (e.g., Promela++ [7]).

Table 3: Decision 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 GPL code.

Table 4: Analyis patterns.

- improved software economics,
- enabling of end-user programming or end-user specification,
- enabling of domain-specific analysis, verification, optimization, and/or transformation.

The decision patterns we have identified are listed in Table 3. We discuss some of them in more detail. XXX

2.3 Analysis

In the analysis phase of DSL development, the problem domain is identified and domain knowledge is gathered. This requires input from domain experts and/or the availability of documents or code from which domain knowledge can be obtained. Most of the time, domain analysis is done informally, but sometimes domain analysis methodologies such as DARE (Domain Analysis and Reuse Environment) [34], DSSA (Domain-Specific Software Architectures) [86], FODA (Feature-Oriented Domain Analysis) [55], or ODM (Organization Domain Modeling) [80] are used. See also [24, Part I].

There is a close link with *knowledge engineering*, which is only beginning to be explored. Knowledge capture, knowledge representation, and ontology development [25] are potentially useful in the analysis phase. The latter is taken into account in ODE (Ontology-based Domain Engineering) [31].

The output of formal domain analysis varies widely, but is some kind of representation of the domain knowledge obtained. It may range from a *feature diagram*, which is a graphical representation of assertions (propositions, predicates) about software systems in a particular domain, to a *domain implementation* consisting of a set of domain-specific reusable components, or a full-fledged *theory* in the case of highly developed scientific domains.

The analysis patterns we have identified are shown in Table 4. The last pattern overlaps with the first two. Code may be used as a source of domain knowledge informally or formally in the context of ODM or another methodology. Tool support for formal domain analysis is discussed in Section 3.2.

Pattern	Description
Language exploitation	DSL is based on an existing language. Important special cases: • 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 systems, or abstract state machines [82].

Table 5: Design patterns.

2.4 Design

Approaches to DSL design can be characterised along two orthogonal dimensions: the relationship between the DSL and existing languages, and the formal nature of the design description (Table 5).

The easiest way to design a DSL is to base it on an existing language. One possible benefit is familiarity for users, but this only applies if the domain users are also programmers in the existing language, which as noted above is often not 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 *specialisation* 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 where a notation is already widely known. For example, many DSLs contain arithmetic expressions which are usually written in the infix-operator style of 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 to language extension 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 characterise. Well-known GPL design citeria such as readability, simplicity, orthogonality, etc., and Tennent's design principles [87] 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. We quote lesson 3 (of a total of 12 lessons learned from real DSL experiments) from [97]:

> **Lesson 3:** You are almost never designing a programming language.
> **Lesson 3 Corollary:** Design only what is necessary. Learn to recognize your tendency to over-design.
> 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.

Once the relationship to existing languages has been determined, a DSL designer must turn to specifying the design before implementation. We distinguish 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 [82]. 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. However, a formal approach should not be discounted. Informal language designs can contain imprecisions that cause problems in the implementation phase. They typically focus on syntax, leaving semantic concerns to the imagination. The discipline required by development of a formal specification of both syntax and semantics can bring problems to light before implementation. Furthermore, as we shall see, formal designs can be implemented automatically by tools, thereby significantly reducing implementation effort.

2.5 Implementation

2.5.1 Patterns

When an (executable) DSL is designed, the most suitable implementation approach should be chosen. The implementation patterns we have identified are shown in Table 6. We discuss some of them in more detail.

First, it should be noted that 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 [85], 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. Development cost is not directly related to

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 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 the base language. Static analysis is limited to that done by the base language processor. Important special cases: • Source-to-source transformation: DSL source code is transformed (translated) into source code of existing language (the base language). • Pipeline: Processors successively handling sublanguages of a DSL and translating them to the input language of the next stage. This pattern also includes examples where only simple lexical processing is required, without complicated tree-based syntax analysis.
Embedding	DSL constructs are embedded in existing GPL (the host language) by defining new abstract data types and operators. Application libraries are the basic form of embedding.
Extensible compiler/interpreter	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	Existing tools and/or notations are applied to a specific domain.
Hybrid	A combination of the above approaches is used.

