Special Issue on Domain-Specific Languages Guest Editors' Introduction

Ralf Lämmel[1] and Marjan Mernik[2]

[1] Free University Amsterdam, The Netherlands
[2] University of Maribor, Slovenia

15 December, 2001

In need of domain support

Programming languages are a programmer's most basic tools. A language is suitable for a programming problem if it makes the programmer productive, and if it allows the programmer to write highly scalable, generic, readable and maintainable code. There are various ways to classify or to group programming languages. One can, for example, focus on the programming paradigm in such a classification (cf. imperative vs. functional vs. logic vs. constraint vs. object-oriented etc., or multi-paradigm). One can also consider 2GLs, 3GLs, 4GLs, or very high-level languages. We want to emphasize the division of programming languages into general-purpose and domain-specific languages. Here, we admit that there is a gray area hosting languages which neither belong strictly to the general-purpose nor to the domain-specific group. With general-purpose languages, one can address large classes of problems e.g., scientific computing, business processing, symbolic processing while a domain-specific language (DSL) facilitates the solution of problems in a particular domain. To this end, a DSL provides built-in abstractions and notations specific to the domain of concern. In one domain, partial differential equations might be of central importance, and hence they deserve the status of a proper abstraction. In many domains, graphical notations are common, and hence a DSL that is suitable for such a domain would be a visual language. DSLs are usually small, more declarative than imperative. The poor man's approach to DSL implementation is to implement a library to be used via an API in the general-purpose language of choice. This approach is not suitable when the domain requires:

- dedicated notation, that is, syntax,
- dedicated abstraction mechanisms,
- dedicated scoping or typing rules,
- domain-specific optimizations,
- domain-specific analyses,
- domain-specific error reporting,
- etc.

A historical note

DSLs have been used in various domains such as robot control, animation, music composition, financial product design, description and analysis of abstract syntax trees, web computing, 3D animation, reverse engineering, modelling of reactive systems, just to mention a few. The idea of DSLs is presumably as old as the notion of programming languages. The APT language (Automatically Programmed Tools), developed in the 1955 at MIT, can be regarded as one of the first DSLs. One of the first research groups systematically working on DSLs was N. J. Lehmann's group at the Technical

University of Dresden in East-Germany. This group started in the nineteen-seventies to pursue fundamental research on the field, and to develop DSLs commissioned by industry on a regular basis. At this time, one did not use the term DSL but "specialized language" or "application-oriented language", and in German "Fachsprache" or "Spezialsprache".

Trade-offs

Applications of DSLs have clearly illustrated the advantages of DSLs over general-purpose languages in areas such as productivity, reliability, maintainability and flexibility. However, the benefits of DSLs are not for free. Without appropriate methodology and tools, the upfront investment for DSL support can exceed the savings obtained by using a DSL for application development. Since the costs of DSL language development and maintenance have to be taken into account, one of the main questions is "When and how to design and implement a domain-specific language?". In similarity to general software development, one can identify the following phases for the development of a DSL: analysis, design, implementation, and finally their deployment. In the analysis phase, the problem domain is identified, and domain knowledge has to be gathered. Then, the DSL is designed to concisely describe applications in the domain. The implementation phase can be done using one of the following approaches:

- A proper compiler or interpreter is developed where standard compiler tools can be used, or tools dedicated to the implementation of DSLs.
- The DSL is modelled as an embedded language or as a domain-specific library where some form of abstraction that is available in an existing language (e.g., functions in C or Haskell) is employed to capture domain-specific operations.
- The DSL is supported via preprocessing or macro processing where DSL constructs are translated to statements in a base language.
- The DSL is implemented by means of an extensible compiler or interpreter, that is, the DSL constructs are added to the existing language implementation, e.g., via reflection.

The above steps show that the development of a domain-specific language is itself a significant software engineering effort, requiring considerable investment of time and resources. In practice, we have to seek for a trade-off between level of DSL support, usability, efficiency, evolvability of the implementation, available resources, and other criteria. The interpretation/compilation approach from above, for example, is very much suited to achieve a full-blown implementation of a DSL. One might employ simple parser generators, or more sophisticated compiler compilers based on executable language definitions—as common for the implementation of general-purpose languages, too. The other approaches listed above (embedding, preprocessing, extensible compiler/interpreter) can be more efficient and attractive in particular cases of DSL development.

