Automated Transformation of Statements within Evolving Domain Specific Languages

Peter Bell
SystemsForge
9/10 St Andrew Square
Edinburgh EH2 2AF
[44] 131.292.6240 (UK)
[1] 646.519.3505 (US)
pbell@systemsforge.com

Abstract. One of the biggest challenges in Domain Specific Modeling is handling the inevitable changes to Domain Specific Language grammars (meta models) as they evolve over time. This paper starts by providing an overview of our use case - a high volume Software Product Line using a databased concrete syntax for storing large numbers of statements in a collection of external Domain Specific Languages.

It then outlines research in related fields - from API versioning to database migrations - that could be applied to the problem of automating the transformation of statements between versions of a Domain Specific Language. It then presents an initial partial catalog of potential transformations that could be used to automate the process of transforming DSL statements at a tooling level, making it much easier for modelers to support evolutions in their DSL grammars.

Categories and Subject Descriptors D.2.11 [**Software Engineering**]: Software Architecture: Languages

General Terms Design, Experimentation, Languages, Theory

Keywords Evolution, Versioning, Domain Specific Language, Model to Model Transformation, Meta Modeling, Language Oriented Programming.

1. Introduction

Over time, most programming languages evolve. While General Purpose Languages (GPL's - Java, C#, Ruby, etc.) generally evolve slowly, Domain Specific Languages (DSLs) often have to evolve rapidly to keep up with a growing understanding of the domain concepts within a project. In addition, with wider acceptance of Agile [1] programming practices it is particularly important for the development of DSLs to be an Agile endeavor, allowing for experimentation and iterative design rather than attempting to perfect the DSLs before starting to build a system.

There is not yet a rich literature on managing the evolution of DSLs, but there are many comparable fields that can suggest patterns useful for managing changes in DSL grammars in an elegant manner. This paper starts by providing an overview of the use case that is driving this work. It then looks at some of the work in related fields from which inspiration can be drawn and then suggests an approach strongly influenced by database migrations and refactoring style patterns for managing and partially automating the transformation of DSL statements as the grammar they need to conform to evolves.

It is important to note that the focus of this paper is the automation of the process of transforming statements within a DSL based on a known change to the grammar for that DSL. It is not concerned with the separate topic of attempting to automate the process of extending the grammar for a DSL. The paper is concerned with automating the evolution of the **statements** within the DSL rather than automating the evolution of the DSLs.

2. Requirements

At SystemsForge, we generate custom web applications for Small to Medium Sized Businesses [5]. We have developed a Software Product Line for developing applications using a combination of a Feature Manager, a Decision Support System, a collection of DSLs for customization and an easy to extend

framework allowing for the addition of custom code where the DSLs are not yet rich enough to implement a specific use case.

We store all of the DSL statements in a database for ease of reuse. Features are comprised of a collection of essential and optional DSL statements. The Decision Support System allows non-technical users to select optional statements by answering questions organized in a hierarchical decision tree. Our semi-technical design partners can then add custom statements in any of the system DSLs to customize applications without having to resort to programming in a GPL. For each project the essential, selected optional and any custom metadata statements are merged into a collection of XML documents which are then consumed by a dynamic framework that interprets the XML, merges it with any custom code and provides the application functionality required.

With the goal of being able to generate thousands of applications a year, we need an automated manner for handling the evolution of our DSLs as we will have hundreds of thousands of statements that will need to be transformed to handle the evolving grammars of the languages in which they were written. We have already developed a system for generating the database schemas for storing the statements based on a grammar. More recently, our research has been focused on an automation-assisted approach to transforming DSL statements based on changes in the DSL grammars. For our use case, we are interested only in the evolution of external DSLs [7] - as opposed to internal [7] (sometimes called embedded [10]) DSLs.

3. Research

We were not able to find a great deal of literature directly concerning approaches to automating DSL statement evolution. There were two projects that looked promising. Gerardo de Geest is involved with some work at Avanade on model:model transformations for DSLs using an XML concrete syntax. Genvoca [4] is an interesting approach that looks at the idea of applying refinement functions to add features to a language. It is focused on program evolution (both the interfaces and their implementations), but the concept is equally relevant to DSL evolution.

In reviewing some currently available Language Workbenches [8], Microsoft DSL Tools [13] and openArchitectureWare [14] did not appear to have any facilities designed for automating the migration of models as the DSLs evolved. MetaEdit+ [11] [12], however, had what appeared to be a well thought out built-in facility for automatically evolving models in concert with their associated meta-models.