Table 6: Implementation patterns for executable DSLs.

implementation method, however, especially if a language development system or toolkit is used to generate the implementation (Section 3). DSL compilers are often called application generators.

An alternative to the traditional approach to the implementation of DSLs is by embedding. 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 problem in a domain 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. While this technique is quite powerful, pitfalls exist in overloading familiar operators to have unfamiliar semantics.

C++ is also emblematic of another specific embedding approach: *template metaprogramming* [23]. In this method, templates are used to achieve compile-time generation of domain-specific code. Significant mileage has been made out of this approach to develop mathematical libraries for C++ which have familiar domain notation but also achieve good performance.

Although the host language in the embedding approach can be any general-purpose language, functional languages are often appropriate, as shown by many researchers [49, 54]. This is due to functional language features such as expressiveness, lazy evaluation, higher-order functions, and strong typing with polymorphism and overloading. Imperative, object-oriented or logic languages are used as host languages to a lesser extent usually due to a lack of these sorts of mechanisms.

Extending an existing language implementation can also be seen as a form of embedding. The difference is usually a matter of degree. In a traditional 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 [5] and in the Simplicissimus GCC compiler plug-in [79], the Montana extensible C++ programming environment [84], and user-definable optimization rules in the GHC Haskell compiler [72]. Some extensible compilers, such as OpenC++ [18], support a metaobject protocol. This is an object-oriented interface for specifying language extensions and transformations.

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 [96]. The current prominence of XML-based DSLs is another instance of this approach [38].

Many DSL endeavours 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.

2.5.2 Implementation tradeoffs

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

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

while some 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 realise because most language processors are not designed with extension in mind.

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, and
- a modular and extensible formal method for DSL design is used so that new features can be added without significant modification to the processing of old features.

Advocates of the embedded approach often criticize DSLs implemented by the traditional approach in that too much effort is put into the syntax. On the other hand, the language semantics is often poorly designed and cannot be easily extended with new features [54]. Indeed, the syntax of a DSL is extremely important and should not be underestimated. The syntax should be as close as possible to the notation used in a domain.

The above-mentioned shortcomings of the traditional approach can be avoided if a formal method is used for the development of the DSL. To date the following formal methods for programming language description have been used: abstract

state machines, algebraic specifications, attribute grammars, denotational semantics, and operational semantics. Furthermore, a useful formal method has to support incrementality, modularity and extensibility of specifications since DSLs change more frequently than GPLs [96, 13]. An example of an extensible formal method is multiple attribute grammar inheritance [66] which enables incremental language development similar to the modular monadic approach often used in embedding [49]. The most productive development of languages is based on high-level automated tools which automatically generate compiler/interpreter from formal specifications.

To return to the embedded approach, its advantages are:

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

However, it is much harder to overcome the disadvantages of the embedded approach:

- 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 [54, 81].

In the functional setting, some of these shortcomings can be reduced by using monads [49] or user-defined optimizations in the GHC compiler (if Haskell is used) [72] for domain-specific optimizations and by using partial evaluation for better overall efficiency [49, 21].

The decision diagram on how to proceed with DSL implementation (Fig. 1) shows when a particular implementation approach is more appropriate.

If in the DSL design phase the exploitation pattern is used then the piggyback pattern, language extension pattern, and language specialization pattern can be implemented using various implementation patterns. Figure 2 shows the implementation cost-benefit tradeoff associated with applying the implementation patterns to realise various design patterns.