Challenges in the DSL field

Without restricting ourselves to DSLs, one can say that language design and implementation are apparently never-ending research activities. If we, for example, look back at the last ten years, we see that major breakthroughs in the field of *modular* language definition and implementation were achieved (cf. modular SOS, action semantics, abstract state machines, use of object-oriented methods and generic programming). Further progress of the DSL field can be expected from these results. The now more and more ubiquitous role of computers, their use in all domains of business and every-day life implies, without any doubt, that DSLs are even more frequently needed in future. End-users need to be enabled to write programs in the notations which are most suitable in a given application domain. Research in the DSL field will hence focus on methods and tools to simplify and to discipline the development of DSLs. Besides these general conceptual challenges, the development of actual DSLs for emerging domains forms a continuous research activity in the field. Finally, we expect an amalgamation of the DSL theme and other research topics, e.g., the more recent field of aspect-oriented programming.

Papers in the special issue

The papers accepted for the CIT special issue on domain-specific languages provide the reader with a contemporary analysis of the spectrum of DSL approaches, with concepts for DSL design, with new shining examples of DSLs, and with discussions of lightweight vs. heavyweight tools, or language support for DSL development. In this manner, the selected papers contribute to the answer of the question "When and how to develop a DSL?". The papers were selected from eleven submissions. There are two invited contributions, and five regular papers. For reasons of page count, the special issue consists of two parts which appear in two consecutive CIT issues. We first list all the papers appearing in the special issue, and then we will shortly introduce the papers included in the present CIT issue.

- D. Wile. Supporting the DSL Spectrum (invited paper)
- A. van Deursen, P. Klint. Domain-Specific Language Design Requires Feature Descriptions (invited paper)
- R. van Engelen. ATMOL: A Domain-Specific Language for Atmospheric Modeling
- A. Berlea, H. Seidl. `fxt` - A Transformation Language for XML Documents
- J. Gil, Y. Tsoglin. *Jamoos* - A Domain-Specific Language for Language Processing
- H. Kienle, D. Moore. `sgmn`: Rapid Prototyping of Small Domain-Specific Languages
- J. Aycock. The Design and Implementation of SPARK, a Toolkit for Implementing Domain-Specific Languages

Part II Arie van Deursen's and Paul Klint's invited contribution addresses a more specific topic, namely, the relation between the notion of feature diagrams and DSLs. Feature diagrams are useful to structure application domains. The invited paper authors indicate the potential of feature diagrams (or a related textual description language) for supporting DSL design. This conception integrates the process of domain analysis and DSL design, and it also imposes more structure on the DSL design process as such.

Alexandru Berlea and Helmut Seidl contribute a DSL approach to the application domain of XML processing. The authors present an architecture for XML transformers which is based on functional programming in SML. The corresponding tool `fxt` is implemented following the preprocessing scheme. That is, the ingredients of a transformation specification are certain forms of patterns and rules which can be enriched by SML code. Such specifications are translated to plain SML while interacting with an efficient pattern matcher. The processing model that underlies `fxt` deliberately deviates from the de-facto standard model underlying XSLT/XPath. This and other design decisions but also certain implementational choices lead to a competitive implementation.

The last two regular papers advocate a similar path to the rapid development of DSLs. Holger M. Kienle and David L. Moore describe the tool `sgmn` which provides a macro language for processing parse trees and producing output in this course. John Aycock reports on the toolkit SPARK designed for implementation of DSLs in the language Python. Both papers share that the corresponding toolkits are lightweight, that the approaches are simple (if not restricted), but the intended audience is well-defined. Also, these authors provide evidence that their lightweight approaches are useful in practice: plenty of actual DSL applications are pointed out, e.g., in the domains of reverse engineering and decompilation tools, operating system configuration, and interface description.

Acknowledgement

We want to thank Arie van Deursen, Paul Klint and Dave S. Wile for their brilliant invited contributions to the special issue. Another important success factor in the project were the referees. Our thanks go to Mikhail Auguston, Dmitry Boulytchev, Mark van den Brand, Kyung-Goo Doh, Gopal Gupta, Gorel Hedin, Jan Heering, Sam Kamin, Günter Kniesel, Tobias Kuipers, Wolfgang Lohmann, Katharina

Mehner, Pierre-Etienne Moreau, Günter Robbert, Anthony Sloane, Andrey A. Terekhov, Eelco Visser, and Eric Van Wyk. Last but not least, we are grateful for the support by the publisher for this special issue. We appreciate the very smooth cooperation with the CIT editor prof. Sven Lončarić and with Vesna Smoljanović from the CIT Editorial Office.

Ralf Lämmel
Free University Amsterdam
The Netherlands
e-mail: Ralf.Laemmel@cs.vu.nl

Marjan Mernik
University of Maribor
Slovenia
e-mail: marjan.mernik@uni-mb.si