When we looked a little further afield we found a number of approaches that could be applicable to automating DSL statement evolution. Given that we store our metadata in a database and that a database schema can be looked at as a type of grammar for a language with a databased concrete syntax, we started by looking at approaches to evolving database schemas. Agile database techniques [2], database refactoring [3] and the migrations [17] approach taken in Ruby on Rails all provided useful ideas.

An API can be considered a type of simple DSL, so we also investigated approaches to API versioning [6]. Finally, we looked at some of the materials available relating to XML Schema evolution including an online repository containing information on a range of approaches to schema evolution [16]. We also felt that the approach taken in Refactoring [9] by cataloging a collection of specific patterns would fit well with the problem space even though we are interested in changes that add to and remove from the total functionality available as well as refactorings that simply implement the same functionality in a different manner.

4. Initial Results

4.1 Types of Transformation

We decided to start by considering three key grammatical Items: Elements, Attributes (Properties include whether they are required and their data type) and Relationships to sub-Elements (properties include their Multiplicities). See Figure 1.

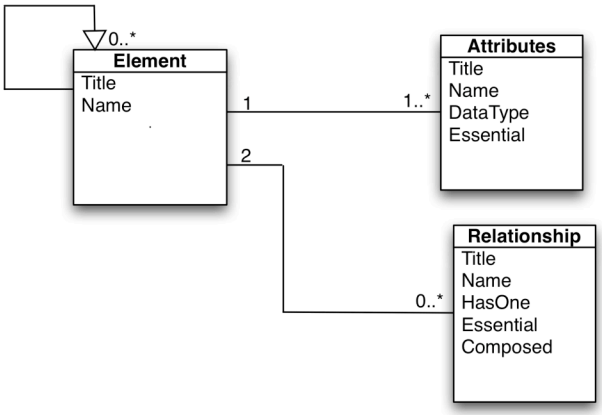

Figure 1. The SystemsForge Meta-grammar

The basic atomic units of transformations are; add, edit, copy, deprecate and delete.

Add - An Item (Element, Attribute or Relationship) can be added to a DSL to make it more expressive.

Edit - An Item can have its name, Properties and/or Relationships changed, perhaps making an Attribute optional or changing the multiplicity of a Relationship.

Copy - Information can be copied between Items. For example, as part of the "change Attribute to Element" transformation, the values for an Attribute could be copied to the new Element to avoid data loss within the transformation.

Deprecate - Deprecation is an important concept that allows for the expression of the intent to Delete an Item in some future version. Some systems such as pure:variants [15] allow for multiple-levels of deprecation. We have not yet determined how strictly we will implement deprecation, but initially we plan on "warning" if deprecated Items are found in a statement and removing them from editing tools so they cannot be added to new statements.

Delete - Occasionally it is necessary to remove an Element from a grammar.

While we are considering only five atomic units of transformation, in practice it is sometimes more useful to consider higher-level transformations which may combine more than one atomic transformation. For example, a common transformation is "change Attribute to Element" which replaces an Attribute of an Element with a related sub-Element. This transformation is performed when an Attribute needs to have Attributes or sub-Elements of its own or when it requires a multiplicity greater than 1. This can be thought of as three atomic transformations: add Element, copy from Attribute to Element and remove Attribute. However, it is more useful to be able to express the higher level of intent by having a distinct transformation for "change Attribute to Element".

4.2 On Interfaces

In this paper we are not yet making a distinction between DSL grammars and the interfaces to those grammars. The distinction is important when you have multiple interacting DSLs. Changes that may affect statements in other DSLs are clearly both more serious and more problematic than changes that will affect only statements within the DSL being transformed. One approach to this problem is by annotating a subset of the Items within a grammar as being part of its Interface and distinguishing transformations that affect those Items from transformations that affect other Items. While such an approach is promising, we have not yet had time to fully think through or catalog a separate collection of transformations related to DSL interfaces, so the current development is initially naively focused only on the effects of evolution within a single DSL.

4.3 An Initial Catalog

Figure 2 (next page) contains a non-exhaustive list of some of the most common transformations we have come across to date when evolving our DSLs.

5. Conclusions

We believe that there is real value in developing a catalog of grammatical transformations - similar to the way that refactorings have been cataloged. We can then develop tooling for automatically migrating most transformations and facilitating the migration of non-automatable transformations at a tooling level, removing the necessity for modelers to have to develop their own solutions to transforming statements as their DSLs evolve.

Of course, transformations are predicated on the concepts used to express the grammars (the meta-meta models), so for any given meta-grammar, the catalog of transformations will vary.