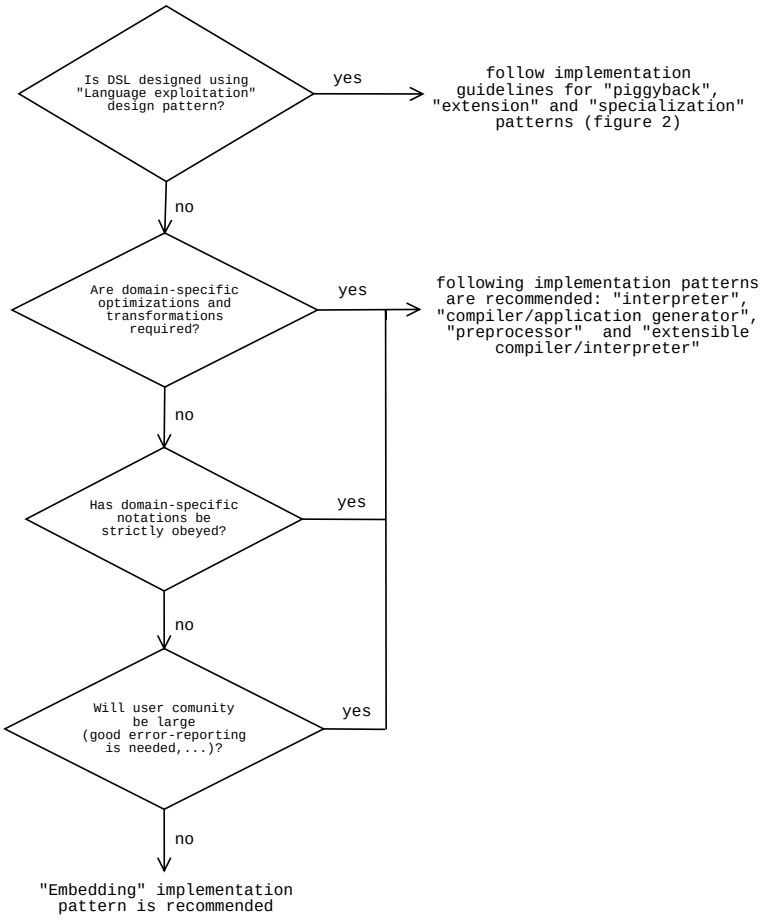

Figure 1: Implementation guidelines.

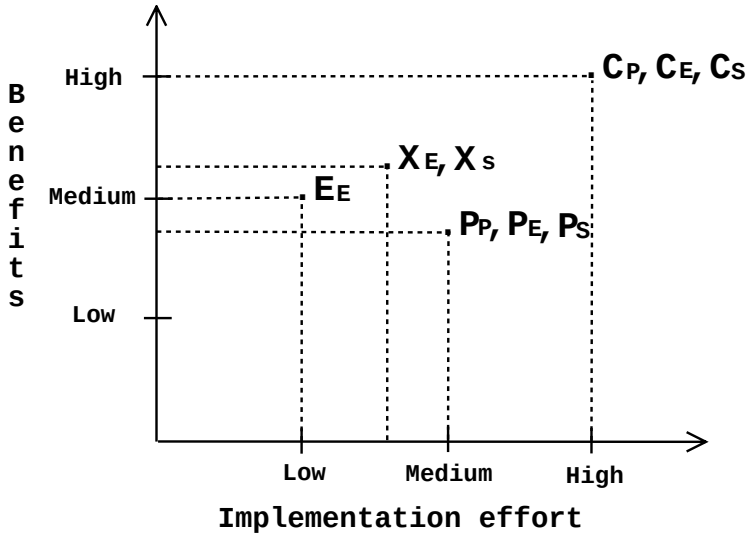

Figure 2: Implementation cost-benefit tradeoff for feasible pairs of design and implementation pattern. Pairs are denoted I_J where I is an implementation pattern (compiler/interpreter (C), preprocessor (P), exensible compiler/interpreter (X), and embedding (E)), and J is a design pattern (piggyback (P), extension (E), and specialization (S)).

Note, that if the existing language is just partially used then the embedding pattern or extensible compiler/interpreter pattern are not suitable. Furthermore, the embedding pattern is not suitable if the existing language has to be restricted. Such language specialization can be implemented using an extensible compiler/interpreter approach in some languages (e.g., Smalltalk). In general, with the extensible compiler/interpreter approach it is much easier to extend the language rather than restrict it.

