Requirements for Domain-Specific Languages

Dimitrios S. Kolovos, Richard F. Paige, Tim Kelly, and Fiona A.C. Polack
Department of Computer Science, University of York
[dkolovos,paige,tpk,fiona]@cs.york.ac.uk

1. Motivation

A domain-specific language (DSL), whether used for model-driven development or programming, is a piece of critical infrastructure that is developed during the system engineering process. As such, a DSL has its own lifecycle, which in turn may encapsulate the lifecycles of many other system development projects. Understanding the requirements for DSLs in general, and in specific project contexts, is critical in order to improve DSL quality and ensure a direct correspondence between the requirements for system engineering projects and the functionality provided by the language. The quality attributes of a DSL – and its supporting environment (e.g., virtual machines, debuggers, integrated environments) – will have an impact on the quality attributes of the overall systems development process, and the resulting products.

We present a partial requirements analysis for DSLs in general, focusing on relevant stakeholders, the system boundary (i.e., where DSLs end and general purpose languages start), and a core set of requirements that are relevant for any DSL. We then discuss open questions, particularly focusing on requirements refinement, wherein more specific domain information needs to be used.

Our discussion is intended to be generic: we do not distinguish between domain-specific modelling and programming languages (except where noted). We therefore refer to *descriptions* as the construct produced by using a DSL. Specific instances of descriptions may be models or programs.

2. DSL Stakeholders

There are three typical DSL stakeholders:

- *System/software engineers*, who are responsible for choosing or implementing an appropriate DSL.
- *Customers* (e.g., business analysts), who are responsible for providing feedback on descriptions (models, programs) produced using a DSL.
- *Developers*, who are responsible for constructing and managing DSL descriptions.

A current trend towards end-user programming suggests that in some contexts the roles of customer and developer may be combined [8].

3. DSL System Boundary

An important issue is identifying the boundary of DSLs: what constitutes a DSL, and what does not? A relative comparison can be made: DSLs show an increased correspondence of language constructs to domain concepts when contrasted with general purpose languages. As a result, a DSL will more accurately represent domain practices and will more accurately support domain analyses (e.g., fault propagation for safety critical systems [9]). Furthermore, a DSL may defer handling of systems engineering problems: other phases of the engineering process, other tools, and other languages may need to be used. Finally, a DSL is often computationally incomplete. We suggest that this is not, by itself, a defining characteristic of DSLs, but one that is a result of aligning the constructs of the DSL with the concepts of the domain of interest.

When we design and use a DSL, we lose generality, and typically lose the ability to apply general-purpose tool support. We also can lose the ability to exploit widespread expertise.

4. Core Requirements for a DSL

We now discuss a number of core requirements for DSLs in general. We make reference to requirements and principles for the design of programming languages [1,2], and also fundamental work on application generators and little languages [3,4,6,7]. We then discuss the issue of requirements refinement, and discuss how additional requirements can be identified for specific instances of DSLs.

Some of the requirements for general-purpose programming languages apply directly to DSLs. However, we argue that these requirements will differ in terms of their relative importance for DSLs compared with general-purpose languages.

The core requirements for a DSL are as follows:

- **Conformity:** the language constructs must correspond to important domain concepts.
- **Orthogonality:** each construct in the language is used to represent exactly one distinct concept in the domain.
- **Supportability:** it is feasible to provide DSL support via tools, for typical model and program management, e.g., creating, deleting, editing, debugging, transforming.
- **Integrability:** the language, and its tools, can be used in concert with other languages and tools with minimal effort. This is essential to integrate the DSL with other facilities used in the engineering process. An alternative requirement for DSLs is **extensibility,** i.e., that the DSL (and its tools) can be extended to support additional constructs and concepts. However, we suggest that integrability is to be preferred as a requirement for DSLs as it preserves *semantic coherence* of the DSL, as well as the desirable requirements of conformity and orthogonality.
- **Longevity**: the DSL should be used and useful for a non-trivial period of time in order to ensure tool support, and to make it possible to quantify to the DSL stakeholders the *payoff* obtained from using the DSL. There is, of course, an assumption with this requirement (and with the use of DSLs in general) that the domain under consideration persists for a sufficiently lengthy period of time to justify the cost of building a DSL and supporting tools.
- **Simplicity:** this is a generally desirable language requirement: a language should be as simple as possible in order to express the concepts of interest and to support its users and stakeholders in their preferred ways of working.