The next step in the project is to develop a comprehensive catalog of grammatical transformations based on the types of grammars and constraints that can be described using XSD Schemas in the form of a DSL for describing the transformations required. We then plan to develop tooling for handling version transformations for both database and XML concrete syntaxes. Once we have the naive case working for single DSLs, we then plan to add the concept of DSL interfaces, and to attempt to develop tooling for supporting transformations across collections of DSLs.

References

[1] Agile Alliance, http://agilemanifesto.org.

[2] Ambler, S. Agile Database Techniques: Effective Strategies for the Agile Software Developer. Wiley 2003 ISBN 0-471-20283-5.

[3] Ambler, S. Refactoring Databases: Evolutionary Database Design. Addison-Wesley Professional 2006 ISBN 0-321-29353-3.

[4] Batory, D., Johnson, C., Macdonald B. and Von Heeder, D. Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 2, April 2002, Pages 191-214.

[5] Bell, P. A Practical High Volume Software Product Line. OOPSLA 2007.

[6] Dig, D., Manzoor, K., Johnson, R., and Nguyen, T. Refactoring-Aware Configuration Management for Object_Oriented Programs. ICSE '07.

[7] Fowler, M. http://www.martinfowler.com/bliki/DomainSpecificLanguage.html.

[8] Fowler, M. http://www.martinfowler.com/articles/languageWorkbench.html.

[9] Fowler, M, Beck, K, Brant, J, Opdyke, W, Roberts D. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional 1999 ISBN 0-201-48567-2.

[10] Freeman S. and Pryce N. Evolving an Embedded Domain-Specific Language in Java. OOPSLA 2006.

[11] Metacase, http://www.metacase.com/.
[12] Metacase, Documentation, http://www.metacase.com/support/45/manuals/mwb/Mw.html.
[13] Microsoft, http://msdn.microsoft.com/vstudio/DSLTools/.
[14] openArchitectureWare, http://www.openarchitectureware.org/.
[15] Pure-systems, http://www.pure-systems.com/Variant_Management.49.0.html.
[16] Rahm, E. and Bernstein, P, http://se-pubs.dbs.uni-leipzig.de.
[17] Ruby on Rails, http://api.rubyonrails.com/classes/ActiveRecord/Migration.html.

Figure 2. Initial Transformations

Name	Motivation	Automate?
Add Element	Add a new element to broaden the expressiveness of a DSL.	Yes
Add Optional Attribute	Improve the expressiveness of an Element by adding an optional Attribute to it.	Yes
Add Essential Attribute	Improve the expressiveness of an Element by adding an essential Attribute to it.	Maybe [1]
Add Optional Relationship	Improve the expressiveness of the DSL by creating a new optional Relationship.	Yes
Add Essential Relationship	Improve the expressiveness of the DSL by creating a new essential Relationship.	Maybe [1]
Change Attribute to Element	Where attribute needs sub-Elements, Attributes or a multiplicity greater than 1.	Yes
Change Element to Attribute	Where an Element can more simply be described as an Attribute of another Element.	Yes[2]
Transform Data Type of Attribute	More accurately describe the data type of an attribute.	Maybe [3]
Make Attribute Optional	Stop requiring an Attribute (often by adding an intelligent default), speeding development of statements within the DSL.	Yes
Make Attribute Required	Require a value for an Attribute.	Maybe [1]
Limit Relationship to has-one	Stop supporting a multiplicity greater than 1.	Maybe [4]
Allow Relationship to have-many	Move a Relationship from supporting has-one to has-many.	Yes
Deprecate Element	Advise that the Element should not be used and may be deleted in a future version.	Yes
Deprecate Attribute	Advise that the Attribute should not be used and may be deleted in a future version.	Yes
Deprecate Relationship	Advise that the Relationship should not be used and may be deleted in a future version.	Yes
Delete Element	Remove an Element from a DSL to remove unnecessary or inappropriate concept.	Maybe [5]
Delete Attribute	Remove an Attribute from a DSL to remove unnecessary or inappropriate Attribute.	Maybe [5]
Delete Relationship	Remove a Relationship from a DSL to remove unnecessary or inappropriate Relationship.	Maybe [5]

[1] Only if a universal default value can be set for all existing statements or if a calculation can be described for setting an initial value for all existing statements.
[2] This transformation cannot be performed on Elements that have Attributes or Relationships.
[3] The field of automatically transforming between data types is well explored. Obviously for some transformations and values this will be possible and for others data loss would ensue.
[4] Only if there are no instances of the Relationship with more than one related instance.
[5] Only if there are no such Items, otherwise data loss would ensue.