2.6 Comparison with other classifications

We start by comparing our patterns with those proposed in [85]. Closely following [35], Spinellis distinguishes three classes of DSL patterns as shown in Table 7. The specific patterns for each class are summarized in Tables 8, 9, and 10. 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 11. 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. As can be seen, we have retained the terminology used by Spinellis whenever appropriate.

Another classification of DSL development approaches is given in [96], 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.

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 12. 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* [44]. The tools generated may vary from a consis-

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

Table 7: Pattern classification proposed by Spinellis.

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 8: Creational 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 9: Structural patterns.

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

Table 10: Behavioral patterns.

Spinellis's pattern	Our pattern
Creational: language extension	Design: language exploitation (language extension)
Creational: language specialization	Design: language exploitation (language 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)

Table 11: 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.

tency checker and interpreter to an integrated development environment (IDE) consisting of a syntax-directed editor, a prettyprinter, an (incremental) consistency checker, 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, and consistency checkers. 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 assumes an interpreter for the DSL to be given and then uses a form of program transformation called *partial evaluation* [50] to remove the interpretation overhead by automatically transforming a DSL program into a compiled program. Other systems 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 include partial evaluation of a DSL interpreter given a specific program among their capabilities.

The input to these systems is a description of various aspects of the DSL to be developed in terms of specialized meta-languages. Depending on the type of DSL, some important language aspects are *syntax*, *prettyprinting*, *consistency checking*, *execution*, *translation*, *transformation*, and *debugging*. It so happens that the meta-languages 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

System	Developed at
ASF+SDF [14]	CWI/University of Amsterdam
AsmL [37]	Microsoft Research, Redmond
Draco [71]	University of California, Irvine
Eli [42]	University of Colorado, University of Paderborn, Macquarie University
Gem-Mex [1]	University of L'Aquila
InfoWiz [69]	Bell Labs/AT&T Labs
JTS [8]	University of Texas at Austin
Khepera [30]	University of North Carolina
Kodiyak [45]	University of Minnesota
LaCon [56]	University of Paderborn (LaCon uses Eli as back-end — see above)
LISA [68]	University of Maribor
Metatool [19]	Bell Laboratories
POPART [95]	USC/Information Sciences Institute
`smgn` [57]	Intel Compiler Lab/University of Victoria
SPARK [3]	University of Calgary
Sprint [21]	LaBRI/INRIA
Stratego [94]	University of Utrecht
TXL [92]	University of Toronto/Queen's University at Kingston

Table 12: Some language development systems and toolkits that have been used for DSL development.

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

Table 13: Development support provided by current language development systems and toolkits for DSL development phases/pattern classes.

System used	DSL	Application domain
ASF+SDF	Box [15]	Prettyprinting
	Risla [26]	Financial products
AsmL	UPnP [93]	Networked device protocol
	XLANG [88]	Business protocols
Eli	Maptool [52]	Grammar mapping
	(Various) [73]	Class generation
Gem-Mex	Cubix [60]	Virtual data warehousing
JTS	Jak [8]	Syntactic transformation
LaCon	(Various) [56]	Data model translation
LISA	SODL [67]	Network application
`smgn`	Hoof [57]	Compiler IR specification
	IMDL [57]	Software reengineering
SPARK	Guide [61]	Web programming
	CML2 [75]	System configuration
Sprint	GAL [90]	Video device drivers
	PLAN-P [89]	Application-specific protocols
Stratego	Autobundle [51]	Software building
	CodeBoost [5]	Domain-specific C++ optimization

Table 14: Examples of DSL development using the systems in Table 12.

(Table 1). The corresponding tool generated by the language development system is a parser.

Although the various specialized meta-languages 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, [82] for further details.

