

Domain-Specific Languages for Software Engineering

Jan Heering

CWI

Department of Software Technology

Kruislaan 413

1098 SJ Amsterdam

The Netherlands

Jan.Heering@cwi.nl

Marjan Mernik

University of Maribor

Faculty of Electrical Engineering and Computer Science

Smetanova 17

2000 Maribor

Slovenia

marjan.mernik@uni-mb.si

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 are BNF (syntax definition), HTML (hypertext markup), SQL (database queries), and VHDL (hardware design). So-called *fourth-generation languages* (4GLs) are usually DSLs for database applications.

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 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 DSL development can be made easier for domain experts not versed in language design is an important topic of current research.

The purpose of this minitrack is to bring together an international audience of researchers and practitioners actively involved in the design, development, and evaluation of DSLs. In “Two-level grammar as an object-oriented requirements specification language” by *Barrett Bryant* and *Beum-Seuk Lee* TLG, a wide-spectrum language based on two-level grammars, is discussed from the viewpoint of requirements specification. Successive refinement steps starting with natural language input lead to more detailed specifications that can be translated to VDM++, an object-oriented extension of VDM, which in turn is translated to Java, yielding an operational prototype.

Next, *S. Mauw*, *W.T. Wiersma*, and *T.A.C. Willemse* in “Language-driven system design” discuss the role of DSLs in the software engineering process. They propose a language-driven approach to system design combining techniques of domain analysis and software engineering. The former is used to obtain the basic concepts and notations of the application domain, while the latter is used to describe language aspects like syntax, semantics, and pragmatics.

In “Post-design domain-specific language embedding: A case study in the software engineering domain”, *Anthony M. Sloane* re-implements two DSLs used by the Odin software build system (the request language and the derivation graph language) by embedding them in Haskell, and compares the performance of the resulting implementation with that of the existing implementation of Odin in Eli.

Kevin A. Schneider and James R. Cordy in “AUI: A programming language for developing plastic interactive software” propose a DSL for developing interactive software that is easily adapted to new environments. To this end their DSL distinguishes a concrete user interface (CUI), an abstract user interface (AUI), and a computational core. The AUI is compiled to Gofer and then bound to a CUI.

In “Generalized reduction modified LR parsing for domain specific language prototyping”, Adrian Johnstone and Elizabeth Scott advocate the use of general context-free parsing in language development environments and present an improved version of Tomita’s generalized LR parsing algorithm for this purpose.

Next, in “ProcessNFL: A language for describing non-functional properties”, Nelson S. Rosa, Paulo R. F. Cunha, and George R. R. Justo define a DSL for describing non-functional requirements. Examples of such requirements are performance and security, among others, and these are used to illustrate the language.

In “High-level executable specification of the Universal Plug and Play architecture”, U. Glässer, Y. Gurevich, and M. Veanes give an abstract state machine semantics of Microsoft’s Universal Plug and Play architecture. This is then turned into an executable version using AsmL, the Abstract state machine Language, which is an executable meta-DSL for semantics description.

Finally, “Source code generator based on a proprietary specification language” by Kresimir Fertalj, Damir Kalpic, and Vedran Mornar explains a DSL for automatically generating source code (SQL statements) for database applications. The DSL consists of source code templates, standard program structures, and special statements for handling meta-data.