Debugging Domain-Specific Languages in Eclipse

Hui Wu and Jeff Gray
Dept. of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294-1170, USA
{wuh, gray} @ cis.uab.edu

Marjan Mernik
Faculty of Electrical Eng. and Computer Science
University of Maribor
2000 Maribor, Slovenia
marjan.mernik @ uni-mb.si

Abstract

Domain-specific languages (DSLs) assist a software developer (or even an end-user) in writing a program using idioms that are closer to the abstractions found in a specific problem domain. Tool support for DSLs is lacking, however, when compared to the capabilities provided for standard general purpose languages (e.g., support for debugging a program written in a DSL is often non-existent). This paper describes a mapping technique for augmenting existing DSL grammars to generate the hooks needed to interface with a supporting infrastructure written for Eclipse that assists in debugging a program written in a DSL.

1 Introduction

A Domain-Specific Language (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" [7]. DSLs assist in the creation of programs that are often more concise than an equivalent program written in a traditional programming language. 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 a programmer can focus on the key abstractions of the problem space.

An extensive summary of the current practice of DSL implementation is provided in [5]. A common approach for implementing DSLs is to create a pre-processor that translates the DSL source into a general purpose language (GPL), such as Java or C++. A benefit of the preprocessor approach is the potential for reuse of the host GPL development infrastructure to generate executable code. However, preprocessing has a serious disadvantage when it comes to the issue of debugging.

Debugging a DSL program using a translator is difficult because it requires knowledge of both the domain and the target GPL. This results in a mismatch of abstraction levels because the programmer must understand the translated code in the GPL, rather than the higher-level description contained in the DSL [2]. Thus, an end user is not debugging with higher level domain-specific concepts and notations, but instead in terms of GPL concepts (e.g., error reporting and debugging are at the level of the generated GPL code).

Ideally, a domain expert should be able to debug at the DSL abstraction level rather than the translated GPL code. This paper advocates an approach (generalized in [3]) for generating the translator and associated debugger from grammar specifications of the DSL. This approach preserves the primary benefit of using a DSL by permitting debugging using domain concepts.

To assist in DSL development, the Eclipse SDK provides the debug platform, which is a framework for building and integrating debuggers. It defines a set of Java interfaces that model common debugging artifacts (e.g., threads, variables, and breakpoints) and actions (e.g., stepping, suspending, resuming, and terminating). Although the platform does not provide a specific implementation of a debugger, it does provide a basic debugger user interface that can be adapted and extended with features specific to a particular language [8]. The remainder of this paper describes a generative approach for constructing DSL programming environments (including a translator to Java and DSL debugger) in Eclipse.

If accepted, ACM Copyright notice will go here

2 DSL Debugger Framework

In our approach, ANTLR (ANother Tool for Language Recognition) is used to specify the grammar of a DSL. ANTLR is a language tool for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C++, or C# actions [1]. From an ANTLR specification, a DSL translator can be generated. A pre-existing ANLTR plug-in for Eclipse is available (http://antlreclipse.sourceforge.net/).

The DSL Debugger Framework (DDF) supports automated debugger generation from an ANTLR specification of a DSL. The DDF provides a practical technique to reuse the existing debugging support in Eclipse. This framework has two components: the Mapping Generator and the Re-Mapping Generator. An overview of DDF is given in this section, with further details in Section 4.

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 debugger 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 debug methods mapping process takes the user's debugging commands from the debugger perspective at the DSL level to determine what type of debugging commands need to be issued to a command line debugger 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 Re-Mapping component. Because the messages from the GPL debugger are command line outputs, which know nothing of the DSL or the Eclipse debug platform, it is necessary to remap the results back into the DDF. The Re-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. Even if the domain expert has knowledge about the underlying GPL, one line of DSL code may be translated into dozens of lines of GPL code (which makes it even more difficult for domain expert to deal with the GPL code).

In the next section, we introduce a small DSL that serves as a case study. Source code mappings are described in Section 3.1, and debug method mapping details are contained in Section 5.

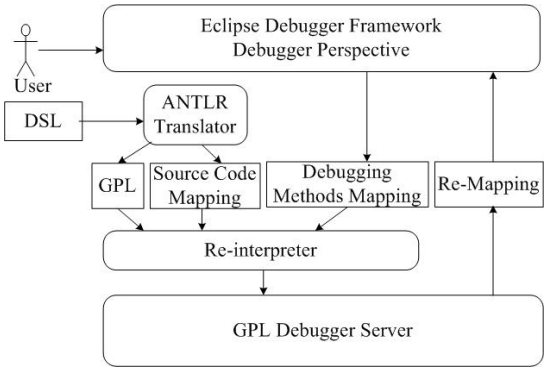

Figure 1: DSL Debugger Framework

3 Case Study: Robot DSL

Before describing the details of the DDF, this section presents a very simple DSL that will be used to illustrate the concept of debugging with a DSL. The Robot DSL consists of four commands that control the 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 2 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 2: Robot DSL Sample Code

```
...
10 command
11   :(RIGHT {
12      dsllinenumber=dsllinenumber+1;
        ...<omitted lines here that perform changes to x, y, timer>
15      filemap.print("  mapping.add(new Map(" + dsllinenumber +",\"Robot.java\","
16      gplbeginline + "," + gplendline + "));");}
17    |LEFT  {
18      dsllinenumber=dsllinenumber+1;
...
```