The level of support provided by these systems in various phases of DSL development is summarized in Table 13. Their main strength lies in the implementation phase. Support of DSL design tends to be weak. Their main assets are the meta-languages 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 12 are given in Table 14. They cover a wide range of appplication domains and implementation patterns. The Box prettyprinting meta-language is an example of a DSL developed with a language development system (in this case the ASF+SDF Meta-Environment) for later use as one of the meta-languages of the system itself. Similarly, the Jak transformational meta-language for specifying the se-

mantics of a DSL or domain-specific language extension in the Jakarta Tool Suite (JTS), was developed using JTS itself. In this case, this involved bootstrapping, since JTS not only requires language definitions to be written in Jak, but is itself written in Jak.

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 15. This is a research area. 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.

As noted in Section 2.3 the output of formal domain analysis is some kind of representation of the domain knowledge obtained. It may range from a feature diagram (see FDL entry in Table 15) to a domain implementation consisting of a set of domain-specific reusable components (see DARE entry in Table 15), or a full-fledged theory in the case of highly developed 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 metagenerator [19] is mentioned in [33].

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 16. Furthermore, we discussed language development systems and toolkits that can be used to facilitate the development process, especially its later phases.

Our survey also implicitly or explicitly showed many opportunities for further work. As indicated in Table 13, 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 12) 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

Analysis framework or tool	Description
Ariadne [80]	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 [34]	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 [91]	DSSA [86] 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 [27]	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 [55] domain analysis methodology. (FDL is an example of the visual-to-textual transformation subpattern in Table 3.)
ODE editor [31]	Ontology editor supporting ODE — see also [25].

Table 15: Some domain analysis frameworks and tools.

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

Table 16: Summary of DSL development phases and corresponding patterns.

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 from *parameterized language building blocks* [1, 21, 44].
- 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) [39]. Can this be helpful in including selected parts of GPLs into a new DSL?
- Provide *"pattern aware" development support.* The Sprint system [21], for instance, provides partial evaluation support for the interpreter pattern (see Section 3.1). Other patterns might benefit from specialized support as well.
- Reduce the need for learning some of the specialized meta-languages 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 meta-language, 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 [22], but it remains to be realized. Some preliminary results are reported in [63].
- 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 [20].

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. Java has no user-definable syntax at all. This is a neglected aspect of GPL design. The discussion in [43] is still relevant.
- 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 [40, 78]. 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

- In this article our approach toward DSL development has been qualitative. Can the costs and benefits of DSLs be reliably quantified?

Acknowledgements Arie van Deursen kindly gave us permission to use the source of the annotated DSL bibliography [28].

References

[1] M. Anlauff, P. W. Kutter, and A. Pierantonio. Formal aspects and development environments for Montages. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF '97)*, Electronic Workshops in Computing. Springer/British Computer Society, 1997.

[2] P. J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, 1995.

[3] J. Aycock. The design and implementation of SPARK, a toolkit for implementing domain-specific languages. In *Journal for Computing and Information Technology* [65], pages 55–66.

[4] 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.

[5] O. S. Bagge and M. Haveraaen. Domain-specific optimisation with user-defined rules in CodeBoost. In *Proceedings 4th International Workshop on Rule-Based Programming (RULE 2003)*, 2003. To appear in *Electronic Notes in Theoretical Computer Science*.

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

[7] A. Basu, M. Hayden, G. Morrisett, and T. von Eicken. A language-based approach to protocol construction. In Kamin [53], pages 1–15. `http://www-sal.cs.uiuc.edu/~kamin/dsl/`.

[8] 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.

[9] 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.

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

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

[12] 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.

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

[14] 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.

[15] 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.

[16] L. Cardelli and R. Davies. Service combinators for web computing. [98], pages 309–316.

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

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

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

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

[21] 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.

[22] 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.

[23] K. Czarnecki and U. Eisenecker. *Generative Programming.* Addison-Wesley, 2000.

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

[25] 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`.

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

[27] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. In *Journal for Computing and Information Technology* [65], pages 1–17.

