Weaving a Debugging Aspect into Domain-Specific Language Grammars

Hui Wu, Jeff Gray, and Suman Roychoudhury
Department of Computer and Information Sciences
The University of Alabama at Birmingham
Birmingham, AL, USA 35294-1170
Phone: 01-205-934-2213
{wuh, gray, roychous}@cis.uab.edu

Marjan Mernik
Faculty of Electrical Engineering and Computer Science
University of Maribor
2000 Maribor, Slovenia
Phone: 386-2-220-7455
marjan.mernik@uni-mb.si

ABSTRACT
A common trend in programming language specification is to generate various tools (e.g., compiler, editor, profiler, and debugger) from a grammar. In such a generative approach, it is desirable to have the definition of a programming language be modularized according to specific concerns specified in the grammar. However, it is often the case that the corresponding properties of the generated tools are scattered and tangled across the language specification. In this paper, separation of concerns within a programming language specification is demonstrated by considering debugging support within a domain-specific language (DSL). The paper first describes the use of AspectJ to weave the debugging semantics into the code created by a parser generator. The paper outlines several situations when the use of AspectJ is infeasible at separating language specification properties. To accommodate such situations, a second approach is presented that weaves the debugging support directly into a grammar specification using a program transformation engine. A case study for a simple DSL is presented to highlight the benefits of weaving across language specifications defined by grammars.

Categories and Subject Descriptors
D.3.1 [**Programming Languages**]: Formal Definitions and Theory-*Syntax, Semantics.* D.2.6 [**Software Engineering**]: Program Environments-*Integrated environments*. F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems-*Parsing*.

Keywords
AOSD, debugging, DSLs, Grammarware

1. INTRODUCTION
A domain-specific language (DSL) is a programming language with concise syntax and rich semantics designed to solve problems in a particular domain. A DSL is usually smaller and easier to use than a general purpose language (GPL), such as Java or C+. Efforts to design, program, and maintain programs written in a DSL are often hindered by the lack of support for an integrated development environment (IDE), which unites an editor, compiler, and debugger in a common toolsuite. Manual construction of the IDE for each new DSL can be time-consuming, expensive, and error-prone. An approach to generate automatically the IDE as a whole, or in part, from the DSL grammar specification preserves all the advantages of using a DSL and reduces the implementation costs of DSL tools.

A source-level debugger is a critical tool to assist a programmer, at any level of abstraction, in discovering the location of a program fault. Most modern IDEs (e.g., Eclipse, JBuilder, and .Net) include detailed support for debugging. A debugger is difficult to build because it depends heavily on the underlying operating system's capabilities and lower-level native code functionality [4]. Although techniques for constructing a debugger for a GPL have been developed over the years, debug support for DSLs has not been investigated deeply.

The DSL Debugging Framework (DDF) [8] is a set of Eclipse plug-ins providing core support for DSL debugging. In the DDF, a language specification is written in ANTLR (ANother Tool for Language Recognition), which is a lexer and parser generator [2]. ANTLR can be used to construct recognizers, compilers, and translators from grammatical descriptions containing Java, C++, or C# actions. A DSL is usually translated into a GPL that can be compiled and executed. From a DSL grammar, the DDF generates GPL code representing the intention of the DSL program (i.e., the DSL is translated to a GPL and the GPL tools are used to generate an executable program). The DDF also generates the mapping information that integrates with the host GPL debugger (e.g., the stand alone command line Java debugger – jdb). The generated mapping code re-interprets the DSL program, and the debugger state, into a sequence of commands that query the GPL debugger server. The responses from the GPL debugger server are mapped back into the DSL debugger perspective. Thus, the end user performs debugging actions at the level of abstraction specified by the DSL, not at the lower-level abstraction provided by the GPL.

