Structured Co-Evolution of Models and Web Application Platforms

Adam Pingel
University of California, Los Angeles
pingel@cs.ucla.edu

Abstract

Web applications exemplify the need for generative programming techniques in part due to the many languages, artifacts, and groups of developers involved. Some problems remain, including those that that arise from the interplay with versioning. This paper proposes addressing these problems with structured program transformations, and explores a framework for the co-evolution of platform artifacts and the models that generate them.

Categories and Subject Descriptors D.2.0 [*Software Engineering*]: General

1. Introduction

Web application platforms are plagued by *scaffolding* – artifacts in many different languages which must be synchronized. During application evolution, these scattered dependencies provide abundant opportunities for amplifying human error. The simplest example of such a dependency is the name of JSP files, which are referenced in numerous XML descriptor files, in addition to the filesystem's reference. Figure 1 illustrates the variety of artifacts and their shared references.

A common method of dealing with the problems posed by scaffolding is generating source code from *domain specific languages* (*DSLs*) designed to succinctly model the behavior of the system. One variant of code generation from a DSL is known as *model-driven software development* (*MDSD*).

For a given instance of the DSL, D, and a set of platform artifacts $P_1 \ldots P_N$, the generation of artifacts from D can be thought of as a function called "project":

$$< P_1 \ldots P_N >= project(D)$$

This solves many of the problems posed by scaffolding, but it leaves others.

Copyright is held by the author/owner(s).
OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-786-5/07/0010.

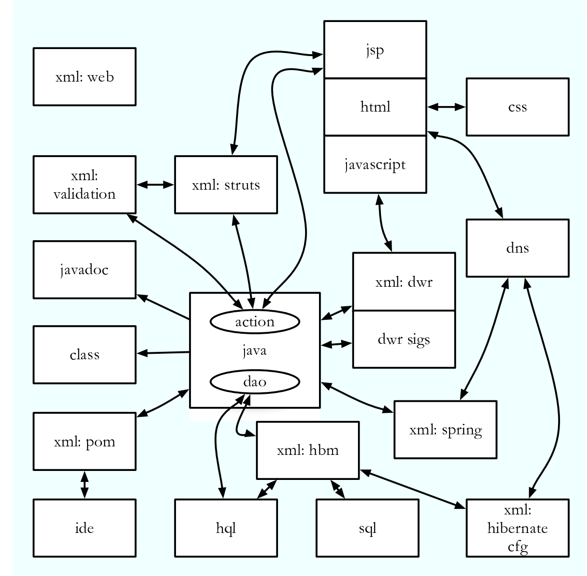

Figure 1. References Among Platform Artifacts $P_1 \ldots P_N$

2. Research Problem

2.1 Model-Driven Co-Evolution

The use of structured refactoring has been shown by MolhadoRef [4] to aid program merging, history preservation, and understanding of evolution. Languages for defining refactorings in other languages such as JunGL [6] are a promising way to implement efficient, correct program transformations. They demonstrate the usefulness of authoring refactorings and other common evolutionary steps as *endogenous* (within the same language) transformations apart from any particular usage. The same issues arise when refactoring the DSL.

Platform artifacts which are always generated from the DSL will escape these problems. In general, however, some of the platform artifacts will contain non-derived information. In those cases, structured transformations offer the same benefits that they do to any versioned artifact.

For a well-defined endogenous transformation, τ, in the DSL, the normal workflow after creating the initial $P_1 \ldots P_N$ is:

$$D' = \tau(D)$$

$$< P'_1 \ldots P'_N >= project(D')$$

To address the need for structured transformations, a function $evolve$ computes a set of transformations for the platform artifacts such that $evolve(\tau, D)$ applied to the platform artifacts is the same as $project(\tau(D))$.

$$D' = \tau(D)$$

$$< \tau_1 \ldots \tau_N >= evolve(\tau, D)$$

$$P'_i = \tau_i(P_i) \forall \tau_i$$

2.2 Platform-Driven Co-Evolution

Another difficulty of MDSD is support for editing artifacts downstream from the DSL. Object/Relational Mapping (ORM) tools such as Hibernate demonstrate the importance of propagating changes from either side of the map. This ability is often referred to as *round tripping* in ORMs. Antkiewicz and Czarnecki [2] discuss the importance of round-trip engineering as it applies to complex Eclipse API's.

JSP pages are examples of artifacts that include both generated code such as forms as well as developer-written code. The generated parts are placed within *protected regions*, making them either read-only or modifiable in a limited way. We propose that they be available for participation in transformations that are bound to DSL-level transformations. The function $coevolve$ computes the corresponding DSL-level transformation for platform-level transformations.

For a developer-initiated transformation, τ_i on artifact P_i:

$$\tau = coevolve(\tau_i, P_i)$$

$$< \tau_1 \ldots \tau_N >= evolve(\tau, D)$$

$$D' = \tau(D)$$

$$P'_i = \tau_i(P_i) \forall \tau_i$$

3. Approach

The initial steps of identifying the platform technologies and building a sample DSL are complete. The technologies chosen for the example platform are a good basis for creating richly interactive, robust web applications. The server-side technologies include Java 5, JSP, Struts 2, Tomcat 6, MySQL 5, and Hibernate 3 as an ORM. Direct Web Remoting (DWR) 2 makes Java methods available to browser-side JavaScript. Browser-side technologies include AJAX support in JSP, DWR 2, and the cross-browser JavaScript library Dojo 0.9. A parser written in ANTLR 3 creates Java model from a toy invoice processing language. Templatized versions of the platform artifacts are instantiated, resulting in a directory structure which is ready to build and deploy with Maven 2.

Helsen [7] identifies a set of requirements for model transformation languages and discusses some challenges, especially those of bidirectional model-to-model languages. Czarnecki and Helsen [3] catalogue and classify several transformation languages, including OMG's QVT at the ATLAS group's ATL [1]. Jouault and Kurtev [5] discuss the differences between the two.

Initial development of the artifact-level and DSL-level program transformations managed by the $evolve$ and $coevolve$ functions is with the ATL suite of tools. Two transformations of low complexity are implemented first: Rename Variable, and Add Variable. These transformations are simple enough to be present intact in a large subset of the platform languages. A third more complex transformation, Replace Association By Attribute, applies to the DSL and several platform artifacts.

4. Goals and Evaluation

The primary goals are the feasibility and applicability of structured co-evolution as framed by the $evolve$ and $coevolve$ functions. The applicability of the approach is relative to the quality of the platform, the quality of the DSL, and the extent to which the provided solutions offer benefits under realistic change scenarios. The highest priorities are expansion of the DSL to exercise more of the platform, and the implementation of more transformations.

References

[1] www.eclipse.org/m2m/atl.

[2] M. Antkiewicz. Round-trip engineering of framework-based software using framework-specific modeling languages. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 323–326, Washington, DC, USA, 2006. IEEE Computer Society.

[3] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[4] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 427–436, Washington, DC, USA, 2007. IEEE Computer Society.

[5] F. Jouault and I. Kurtev. On the architectural alignment of atl and qvt. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1188–1195, New York, NY, USA, 2006. ACM Press.

[6] M. Verbaere, R. Ettinger, and O. de Moor. Jungl: a scripting language for refactoring. In D. Rombach and M. L. Soffa, editors, *ICSE'06: Proceedings of the 28th International Conference on Software Engineering*, pages 172–181, New York, NY, USA, 2006. ACM Press.

[7] M. Voelter, T. Stahl, J. Bettin, A. Haase, and S. Helsen. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, July 2006.