- **Quality:** the language shall provide general mechanisms for building quality systems. This may include (but is not limited to) language constructs for improving reliability (e.g., pre- and postconditions), security, safety, etc.

There are additional requirements that, while useful and desirable, need not be necessary for building all DSLs. These are as follows:

- **Scalability:** the language provides constructs to help manage large-scale descriptions. Of course, some DSLs will only be used to build small systems.
- **Usability:** this includes requirements such as space economy, accessibility, understandability – characteristics that are desirable, and which may be partly covered by the core requirements (e.g., simplicity can help promote understandability).

Many of the above requirements are also applicable to general-purpose modelling and programming languages. However, as we mentioned earlier, their relative importance differs when comparing general-purpose languages and DSLs. We have listed the requirements for DSLs in what is, in our opinion, a reasonable order of priority, but of course experiment and measurement is needed to validate this,

5. Open Issues

There are a number of open issues with respect to DSL requirements that require further investigation.

1. How do requirements refine when a *specific* domain is considered? The previously mentioned requirements apply to all DSLs, but each domain of interest will introduce further requirements that will undoubtedly conflict. For example, consider the domain of safety critical systems where systems need to be certified as acceptably safe to use against a particular safety standard. Normally, evidence must be gathered for the certification process, e.g., via automated testing, peer reviews, and automated analysis. This in turn may place a requirement on a DSL for safety critical systems of *analysability*, i.e., the descriptions created are amenable to automated analysis. In order to enable analysis, the DSL may need to be augmented, e.g., with information on failure models of components. This extension may contradict the requirement for conformity.

2. How can we measure the cost-versus-benefit of using DSLs as opposed to general-purpose languages? Many of the above requirements assume that a cost-benefit analysis has been carried out (e.g., the longevity requirement). Such information will be of importance in validating other requirements, e.g., integrability and simplicity.

3. Using DSLs has a number of side-effects, in particular, the implicit use of a number of (loosely coupled) languages throughout development, the use of a federation of tools versus an integrated development environment, etc. Of particular interest is the use of proprietary versus open standards for defining languages (e.g., using MOF 2.0 to build modelling languages) and for building tools (e.g., using Eclipse EMF/Ecore to build a modelling tool). An open issue

is how to isolate each side-effect and identify the overall effect of each decision on the cost-benefit analysis.

4. Finally, DSLs are often claimed to provide less reusability, since the language may be more abstract, less expressive, and possessing of less tool support. But DSLs are generally simpler and smaller than general-purpose languages. Identifying requirements for DSLs has highlighted this issue, and it is important to clarify the impact of using DSLs on language reuse if we are to better understand the costs and benefits of their application.

References

1. C.A.R. Hoare. Hints on Programming Language Design, SIGACT/*SIGPLAN Symposium on Principles of Programming Languages*, Boston, Oct. 1973.
2. N. Wirth. On the Design of Programming Languages, *IFIP World Congress 1974*.
3. J.C. Cleaveland. Building application generators. *IEEE Software*, July 1988.
4. A. van Deursen and P. Klint. Little languages: little maintenance? *Journal of Software Maintenance* 10:75-92, 1998.
5. R.F. Paige, J.S. Ostroff, and P.J. Brooke. Principles of modeling language design. *Info. Soft. Tech.* 42(10), Elsevier, June 2000.
6. M. Mernik, J. Heering, A.M. Sloane. When and how to develop domain-specific languages. *ACM Comp. Surv.* 37(4), December 2005.
7. D. Wile. Lessons learned from real DSL experiments. *Science of Computer Programming* 51 (2004), 265-290.
8. J. Ruthruff, M. Burnett, G. Rothermel. Interactive fault localization techniques in a spreadsheet environment. *IEEE Trans. Soft. Eng.* 32(4), April 2006.
9. M. Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Proc. FESCA 2005, ENTCS 141(3),* Elsevier, April 2005.