Using the DDF, a DSL debugger can be generated automatically from the DSL grammar provided that an explicit mapping is specified between the DSL and the translated GPL. To define this mapping, additional semantic actions inside each grammar production are defined. A crosscutting concern emerges from the

SAC'05, March 13-17, 2005, Santa Fe, New Mexico, USA.

addition of the explicit mapping in each of the grammar productions. The manual addition of the same code in each grammar production results in much redundancy that can be better modularized using an aspect-oriented approach applied to grammars. The primary contribution of the paper is a technique for better separation of concerns in Grammarware, which comprises grammars and all grammar-dependent software (e.g., lexer, parser, and compiler) [6].

The outline of the rest of the paper is as follows. In Section 2, an overview is presented of two different approaches for weaving debugging support into a language specification. A small case study is introduced in Section 3 to serve as an illustration of the two approaches. Related work and a conclusion represent the final sections of the paper.

2. GRAMMAR WEAVING APPROACHES

This section outlines two different approaches for weaving a debugging concern into a DSL programming environment. Section 3 provides more detailed examples of each approach applied to a simple DSL case study. Each approach assumes that an ANTLR grammar is used to specify the syntax and semantics of a DSL. ANTLR permits semantic action code written in a GPL to be attached to each grammar production.

The first approach to modularizing a debugging concern in a DSL assumes the existence of an aspect weaver for the generated GPL. For example, AspectJ is an aspect-oriented extension to Java that assists in modular implementation of numerous crosscutting concerns [1]. In Figure 1, ANTLR automatically generates the lexer and parser from the DSL grammar. Assuming the generated parser is in Java, AspectJ can be used to define a debugging aspect that weaves the debug mapping code to generate a new lexer and parser (*Lexer'* and *Parser'*). After the debug concern is weaved into the lexer and parser, DDF uses the transformed GPL and mapping code to generate the DSL debugger [8].

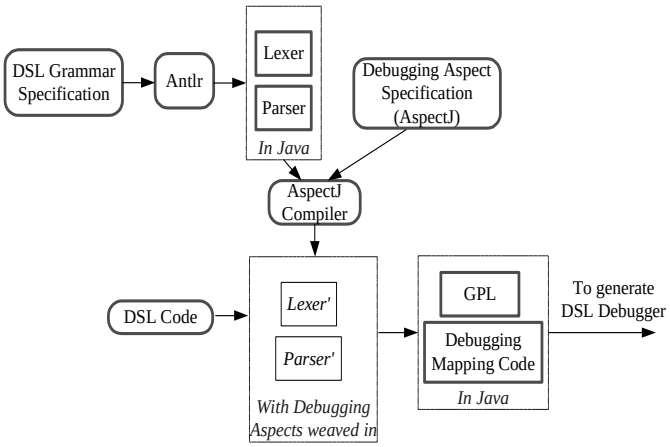

Figure 1. Post-ANTLR Processing (AspectJ Approach)

The lack of mature aspect weavers for many languages (e.g., Object Pascal, C, or Ada) is a serious disadvantage of the first approach. That is, the first approach requires an aspect weaver for the generated GPL as the mechanism for modularizing the debug concern. Another disadvantage of the first approach is that it requires the developer of the DSL to have detailed knowledge of the code generator within ANTLR in order to construct the appropriate pointcuts. In some cases, the translation is done by a legacy parser, which creates a difficulty because the generated parser code can be messy and generally unreadable by a human. One line in a DSL can translate into dozens of lines of GPL code.

In the second approach toward modularizing concerns in a grammar, the Design Maintenance System (DMS) [3] is used to weave the debugging concern directly into the *grammar* itself, rather than the generated GPL source. DMS is a program transformation engine and re-engineering toolkit developed by Semantic Designs (www.semdesigns.com). It facilitates the transformation of one program representation into a new representation and provides lower-level transformation functions such as parsing, abstract syntax tree (AST) generation/manipulation, pretty printing, powerful pattern matching, and source translation capabilities [3]. DMS provides pre-constructed domains for several dozen languages such as Java, C++, and Object Pascal. In addition to the available parsers, the underlying rewriting engine of DMS provides the machinery needed to perform invasive software transformations on legacy code [5]. For the requirements of this project, a DMS domain was created that is capable of parsing and transforming grammars specified in ANTLR.

