Application Software, Domain-Specific Languages, and Language Design Assistants

J. Heering

Software Engineering (SEN)

SEN-R0010 May 31, 2000

Report SEN-R0010
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Application Software, Domain-Specific Languages, and Language Design Assistants

Jan Heering
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Jan.Heering@cwi.nl

ABSTRACT
While application software does the real work, domain-specific languages (DSLs) are tools to help produce it efficiently, and language design assistants in turn are meta-tools to help produce DSLs quickly. DSLs are already in wide use (HTML for web pages, Excel macros for spreadsheet applications, VHDL for hardware design, ...), but many more will be needed for both new as well as existing application domains. Language design assistants to help develop them currently exist only in the basic form of language development systems. After a quick look at domain-specific languages, and especially their relationship to application libraries, we survey existing language development systems and give an outline of future language design assistants.

1998 ACM Computing Classification System: D.3

Keywords and Phrases: application software, application domain, domain-specific language, task-oriented language, end user programming, language development system, language-based tool generation, language design assistant, language concept, description by example.

Note: To be presented at SSGRR 2000, L'Aquila, Italy. This research was supported in part by the Telematica Instituut under the *Domain-Specific Languages* project.

1 Domain-Specific Languages

Many computer languages are *domain-specific* rather than general purpose. Domain-specific languages (DSLs) are also called *task-specific*, *application-oriented*, or *problem-oriented*. Some well-known DSLs with their application domains are listed in Table 1. So-called *fourth-generation languages* (4GLs) are usually DSLs for database applications. A good reference, which focuses on the key role of DSLs in end user programming, is [18]. A recent DSL bibliography is [9].

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 (Table 1) on the left and general purpose programming languages such as C++ on the right. On this scale, Cobol would be somewhere between HTML and C++, but much closer to the latter.

In combination with an *application library*, any general purpose programming language 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 general purpose languages. A DSL offers domain-specific notations from the start. Their importance cannot be overestimated as they are directly related to the suitability for end user programming and, more generally, the programmer 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 general purpose language using an application library can only express these constructs indirectly. Again, a DSL would incorporate domain-specific constructs from the start.

Nevertheless, application libraries are formidable competitors to DSLs. For once, designing and implementing a DSL is far from easy. A domain expert is usually not an expert in language design, which is a distinct (meta-)domain of expertise in itself. How this situation can be improved is the main topic of this article, but even with improved DSL development tools, application libraries will remain the most cost-effective solution in many cases.

There are other factors complicating the relative merits of DSLs and application libraries. A case in point is Microsoft Excel. Its macro language is a DSL for spreadsheet applications which adds programmability to Excel's fundamental interactive mode. Using COM, Microsoft's software component technology, Excel's implementation has been restructured into an application library or tool box 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* and is described in more detail in [5]. Unlike 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.

Table 1: Some widely used domain-specific languages.

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

2 Existing Language Development Systems

As noted, DSL development is hard, requiring both domain knowledge and language development expertise. The development process can be speeded up by using a *language development system*. Some representative ones are listed in Table 2. They have widely different capabilities and are in widely different stages of development, but are based on the same general principle: *they generate programming tools from language descriptions.*

The input to these systems is a description of various aspects of the DSL to be developed in terms of specialized meta-languages. Some important language aspects are listed in Table 3. It so happens that the meta-languages used for describing these aspects are themselves DSLs for the particular aspect in question. For instance, an important language aspect is syntax, which is usually described in something close to BNF, the well-known DSL and *de facto* standard for syntax specification (Table 1). The corresponding tool generated by the language development system is a parser.

The tool generation capabilities of the language development systems listed in Table 2 are shown in Table 4. All of them can generate lexical scanners, parsers, and prettyprinters, many of them can produce syntax-directed editors, typecheckers, and interpreters, and a few can produce various kinds of software renovation tools. These tools are as useful for DSLs as they are for programming languages.

Although the various specialized meta-languages used for describing language aspects differ from system to system, they are usually *rule based*. For instance, depending on the system, the typechecking of language constructs has to

Table 2: Some representative language development systems.

System	Developed at
ASF+SDF Meta-Environment [7]	CWI and University of Amsterdam
Centaur [3]	INRIA Sophia-Antipolis
Eli [13]	University of Paderborn
Gem-Mex [1]	University of L'Aquila
PSG [2]	Technical University of Darmstadt
Software Refinery [16]	Reasoning Systems, Palo Alto
Synthesizer Generator [19]	Cornell University

Table 3: Some language aspects.

Syntax
Prettyprinting
Typechecking
Interpretation
Translation
Debugging

be described in terms of *attributed syntax rules* (an extension of BNF), *conditional rewrite rules*, *inference rules*, or *transition rules*.

Some examples of DSL development using a language development system are given in Table 5. 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. Risla is a language for describing loans and mortgages offered by banks. Its constructs are described in terms of their translation to Cobol. The LaCon system is not listed among the language development systems in Table 2, but will be discussed in the next section.

