Grammar-Driven Generation of Domain-Specific Language Tools

Hui Wu
Department of Computer and Information Science
University of Alabama at Birmingham
Birmingham, AL, USA 35294-1170
+1 205 934 2213
wuh@cis.uab.edu

Abstract

Domain-specific languages (DSLs) assist an end-user programmer in writing programs using idioms that are closer to the abstractions found in a specific problem domain. Language testing tool support for DSLs is lacking when compared to the capabilities provided in standard general purpose languages (e.g., Java and C++). For example, support for debugging a program written in a DSL is often nonexistent. This research abstract describes a grammar-driven technique to build a testing tool generation framework through automated transformation of existing DSL grammars. The modified grammars generate the hooks needed to interface with a supporting infrastructure written for an Integrated Development Environment that assists in debugging, testing, and profiling a DSL program.

Categories and Subject Descriptors D.3.1 [**Programming Languages**]: Formal Definitions and Theory – *syntax, semantics*, D.2.5 [**Software Engineering**]: Testing and Debugging – *debugging aids, testing tools*.

General Terms Experimentation, Languages, Verification.

Keywords Debugging, DSLs, mapping, testing, grammar.

1. Problem Statement

A DSL is a "programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain" [2]. DSLs assist in the creation of programs that are often more concise than an equivalent program written in a general purpose language (GPL), such as Java or C++. A key benefit of using a DSL is the isolation of accidental complexities typically required in the implementation phase (i.e., the solution space) such that an end-user programmer can focus on the key abstractions of the problem space. Examples of DSLs include parser generators like YACC. DSLs also have been created for financial, engineering, and medical domains.

An extensive summary of the current practice of DSL implementation patterns is provided in [5]. A popular approach for implementing DSLs is to create a code generator that translates the DSL source code into a GPL. However, the translation has a serious disadvantage when it comes to the issue of debugging and testing because it requires knowledge of both the problem domain and the target GPL. This results in a mismatch of abstraction levels because the end-user programmer must understand the translated code in the GPL, rather than the higher-level domain intention contained in the DSL [3]. Thus, an end-user may not debug and test their programs with higher-level domain-specific concepts and notations, but instead in terms of GPL concepts (e.g., test case reporting and debugging are represented as generated GPL code).

Ideally, an end-user programmer should be able to debug and test their programs at the DSL abstraction level rather than the translated GPL code. This research abstract describes an approach for generating the associated language tools (e.g., debugger, testing engine, and profiler) from grammar specifications of the DSL. This approach preserves the primary benefit of using a DSL by permitting debugging, testing, and profiling using domain concepts. Manual construction of the testing tools for each new DSL can be time-consuming, expensive, and error-prone. For example, a debugger is difficult to build because it depends heavily on the underlying operating system's capabilities and lower-level native code functionality [6]. It is hypothesized that an approach to generate testing tools automatically from the DSL grammar specification preserves all the advantages of using a DSL and reduces the implementation costs of DSL tools.

The mapping information from the DSL to its generated GPL is crucial. To define the mapping, additional semantic actions inside grammar productions are needed. A crosscutting concern emerges from the addition of the explicit mapping in grammar productions. The manual addition of the same code in grammar productions results in much redundancy that can be better modularized using an aspect-oriented approach applied to grammars [8].

2. Research Hypothesis

This research has adopted a code generator approach whereby the DSL is converted to a GPL. Instead of building new debuggers, testing engines, and profilers from scratch, the mapping information is generated as hooks into a plug-in for Eclipse, and existing GPL language tools are reused (e.g., Java debugger, JUnit, and JFluid). The result is a special-purpose IDE for a DSL that targets a narrow domain, providing high levels of specificity and automation at relatively low development cost.

There are many different categories of DSLs. Currently, our research has concentrated on three types of DSLs (e.g., imperative DSL, declarative DSL, and hybrid DSL). An imperative DSL is focused on assignment expressions and control flow statements. A declarative DSL is based on declarations that state the relationship between inputs and outputs. A hybrid DSL is embedded within GPL code (e.g., YACC and CUP), or vice versa.

3. Proposed Solution

We have initiated an early phase of this effort to assess the feasibility of the proposed approach toward generating debuggers for DSLs (called the DSL Debugger Framework – DDF). The initial work investigated the generation of a debugger for a small imperative DSL language for controlling robots, and also a

OOPSLA'06 October 22-26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

declarative DSL for specifying feature models. The grammars for these DSLs already existed and were specified in ANTLR [1]. The DDF provides a practical technique to reuse the existing debugging support in Eclipse [7] and the Java debugger. The overview of the DDF gives the central idea of the proposed research. A description of the initial results, including video demonstrations and papers, can be found at http://www.cis.uab.edu/wuh/DDF.