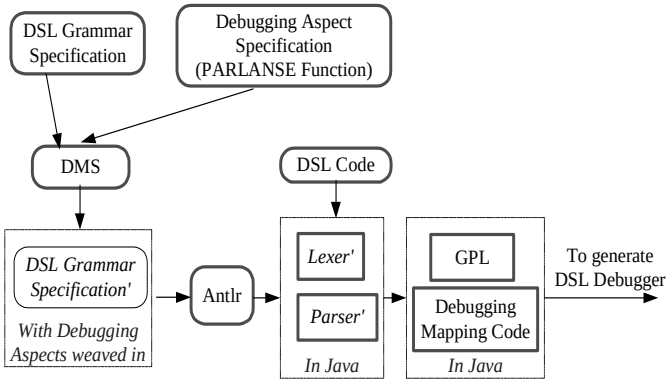

Figure 2. Pre-ANTLR Processing (DMS Approach)

In Figure 2, a debugging aspect is specified as a DMS function written in the PARLANSE language, which provides transformation functionality using pattern matching and rewrite specifications on the AST of a source program (in this case, the source is actually a grammar file). PARLANSE ("Parallel Language for Symbolic Expression") is a parallel programming language designed by Semantic Designs, intended to allow software engineers to develop programs that manipulate symbolic values in an efficient manner on conventional scalar multiprocessors inside DMS [3]. In Figure 2, before the grammar is even processed by ANTLR, it is first pre-processed by DMS in order to weave the debugging aspect into the original grammar productions. The transformed grammar is then submitted to ANTLR in order to generate the parser and lexer for a specific GPL. The key contribution of this approach is the transformation of the grammar itself. The specification of the debug mapping is modularized in a single place – the DMS transformation function.

The second approach has the side benefit of language independence. It does not matter which GPL serves as the generated target. The DMS ANTLR domain is capable of parsing the grammar and adding the needed debug transformations for a large set of programming languages. An example of each of the two approaches is provided in the next section, which introduces a simple DSL.

3. A CASE STUDY

This section presents a very simple DSL that will be used to illustrate the concept of an aspect to support debugging in DSLs. The Robot DSL consists of four commands that control robot movement: up, down, right, and down. Every command will increase or decrease the position of the robot along the x or y coordinates. As a side effect, each command will also increase the timer by one. Additional Robot DSL statements are: initial statement, set statement, and print statement. Figure 3 is sample code written in the Robot DSL - line 2 initialize the robot's beginning position as (0, 0); line 5 forces (5, 6) as the robot's new current position; line 8 prints the robot's current position.

```
1     begin
2       init Position(0,0)
3       left
4       down
5       set Position(5,6)
6       up
7       right
8       print Position
9     end
```

Figure 3. Robot DSL Sample Code

Figure 4 is part of the Robot DSL grammar specification that translates the Robot DSL to Java. From this grammar, ANTLR will generate a lexer and parser for the Robot DSL. Lines 10, 11, and 14 represent the syntax of the Robot DSL in BNF format. Lines 12 and 13 are semantic actions (in Java) for the "right" Robot command; lines 15 and 16 specify the semantic action for the "left" command.

```
…
10 command
11   :( RIGHT {
12        fileio.print("x=x+1;// move right");
13        fileio.print("time=time+1;");}
14      |LEFT {
15        fileio.print("x=x-1;// move left");
16        fileio.print("time=time+1;");}
…
```

Figure 4. Robot DSL Grammar Specification

3.1 Weaving at the Generated Code Level

The debug mapping for the DSL debugger was originally specified manually at the DSL grammar level (see Figure 6). For example, line 11 to line 18 represents the semantic action of the "right" command. Line 12 keeps track of the Robot DSL line number; line 14 records the first line of the translated GPL code segment; line 16 marks the last line of the translated GPL code segment; line 17 and line 18 generate the mapping code statement used by the DDF.