3 Toward Language Design Assistants

Actually, the language development systems listed in Table 2 incorporate little language design knowledge. Their main assets are the meta-languages they support, and in some cases a meta-environment to aid in constructing and debugging language descriptions. Even though tailored toward the language aspect they have to describe, these meta-languages are often hard to use. In many cases they were selected primarily for their favorable mathematical or logical properties rather than their user-friendliness. Also, these properties tend to become less important when language descriptions become large.

To turn language development systems into true *language design assistants* (LDAs) [11], improvements can be sought in various directions:

- Incorporation of *language concepts* and *design rules*.
- *Visual* or *semi-visual* meta-languages. The latter are partly graphical and partly textual.
- *Description by example* of some (necessarily limited) language aspects such as prettyprinting or syntax.

Clearly, these improvements are not simple. LDAs will become far more complex than language development systems, which are not particularly simple to begin with. Fortunately, LDAs may be approached step by step. Also, not all of the above improvements are completely new:

- The LaCon system (Table 5) allows domain experts to compose elements and properties of a DSL by simple yes/no decisions. It automatically checks consistency of user decisions, computes their consequences, and provides design style advice. To generate an implementation, it uses the Eli language development system (Table 2) as back-end.
- The Gem-Mex system (Table 2) already supports a semi-visual notation for the transition rules it uses to define the typechecking, interpretation and translation of language constructs.
- Less concretely, no longer pursued but nevertheless interesting plans for the Language Development Laboratory included a library of reusable language constructs, a knowledge base containing knowledge of languages and their compilers/interpreters, and a tool for language design [10, 15].

In the remainder of this section we focus on the incorporation of language design knowledge and on description by example of selected language aspects.

3.1 Incorporation of Language Concepts and Design Rules

Basically, the language designer using an LDA picks suitable language building blocks from the language knowledge base (language library), customizes them, and composes them into larger and larger language fragments. It may be necessary to add entirely new building blocks and concepts in the process, especially domain-specific ones, since it cannot be expected that everything required is already present in customizable form. The LDA provides feedback during customization and composition. Finally, the finished design is implemented by a language development system that serves as back-end to the LDA.

More specifically, the main elements and notions that would play a key role in an LDA are:

- **Language concept** Some examples are given in Table 6. They are rather diverse. For instance, some language concepts, such as "statement", "expression",

Table 4: Tool generation capabilities of representative language development systems.

System	Generated tools
ASF+SDF Meta-Environment	Scanner/parser (generalized LR), prettyprinter, syntax-directed editor, typechecker, interpreter, origin tracker, translator, renovation tools, ...
Centaur	Scanner/parser (LALR), prettyprinter, syntax-directed editor, typechecker, interpreter, origin tracker, translator, ...
Eli	Scanner/parser, typechecker, interpreter, translator, ...
Gem-Mex	Scanner/parser, typechecker, interpreter, translator, debugger, ...
PSG	Scanner/parser, syntax-directed editor, incremental typechecker (even for incomplete program fragments), interpreter
Software Refinery	Scanner/parser (LALR), prettyprinter, syntax-directed editor, object-oriented parse tree repository (including dataflow relations), Y2K/Euro tools, program slicer, ...
Synthesizer Generator	Scanner/parser (LALR), prettyprinter, syntax-directed editor, incremental typechecker, incremental translator, ...

Table 5: Examples of DSL development using a language development system.

DSL	Application domain	System used
Box [4]	Prettyprinting	ASF+SDF Meta-Environment
Cubix [14]	Virtual data warehousing	Gem-Mex
Risla [8]	Financial products	ASF+SDF Meta-Environment
(Various)	Data model translation	LaCon [12]

Table 6: Some language concepts.

syntax	sentence	abstract syntax	signature	term		
program	procedure	function	statement	expression	assignment	goto
conditional	loop	exception handler	input	output	operand	value
type	subtype	inheritance	strong typing	overloading	polymorphism	untyped
record	class	subclass	template class	object	module	parameterized module
import	export	actualization	instantiation	external	library	
concurrency	thread	process	exception	overflow		
scope	block	static scope	dynamic scope	local variable	global variable	static variable
file	parameter	argument				
state	heap	call stack	stack frame	side-effect	data flow	control flow
typechecking	interpretation	translation	transformation	abstract interpretation	data flow analysis	control flow analysis

or "loop", correspond more or less directly to language constructs. These are *language building blocks* (see below). Other ones, such as "scope" or "side-effect" have the character of an attribute to a language building block, while "state" or "call stack" refer to the dynamic behavior of programs or to the language's implementation. Note that some of the concepts in the bottom row correspond to language aspects mentioned in the previous section.

A more precise classification of language concepts from the perspective of their use in an LDA is currently being attempted. Some concepts may turn out to be more useful than others.

