Chapter 16
Ontology-Supported Design of Domain-Specific Languages: A Complex Event Processing Case Study

István Dávid
Budapest University of Technology and Economics, Hungary

László Gönczy
Budapest University of Technology and Economics, Hungary

ABSTRACT

This chapter introduces a novel approach for design of Domain-Specific Languages (DSL). It is very common in practice that the same problems emerge in different application domains (e.g. the modeling support for complex event processing is desirable in the domain of algorithmic trading, IT security assessment, robust monitoring, etc.). A DSL operates in one single domain, but the above-mentioned cross-domain challenges raise the question: is it possible to automate the design of DSLs which are so closely related? This approach demonstrates how a family of domain-specific languages can be developed for multiple domains from a single generic language metamodel with generative techniques. The basic idea is to refine the targeted domain with separating the problem domain from the context domain. This allows designing a generic language based on the problem and customizing it with the appropriate extensions for arbitrary contexts, thus defining as many DSLs and as many contexts as one extends the generic language for. The authors also present an ontology-based approach for establishing context-specific domain knowledge bases. The results are discussed through a case study, where a language for event processing is designed and extended for multiple context domains.

PROBLEM FOUNDATION

Domain-Specific Languages (DSL) are key elements of model driven engineering. They serve for describing the concepts of a given field (domain) in order to enable accurate and compact modeling (Fowler, 2010).

DOI: 10.4018/978-1-4666-6042-7.ch016

The domain itself is the most major factor in designing DSLs, since it fundamentally determines the concepts of the modeling language and interrelationships among them. Our research points out that the influence of the domain is a complex phenomenon and if investigated from a sufficiently high abstraction level, a more efficient DSL design methodology can be employed.

The targeted domain can be separated into two aspects: the problem domain and the context domain. The problem domain is the set of requirements being actually modeled with the language and is associated with at least one but usually with many context domains. Both aspects are required to characterize a DSL. This can be formally defined in the following way.

Let *P* denote the set of problem domains and *C* denote the set of context domains. Given this notation, the set of every possible domain which can be targeted using a domain-specific language would be:

$$D = P \times C.$$

(Of course, not every problem-context combination results in a meaningful domain.) A domain-specific language targets exactly one *d* domain from the set *D*. A family of languages for a given problem domain *p* is a set of languages with a fixed problem but with an arbitrary set of context domains:

$$L_p = p \times C_p,$$

where C_p denotes the subset of context domains which would create a meaningful domain with *p*.

This level of refinement of the targeted problem is important because it is common that the same problem arises in several contexts (such as online processing of high volume information in the domains of sensor networks, IT infrastructure management, and stock market informatics). Implementing a "general" DSL, which is sufficient for proper modeling in all the contexts, would lead to losing the desired description power. On the other hand, implementing a language family (i.e. a DSL for every context) would produce significant overlaps among the languages' concepts – because of the same targeted problem.

A solution, which allows the separation of reusable language parts and context-dependent elements, would fit well with the overall Model-Driven Architecture (MDA) approach, where platform-independent elements of high abstraction level are getting merged with platform-definition models to yield a more concrete model (Aßmann & Zschaler, 2006).

It is important to understand that from now on, we strictly distinguish one DSL from another by investigating both its problem domain and its context domain. This means extending a common "language-stub" of a given problem for arbitrary contexts would result in as many different DSLs as many contexts we take. This is, of course in accordance with the practice; we just introduced here a different definition for the *domain* itself.

This chapter presents a technique which facilitates semi-automated engineering of families of domain-specific languages closely related to each other because of the same targeted problem but in different context domains. The approach hence aims at the creation of a framework where reusable and custom context-specific language elements help an efficient development. Reusable parts depict the core of the problem; context-specific customizations aim at extending these parts with the knowledge from the targeted context domain. Therefore, reusable elements (e.g. common domain entities, relationships, data types) happen to be generic over different contexts and remain intact despite the context-specific customizations. Furthermore, we do not require the reusable parts being a DSL of full value, instead: generic parts with context-specific extensions shall piece a DSL together.

The approach can also handle the problem of the changing requirements and the consequent changes in the DLS's language structure (Spinellis, 2001).

Context-specific extensions are derived from some kind of domain knowledge related to the targeted context domain. An important question is how domain knowledge can be represented in order