Figure 3: Robot DSL Specifications in ANTLR

3.1 Source Code Mapping

A Robot DSL source file is translated into an equivalent Java representation. During the translation process, the ANTLR grammar is augmented with additional semantic actions that generate the mapping files needed for the DSL debugger.

Figure 3 represents a fragment of the Robot DSL grammar in ANTLR. Line 11 indicates the start of the grammar production to process a "right" command, with lines 12 through 16 providing the semantic actions needed to execute the intention of "right" in Java. Lines 12, 15, and 16 represent the debug mapping information that contains the line number of the right command (variable **`dsllinenumber`** in line 15) in the Robot DSL. The mapping contains the following information:

1) the DSL line number (line 15),
2) the translated Java file name (line 15),
3) the line number of the first line of the corresponding code segment in Robot.java (variable **`gplbeginline`** on line 16),
4) the line number of the last line of the corresponding code segment in Robot.java (variable **`gplendline`** on line 16).

4 Debugger Implementation

When a debug session is invoked on a Robot program, the DDF launches a jdb session on the underlying translated Java code. jdb is a simple command-line debugger for Java classes [4]. It is a demonstration of the Java Platform Debugger Architecture that provides inspection and debugging of a local or remote Java Virtual Machine. The key implementation challenge concerned the generation of the Robot language debugger from a grammar, and how to map Java variable information back to the DSL debug variable view.

The DDF uses jdb as a debugging server to provide all of the required debug information, which is then passed on to the Eclipse debug perspective. The services that jdb supplies include starting a debug session, terminating a debug session, suspending at certain points of the execution, resuming execution, stepping over, setting or clearing a breakpoint, and retrieving the value of a specific variable. Every possible action from the Eclipse debug perspective will be reinterpreted by invoking the necessary sequence of jdb debugging commands.

The jdb is used to simulate the Robot DSL's debugger behavior instead of implementing the DSL debugger from scratch. This offers great potential for reuse for each new DSL environment that is created. Almost every GPL has its own version of a command-line debugger [6], so our technique is not bound to a particular GPL.

When a program is in debug mode, Eclipse triggers a DebugEvent for each event that occurs as a program is being debugged [8]. The DDF listens to all relevant debug events and updates the views accordingly. The entire Robot DSL debugger perspective is shown in Figure 4. The views included in this perspective are the following:

- **Variables View:** During a debug session, the variables declared in the Robot program will appear in the variables view. The values of each variable will be updated whenever there is an action performed in the debugging perspective.
- **Editor:** During debug mode, the text editor will show the breakpoints defined on specific lines of code (a blue dot will appear on the left side of text editor border). During a debugging session, a blue arrow (program pointer) will appear also on the left side of text editor border to indicate the current execution line. Along with the stepping action performed by user, the arrow will move accordingly.
- **Debugger View:** The tree structure reveals information about the current debug session (e.g., the name of the program being debugged, current threads, and underlying processes).
- **Breakpoints View:** Information about the editor position of all valid breakpoints.