- **Language building block** *Language concept* corresponding more or less directly to a language construct, such as "statement", "expression", or "loop". It may have many attributes, which themselves correspond to *language concepts* of a different kind, such as (abstract) syntax, typechecking, interpretation, data and control flow, side-effects, exceptions, among others.
- **Relation** Relation between *language concepts.* May itself be a (higher-order) *language concept.* Attributes like "scope" and "side-effect" are unary relations. "Implementation" would be an example of a binary one.
- **Language knowledge base** Knowledge base of *language concepts* and their *relations.* It may include theories of the concepts it contains. These may range from rudimentary to elaborate.
- **Customization** The process of adapting *language building blocks* during the language design process by means of instantiation, transformation, or generation.
- **Composition** (Partly) *customized language building blocks* can be composed into larger language fragments. In this way the DSL is constructed step by step.
- **Constraint checking** is triggered by *customization* and *composition* to check, at least to some extent, the validity of the resulting *language building block.* The constraint checker is a parameter of the LDA design.
- **Design language** Meta-language allowing the user to formulate language design questions and interrogate/browse the *language knowledge base.*
- **Language knowledge representation language** Visual or semi-visual meta-language to express *language concepts* and their *relations.* It has well-defined sublanguages to express *language building blocks* and their attributes. These sublanguages are compiled to the corresponding meta-language(s) of the *language development system* that is used as a back-end.
- **Language development system** Back-end of the LDA and parameter of the LDA design. The LDA facilitates the language description process and then uses a language development system to generate the tooling from the finished description. As noted, LaCon uses Eli as back-end. In our case, the ASF+SDF Meta-Environment (Table 2) will be the back-end. Many of the other systems mentioned in the previous section would be suitable as well.

3.2 Description by Example of Selected Language Aspects

For some language aspects description by example (DBE) may become a user-friendly alternative to (or addition to) exhaustive description in terms of a specialized meta-language. We should not be too ambitious. DBE of the interpretation or translation of a language is not realistic, but prettyprinting or syntax may be sufficiently limited for DBE to become useful.

In fact, prettyprinting is a special case of text formatting. DBE of the latter is supported to some extent by Microsoft Word, which is capable of defining "styles" from user supplied examples. Another system supporting formatting by example is Tourmaline [17]. It contains rules that try to determine the role of different parts of a header in a text document, such as section number, title, author, and affiliation, as well as the formatting associated with each part. The results are displayed in a dialogue box for the user to inspect and correct.

DBE of syntax for the purpose of language development was already suggested in [6], but to the best of our knowledge not put into practice. Since then, considerable progress has been made both in the theory of syntax inference from example sentences as well as in the computing power that can be brought to bear on the inference process.

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. For instance, prettyprinting by example would try to infer general prettyprinting rules from a test suite of prettyprinted constructs or programs in the language under development. The inferred rules might be expressed in a language like Box (Table 5), the prettyprinting meta-language of the ASF+SDF Meta-Environment, but the test suite itself would not require this language.

The inference mechanism used in the implementation of DBE may range from very limited and predictable generalization, hardly deserving to be called inference, to full-fledged inductive inference. The latter has a probabilistic character in the sense that the inferred generalization is in some sense the simplest one that is correct on the examples given to it, but it need not be the one intended by the user. To find out, the user may have to inspect the generated rules, which is rather unattractive. And if the generalization turns out to be incorrect, it may again be hard to find out which examples have to be added or changed.

These are largely unsolved problems, partly offsetting the advantages of the stronger forms of DBE suffering from them, so simple, predictable DBE is to be preferred.

Acknowledgement

The basic idea for a language design assistant emerged in discussions with Paul Klint.

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] R. Bahlke and G. Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8:547–576, 1986.

[3] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The System. *ACM SIGPLAN Notices*, 24(2):14–24, 1989. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.*

[4] 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.

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

[6] 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.

[7] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping*, volume 5 of *AMAST Series in Computing.* World Scientific, 1996.

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

[9] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages — An annotated bibliography. *ACM SIGPLAN Notices*, 2000. to appear, also: www.cwi.nl/~arie/papers/.

[10] J. Harm, R. Lämmel, and G. Riedewald. The Language Development Laboratory. In M. Haveraaen and O. Owe, editors, *Selected papers from the 8th Nordic Workshop on Programming Theory (NWPT '96)*, Research Report 248, pages 77–86. University of Oslo, Department of Informatics, 1997. also: www.cwi.nl/~ralf.

[11] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, March 2000. also: CoRR E-print Server xxx.lanl.gov/abs/cs.PL/9911001.

[12] U. Kastens and P. Pfahler. Compositional design and implementation of domain-specific languages. In R. N. Horspool, editor, *Systems Implementation 2000*, pages 152–165. Chapman and Hall, 1998.

[13] U. Kastens, P. Pfahler, and M. Jung. The Eli system. In K. Koskimies, editor, *Compiler Construction (CC '98)*, volume 1383 of *Lecture Notes in Computer Science*, pages 294–297. Springer-Verlag, 1998.

[14] 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.

[15] R. Lämmel. An interactive language library. In H. Kuchen, editor, *Proceedings Arbeitstagung Programmiersprachen*, Arbeitsbericht 58. Westfälische Wilhelms-Universität Münster, September 1997. also: www.cwi.nl/~ralf.

[16] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Communications of the ACM*, 37(5):58–70, May 1994.

[17] B. A. Myers. Tourmaline: Text formatting by demonstration. In A. Cypher et al., editors, *Watch What I Do: Programming by Demonstration*, pages 309–321. MIT Press, 1993. also: www.acypher.com/wwid/.

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

[19] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, third edition, 1989.