These semantic actions are repeated in *every* terminal production; the same mapping statements for the "left" command appear in lines 20, 22, and 24 to 26. Although the Robot DSL is simple (due to space limitations), it is not uncommon to have grammars with hundreds of production rules. In such cases, much redundancy will exist because the debug mapping code is replicated across each production. Of course, because the debug mapping concern is not properly modularized, changing any part of the debug mapping has a rippling effect across the entire grammar. An aspect-oriented approach can offer much benefit in such a case, even though the main concern emerges at the grammar level.

```
…
 6   after(int commandname):
 7       call(void antlr.Parser.match(int))
 8    && args(commandname)
 9                  { match(commandname); }
10   pointcut count_dsllinenumber():
11       call (void P.command());
12   after(): count_dsllinenumber(){
13            { dsllinenumber=dsllinenumber+1;}
…
```

Figure 5. Debugging Aspects in AspectJ Notation

An aspect for capturing the debug mapping (using AspectJ) is specified in Figure 5. The pointcut **`count_dsllinenumber`** is a command method called by class "P," which is a parser class that is automatically generated by ANTLR. This aspect executes **`dsllinenumber=dsllinenumber+1;`** after all calls to **`void P.command()`**, regardless of the specific command method returned. This aspect counts the DSL line number at the DSL source code level. Whenever there is a DSL command or statement, the counter will increase by one. During the design phase of the Robot grammar, the "begin" and "end" statements were not defined as commands, which force these two statements to be handled differently as specified in line 6 to line 9 of Figure 5. The method **`match(commandname)`** only picks up "begin" and "end" statements.

The aspect of Figure 5 handles the increment of the DSL line number that is weaved at the beginning of each production. Several other aspects are needed to specify the complete debug mapping. Although space does not permit all to be shown, another aspect is to locate the first and last line number of the translated segment of GPL code. This aspect is difficult to define using the AspectJ notation. After the weaving process is accomplished by AspectJ, the Parser of Figure 1 becomes *Parser'*, which not only translates the DSL to the GPL, but also generates the necessary mapping code needed by DDF for automatically generating the DSL debugger.

3.2 Weaving at the DSL Grammar Level

Although the Post-ANTLR processing approach using AspectJ can solve the crosscutting problems in the DSL grammar, this method is infeasible when an aspect weaver does not exist for the generated GPL. The results of the previous section were favorable because the generated code was Java, which allowed AspectJ to be used to do the post-ANTLR weaving. A different technique is needed when the parser generates a GPL that does not have an aspect weaver. As mentioned in Section 2, a program transformation system (e.g., DMS) can be used to weave crosscutting concerns into the actual grammar definition. After weaving the aspects into the grammar using DMS, the changes in

```
…
10 command
11  :( RIGHT {
12       dsllinenumber=dsllinenumber+1;
13       fileio.print(" x=x+1;// move right");
14       gplbeginline=fileio.getLinenumber();
15       fileio.print(" time=time+1;");
16       gplendline=fileio.getLinenumber();
17       filemap.print("mapping.add(newMap(" + dsllinenumber + ",\"Robot.java\"," +
18                     gplbeginline + "," + gplendline + "));");}
19     |LEFT {
20       dsllinenumber=dsllinenumber+1;
21       fileio.print(" x=x-1;// move left");
22       gplbeginline=fileio.getLinenumber();
23       fileio.print(" time=time+1;");
24       gplendline=fileio.getLinenumber();
25       filemap.print("mapping.add(newMap(" + dsllinenumber + ",\"Robot.java\"," +
26                     gplbeginline + "," + gplendline + "));");}
…
```

Figure 6. Robot DSL Grammar Specifications in ANTLR Notation

terms of aspects will automatically propagate into the generated parser through the grammar productions. Unlike the first approach described in Section 3.1, it is not necessary to weave into the generated parser because the debugging concern is weaved at an earlier stage in the grammar itself.