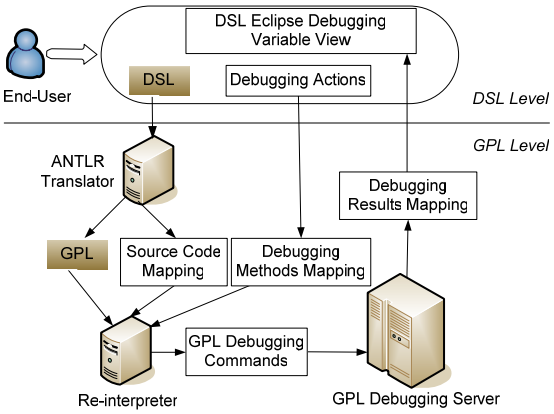

Figure 1: DSL Debugger Framework (DDF)

An illustrative overview of the DDF is shown in Figure 1. The Mapping Generator components comprise the source code mapping process and the debugging methods mapping process (middle of figure). The results from these two mapping processes are reinterpreted into the GPL debugging server commands against the translated GPL code. The ANTLR translator generates GPL code and mapping information from the DSL source. The source code mapping process uses the generated mapping information to determine which line of the DSL code is mapped to the corresponding segment of GPL code. It indicates the location of the GPL code segment. The debugging method mapping process takes the user's debugging actions from the debugger perspective at the DSL level to determine what type of debugging commands need to be issued to a debugger server at the GPL level.

The GPL debugger server responds to the debugger commands sent from the re-interpreter. The result from issuing the debug command at the GPL level is sent back to a debugging results mapping component. Because the messages from the GPL debugger are command line outputs in GPL notation, which know nothing of the DSL or the Eclipse debug platform, it is necessary to remap the results back into the DDF. The debugging results mapping component maps the messages back to the DSL level through the wrapper interface. The domain expert only interacts directly with the debugging perspective at the DSL level. The construction of the framework to generate a DSL testing engine and profiler is similar to the general process described in Figure 1.

4. Expected Contributions

The following summarizes the key contributions of scientific merit and broad impact of the doctoral research presented in this extended abstract: 1) Investigation into the techniques to debug, test, and profile DSLs. 2) Provide a DSL language tool generation framework that generates tools (e.g., debugger, testing engine, or profiler) automatically from DSL grammar specifications. For different DSLs, slight modification of certain components of the framework may be considered necessary, but the architecture of the framework is generic. 3) Exploration of the technique for better separation of concerns in grammars to support DSL testing tools using aspects; this represents an application of Grammarware [4].

5. Concluding Remarks

This research presents an approach that generates the testing tools (e.g., debugger, test engine, and profiler) needed to use a DSL from a language specification captured in a grammar. An additional objective of the research concerns the topic of weaving aspects into grammars during the tool generation process. A key enabler of the proposed research is the application of AOP to support a new generative approach for language tool construction by weaving tool concerns into grammars. Successful completion of the proposed research would advance the capabilities of domain experts and end-user programmers by providing an adequate DSL testing tool suite. Various experimental validation efforts will be conducted to test this framework's applicability, scalability, and reliability. In order to provide such assessment, DSLs obtained from industry and research collaborators will help to drive the evaluation process, such as hybrid DSLs when the DSL is embedded within a host GPL (e.g., the SWUL language for creating Java Swing user interfaces). The ability of the generated debugger and test engine to detect errors in the DSL is also an assessment criterion. Complex grammars will serve as test cases to determine the benefits of grammar weaving.

Acknowledgement

This work was supported by an IBM Eclipse Innovation Grant.

References

[1] ANTLR - ANother Tool for Language Recognition, available at http://www.antlr.org/

[2] A. van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," *ACM SIGPLAN Notices*, vol. 35, no. 6, pp. 26-36, June 2000.

[3] R. E. Faith, *Debugging Programs after Structure-Changing Transformation*, Ph.D. Dissertation, Dept. of Computer Science, Univ. of North Carolina at Chapel Hill, available at http://www.cs.unc.edu/~faith/faith-dissertation-1997.ps, 1998.

[4] P. Klint, R. Lammel, and C. Verhoef, "Towards an Engineering Discipline for Grammarware," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 3, pp. 331-380, July 2005.

[5] M. Mernik, J. Heering, and A. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316-344, December 2005.

[6] J. B. Rosenberg, *How Debuggers Work-Algorithms, Data Structures, and Architecture*, John Wiley and Sons, 1996.

[7] D. Wright and B. Freeman-Benson, "How to Write an Eclipse Debugger," *Eclipse Corner*, available at http://www.eclipse.org/articles/index.html, Fall 2004.

[8] H. Wu, J. Gray, S. Roychoudhury, and M. Mernik, "Weaving a Debugging Aspect into Domain-Specific Language Grammars," *Symposium for Applied Computing (SAC) – Programming for Separation of Concerns Track*, Santa Fe, NM, pp. 1370-1374, March 2005.