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

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

[30] R. E. Faith, L. S. Nyland, and J. F. Prins. Khepera: A system for rapid implementation of domain specific languages. In Ramming [74], pages 243–55.

[31] 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 Press, 2002.

[32] K. Fertalj, D. Kalpič, and V. Mornar. Source code generator based on a proprietary specification language. In HICSS-35 [47].

[33] 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.

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

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

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

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

[38] K. Gondow and H. Kawashima. Towards ANSI C program slicing using XML. In M. G. J. van den Brand and R. Lämmel, editors, *Second Workshop on Language Descriptions, Tools and Applications (LDTA '02)*, 2002.

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

[40] A. Granicz and J. Hickey. Phobos: Extending compilers with executable language definitions. In HICSS-36 [48].

[41] J. Gray and G. Karsai. An examination of DSLs for concisely representing model traversals and transformations. In HICSS-36 [48].

[42] 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.

[43] J. Heering and P. Klint. The syntax definition formalism SDF. In J. A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, chapter 6. ACM Press, 1989.

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

[45] 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.

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

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

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

[49] 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.

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

[51] 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.

[52] 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.

[53] 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/`.

[54] S. Kamin. Research on domain-specific embedded languages and program generators. *Electronic Notes in Theoretical Computer Science*, 14, 1998.

[55] 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.

[56] 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.

[57] H. M. Kienle and D. L. Moore. `smgn`: Rapid prototyping of small domain-specific languages. In *Journal for Computing and Information Technology* [65], pages 37–53.

[58] N. Klarlund and M. Schwartzbach. A domain-specific language for regular sets of strings and trees. In *IEEE Transactions on Software Engineering* [98], pages 378–386.

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

[60] 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.

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

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

[63] M. Mernik, M. Črepinšek, G. Gerlič, V. Žumer, B. R. Bryant, and A. Sprague. Learning context-free grammars using an evolutionary approach. Technical report, University of Maribor and The University of Alabama at Birmingham, 2003.

[64] M. Mernik and R. Lämmel (eds.). Special issue on domain-specific languages, Part I. *Journal for Computing and Information Technology*, 9(4), 2001.

[65] M. Mernik and R. Lämmel (eds.). Special issue on domain-specific languages, Part II. *Journal for Computing and Information Technology*, 10(1), 2002.

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

[67] 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 Press, 2001.

[68] 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.

[69] L. Nakatani and M. Jones. Jargons and infocentrism. In Kamin [53], pages 59–74. `http://www-sal.cs.uiuc.edu/~kamin/dsl/`.

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

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

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

[73] P. Pfahler and U. Kastens. Configuring component-based specifications for domain-specific languages. In HICSS-34 [46].

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

[75] 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.

[76] D. T. Ross. Origins of the APT language for automatically programmed tools. In R. L. Wexelblat, editor, *History of Programming Languages*, pages 279–338. Academic Press, 1981.

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

[78] J. Saraiva and S. Schneider. Embedding domain specific languages in the attribute grammar formalism. In HICSS-36 [48].

[79] 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.

[80] 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.

[81] A. M. Sloane. Post-design domain-specific language embedding: A case study in the software engineering domain. In HICSS-35 [47].

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

[83] Y. Smaragdakis and D. Batory. Application generators. Technical report, Department of Computer Science, University of Texas at Austin, n.d. `http://www.cc.gatech.edu/~yannis/generators.pdf`.

[84] 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.

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

[86] 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.

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

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

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

[90] S. A. Thibault, R. Marlet, and C. Consel. Domain-specific languages: From design to implementation — application to video device drivers generation. [98], pages 363–377.

[91] 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.

[92] The TXL Programming Language, 2003. `http://www.txl.ca/`.

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

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

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

[96] D. S. Wile. Supporting the DSL spectrum. In *Journal for Computing and Information Technology* [64], pages 263–287.

[97] D. S. Wile. Lessons learned from real DSL experiments. In HICSS-36 [48].

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