In Figure 6, the Robot DSL grammar contains an ANTLR specification of BNF syntax (e.g., line 10, 11, and 19). The semantic action is specified using Java by separating the action code with a pair of curly braces. Note that the Java domain is embedded within ANTLR, which makes it difficult to parse two different syntactic constructs (i.e., ANTLR and Java) using any one particular parser. A naïve solution would be to include all the tokens and productions from both domains to form a combined grammar and then generate the parser using the DMS parser generator. However, this approach does not make use of the existing DMS Java grammar/parser. A better approach would be to reuse the existing DMS Java tools and separate the ANTLR grammar productions from the Java grammar productions, but still parse the input source containing tokens from both languages. This requires a minor extension of the DMS ANTLR grammar. To parse the embedded semantic action (i.e., essentially Java code) within the ANTLR domain, a special string token called ANTLR_ACTION is used. The regular expression associated with this token is as follows:

```
#token ANTLR_ACTION [STRING] "{ (\\[{}\]|[\{}])* \}"
```

ANTLR_ACTION is a token that describes a string pattern beginning with a left curly brace, ending with a right curly brace, and containing any characters in between. Having specified each grammar production's semantic action as a single ANTLR_ACTION node, DMS can parse the ANTLR grammar specification (combined with Java semantic actions) to construct an AST for that grammar instance. Note that the semantic actions are stored as string expressions at the ANTLR_ACTION nodes of the syntax tree.

The next step involves retrieving the associated string expressions from the specific ANTLR_ACTION nodes and parsing them with the DMS Java parser. However, an inherent difficulty in using a regular Java parser is that the string expressions linked to an ANTLR_ACTION node are not complete Java programs, only fragments (i.e., statement blocks). Therefore, to avoid exceptions thrown by the predefined DMS Java parser, minor modifications are made to the root node (i.e., starting production in the Java grammar specification file) and the parser is regenerated to allow partial parsing. Because the approach specifically targets the translation from a DSL to a GPL, the semantic actions in an ANTLR grammar specification are primarily method call statements (with one string parameter, see Figure 4 line 12, 13, 15, and 16).

After the parse tree for the ANTLR_ACTION nodes are retrieved using the modified Java parser, new debugging aspects are weaved using the ASTInterface API provided by DMS. The API provides methods for modifying a given syntax tree to regenerate a new tree structure. The steps describing the above process are shown in Figure 7. Due to lack of space, the aspect source code and the complete PARLANSE source code presenting these steps is not shown here, but is available at the project website (http://www.cis.uab.edu/wuh/DDF).

1. Specify ANTLR grammar specification
2. Specify Java semantic actions using DMS regular expression
3. Generate ANTLR Parser
4. Generate abstract syntax tree with ANTLR_ACTION nodes
5. Search ANTLR_ACTION nodes from the generated AST
6. Retrieve ANTLR_ACTION nodes and store them in a hash map
7. Retrieve associated string expression from each ANTLR_ACTION node
8. Modify the regular Java parser by changing the starting production
9. Parse the associated string expressions as regular Java statement lists
10. Transform the statement lists using the ASTInterface API
11. Regenerate the ANTLR_ACTION nodes with debugging aspects weaved in
12. Output the complete ANTLR AST (with modified action nodes)

Figure 7. Steps to Weave Debugging Aspects into a Grammar

4. RELATED WORK

There is little related work in the area of aspect weaving at the grammar level. This section provides a brief overview of known work in the area. The Aspect-Oriented Compiler, proposed by Oege de Moor et al., is a technique for making compiler 'aspects' first-class objects that can be stored, manipulated and combined. The examples demonstrate a weaving process that is purely name-

based and not dependent on sophisticated program analyses. They parameterize the name analysis and type analysis modules and applied them to different attribute grammars within a functional language framework [9].

The Language Implementation System on Attribute Grammars (LISA) tool is a grammar-based system to generate a compiler, interpreter, and other language-based tools (e.g., finite state automata visualization editor) [7]. Using templates, LISA is able to describe the semantic rules that are independent of grammar production rules. LISA achieves better modularization than ANTLR by templates and inheritance formalism.

JastAdd is a Java based compiler construction system for AST transformation using JavaCC and tree-building using JJTree. The behaviors of a compiler (e.g., name analysis, type checking, code generation, and unparsing) can be modularized into different aspects in JastAdd which then weaves the behaviors together into classes using AOP techniques, providing a safer and more powerful way to construct a compiler system [10].

