Chapter 40
Comparison between Internal and External DSLs via RubyTL and Gra2MoL

Jesús Sánchez Cuadrado
Universidad Autónoma de Madrid, Spain

Javier Luis Cánovas Izquierdo
École des Mines de Nantes – INRIA – LINA, France

Jesús García Molina
Universidad de Murcia, Spain

ABSTRACT

Domain Specific Languages (DSL) are becoming increasingly more important with the emergence of Model-Driven paradigms. Most literature on DSLs is focused on describing particular languages, and there is still a lack of works that compare different approaches or carry out empirical studies regarding the construction or usage of DSLs. Several design choices must be made when building a DSL, but one important question is whether the DSL will be external or internal, since this affects the other aspects of the language. This chapter aims to provide developers confronting the internal-external dichotomy with guidance, through a comparison of the RubyTL and Gra2MoL model transformations languages, which have been built as an internal DSL and an external DSL, respectively. Both languages will first be introduced, and certain implementation issues will be discussed. The two languages will then be compared, and the advantages and disadvantages of each approach will be shown. Finally, some of the lessons learned will be presented.

INTRODUCTION

Software applications are normally written for a particular activity area or problem domain. When building software, developers have to confront the semantic gap between the problem domain and the conceptual framework provided by the software language used to implement the solution. They must express a solution based on domain concepts using the constructs of a general purpose programming language (GPL), such as Java or C#, which typically leads to repetitive and error prone code. This

DOI: 10.4018/978-1-4666-6042-7.ch040

encoding task is considered to be "not very creative, and more or less waste or time," and existing code maintenance is difficult (Dmitriev, 2004). Since the early days of programming, domain-specific languages (DSLs) have therefore been created as an alternative to using GPLs.

DSLs allow solutions to be specified by using concepts of the problem domain, thus reducing the semantic gap between them, and thereby improving productivity and facilitating maintenance, as a number of studies and case studies report (Weiss & Lai, 1999; Ledeczi, Bakay, Maroti, Volgyesi, Nordstrom, Sprinkle & Karsai, 2001; Kelly & Tolvanen, 2008; Kosar, Mernik & Carver, 2011). DSLs are not new (Bentley, 1986), for instance SQL, Pic or Make are well-known examples, but the interest in them has increased considerably in the last decade with the emergence of *model-driven development* paradigms ("MDA Guide," 2001; Kelly & Tolvanen, 2008; Greenfield, Short, Cook & Kent, 2004; Voelter, 2008), which provide systematic frameworks for the building and use of DSLs, their core being meta-modeling.

Model-driven paradigms are based on three basic principles. Firstly, a software application is partially (or totally) described using models, which are high-level abstract specifications, rather than using solely a GPL. Secondly, these models are expressed with DSLs which are created by applying meta-modeling (i.e. the DSL abstract syntax is represented as a meta-model). Thirdly, automation is achieved by means of model transformations which are able to directly or indirectly transform models (e.g., DSL programs) into the final code of the application by creating intermediate models. Two kinds of model transformation languages are therefore needed (Czarnecki & Helsen, 2006): model-to-model transformation languages, which allow us to express how models are mapped into models, and model-to-text transformation languages, which allow us to express how models are mapped into text (e.g., GPL code). Model-based techniques can also be applied in software modernization tasks, and a third kind of model transformation with which to extract models from legacy software artifacts (e.g., GPL code or a XML document) is then involved, which is normally called text-to-model transformation.

A DSL normally consists of three basic elements: abstract syntax, concrete syntax, and semantics. The abstract syntax expresses the construction rules of the DSL without notational details, that is, the constructs of the DSL and their relationships. Meta-modeling provides a good foundation for this component, but other formalisms such as BNF have also been used over the years. The concrete syntax defines the notation of the DSL, which is normally textual or graphical (or a combination of both). There are several approaches for the semantics (Kleppe, 2008), but it is typically provided by building a translator to another language (i.e., a compiler) or an interpreter.

Several techniques have been proposed for the implementation of both textual DSLs (Fowler, 2010; Mernik, Heering & Sloane, 2005) and graphical DSLs (Kelly & Tolvanen, 2008; Cook, Jones, Kent & Wills, 2007). In this work we focus on textual DSLs, and particularly consider two kinds or styles according to the implementation technique used: external DSLs and internal DSLs. An external DSL is typically built by creating a parser that recognizes the language's concrete syntax, and then developing an execution infrastructure if necessary. An internal DSL, however, is implemented on top of a general purpose language (the host language), and reuses its infrastructure (e.g., concrete syntax, type system and run-time system), which is extended with domain specific constructs. The DSL is therefore defined using the abstractions provided by the host language itself. For instance, in an object-oriented language, method calls can be used to represent keywords of the language. Languages with a non-intrusive syntax (e.g., LISP, Smalltalk or Ruby) are well suited for use as host languages.

A number of design decisions must be made when building a DSL, such as those related to its concrete syntax, how the language semantics is