5. CONCLUSION

A DSL offers end users a notation for specifying the intent of a software system using idioms appropriate to the domain of interest. This paper presented an approach that generates the tools needed (e.g., editor, compiler, and debugger) to use a DSL from a language specification captured in a grammar. Specifically, the paper focused on issues regarding the topic of debugging support for a DSL development environment.

The difficulty of manual implementation of a DSL debugger as part of an IDE led to the idea of generating the debugger from a language specification. Yet, the decomposition of a language specification along the dimension of grammar productions forces some concerns to be scattered and tangled within the grammar. The specific contribution of this paper is the illustrated modularization of the debugging concern within the DSL grammar using AOP principles. The paper presented two approaches for weaving the debugger concern in conjunction with the DDF plug-in.

The first approach may be applicable in those cases when an aspect weaver is available for the generated GPL (i.e., AspectJ can be used when the GPL is Java). However, weaving into the generated GPL requires detailed knowledge of the parser generator such that appropriate pointcuts can be identified in the generated source. In those situations where an aspect weaver is not readily available for the generated GPL, the DMS approach for transforming the representative grammar is more suitable. The DMS transformation has more accidental complexities in terms of implementation, but does not require detailed knowledge of the GPL code generator. The effort required to adopt the DMS approach can be reduced when the transformation library of debugging aspects is further refined. The debugging aspect semantics is tied to a specific underlying GPL, but the weaving mechanism can be reused.

With respect to ongoing and future work, the grammar transformation approach using DMS is being applied to larger grammars obtained from several research projects. Because writing the PARLANSE functions to handle low-level program transformation is also tedious, we are developing a domain-specific aspect language for weaving the aspects into grammars to generate the PARLANSE functions. A long-term goal of the project is to investigate and construct a Domain-Specific Language Unit Test Framework (DUTF) to complement the DDF. The purpose of the DUTF will be to assist in the construction of test cases for DSL programs, much in the sense that JUnit is used to assist in automated unit testing of Java programs [11]. Failed test cases reported within the DUTF will reveal the presence of a program fault, and the DDF can then be used to identify the fault location. It is our intuition that the DUTF will represent another crosscutting concern that will need to be modularized within the DSL language specification as a grammar aspect.

6. REFERENCES

[1] The AspectJ web site (http://aspectj.org).

[2] *ANTLR - ANother Tool for Language Recognition*, available from http://www.antlr.org/.

[3] Ira Baxter, Christopher Pidgeon, and Michael Mehlich, "DMS: Program Transformation for Practical Scalable Software Evolution," *International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, May 2004, pp. 625-634.

[4] Jonathan B. Rosenberg, *How Debuggers Work- Algorithms, Data Structures, and Architecture*, John Wiley & Sons. Inc, New York, NY, 1996.

[5] Uwe Aßmann, *Invasive Software Composition*, Springer-Verlag, 2003.

[6] Paul Klint, Ralf Lammel, and Chris Verhoef, "Towards an Engineering Discipline for Grammarware," http://www.cs.vu.nl/grammarware/.

[7] Marjan Mernik, Matej Crepinsek, Tomaz Kosar, Damijan Rebernak, and Viljem Zumer, "Grammar-Based Systems: Definition and Examples," *Journal of Informatica*, accepted for publication - 2004.

[8] Hui Wu, Jeff Gray, and Marjan Mernik, "Debugging Domain-Specific Languages in Eclipse," *OOPSLA Eclipse Technology Exchange Poster Session*, Vancouver, BC, October 2004.

[9] Oege de Moor, Simon Peyton-Jones, and Eric Van Wyk, "Aspect-Oriented Compilers," *Generative and Component-Based Software Engineering*, Springer-Verlag LNCS 1799, September 1999, pp. 121-133.

[10] Görel Hedin and Eva Magnusson, "JastAdd-an Aspect-Oriented Compiler Construction System," *Science of Computer Programming*, April 2003, pp. 37-58.

[11] The JUnit web site (http://www.junit.org).