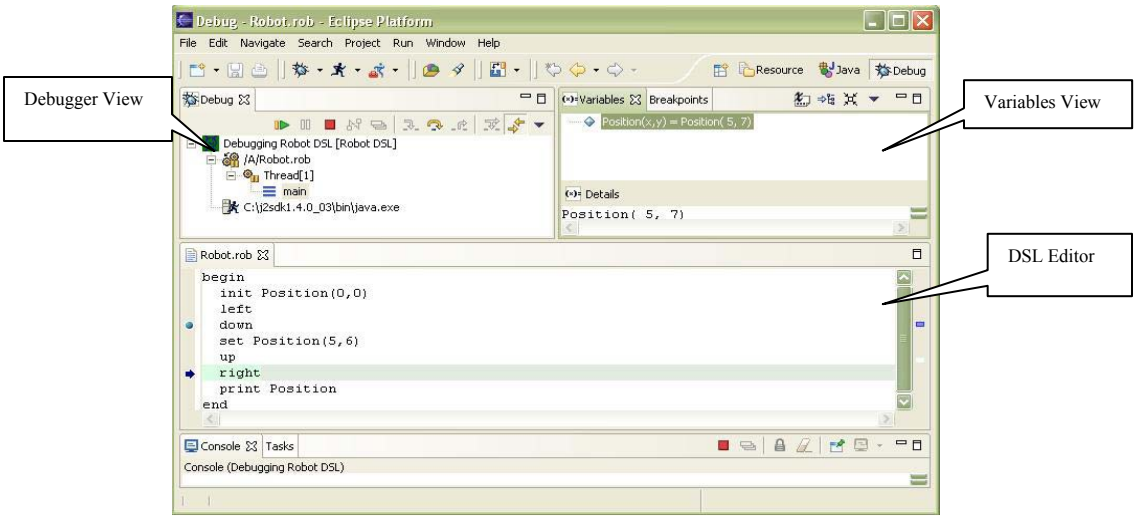

Figure 4: DSL Debugger Perspective in Eclipse

5 Debugging Mapping Methods

The most essential parts of DDF are the mappings between the DSL to the underlying GPL. In addition to the DSL-to-GPL mapping to capture the functional behavior of the DSL program, other mappings are required to specify the interactions with the associated GPL command-line debugger. Each of these mappings is specified as an ANTLR semantic action. The specifics of the required mappings are illustrated in Figure 5, with the type of mapping named in the first column, the DSL code in column 2, and the respective GPL mapping in column 3. Although the examples presented in the section are tied to the simple Robot language, the DDF mapping and interaction with the jdb and debug platform are actually independent of any specific DSL. The remainder of this section discusses the details of the debugging mapping types.

Mapping	Robot.rob (DSL)	Robot.java (GPL)
Code	… 3 **left** 4 **down** …	… 11 **//left move** 12 **x = x - 1;** 13 **time=time+1;** 14 15 **//down move** 16 **y = y - 1;** 17 **time=time+1;** …
Breakpoint	Set breakpoint at line 3	stop at Robot: 12 (jdb command)
Step Over	Step over line 3	stop at Robot: 16 cont (jdb commands)

Figure 5: DSL Debugging Mapping Methods

5.1 Breakpoints

To set a breakpoint in the jdb, a string command is issued that contains the source file and line number. For example, the command, "**`stop at MyClass: 22`**" would set a breakpoint at the first instruction for line 22 of the source file called MyClass.java [4]. Whenever a user declares a breakpoint in a DSL program, the corresponding line number in the GPL must be obtained from the mapping and passed along to jdb. In Figure 5, the 3rd row is an example of a breakpoint mapping from the Robot DSL to Java. If a breakpoint were set at line 3 of Robot.rob, the command generated to jdb would be "**`stop at Robot: 12.`**" The same principle applies to removing a breakpoint. Removing an existing breakpoint in Robot.rob at line 3 would correspond the jdb command of "**`clear Robot: 12`**." The DDF will intercept all events associated with the Eclipse debug platform, perform the associate mapping, and marshal the request and response from the jdb.

5.2 Stepping

Because a line of DSL code will be translated into one or more lines of Java code, the jdb "step" command cannot be used to step over actions at the DSL level (i.e., the one-to-many mapping from a line of DSL source to multiple lines of Java prohibit the simple use of "step"). The solution used in DDF is to find out the first line number of the *next* DSL statement's translated code segment. A breakpoint is then set at the next DSL

statement, and the jdb "`cont`" command is issued until that breakpoint is reached. An example of this mapping is shown in the 4th row of Figure 5: a step over at line 3 of Robot.rob (line 12 of generated Java) sets a breakpoint at line 16 (the next command in the Java mapping) followed by "`cont.`" This simulates the notion of stepping over a line in the DSL code.

5.3 Variable Values

The variables view in the debug perspective must map values from the Java state back to the equivalent DSL variables. In jdb, a variable value is obtained using the "`print`" command. For variables or fields of primitive types, the actual value is retrieved directly. In our Robot example, only three variables were used in the *generated* code (i.e., integers x, y and time). DDF will generate "print x", "print y", and "print time" to query the state of these variables from jdb. However, the Robot DSL does not have these three variables; the Robot DSL has a single Position variable, which is a composition of the x and y variables in the generated Java (see Figure 2, line 2). We reinterpret the raw data returned from jdb and reconstitute the Position value. The Position value is passed to the variables view of the debug platform.

5.4 Other Debugging Methods

Other debugging primitives, such as terminate and resume, have a straightforward mapping from the DSL debug perspective to the jdb (e.g., terminate corresponds to the jdb "`exit`" command, and resume is the "`cont`" command).

6 Conclusion

Eclipse's debugging framework provides an excellent user interface to support debugging capabilities for DSL environments. The contribution of our research integrates the Eclipse debug platform with a command-line debugger, along with generated mappings that associate lines of the DSL source to the corresponding GPL target (in this paper, the GPL is Java). The primary benefit of the approach is the ability to debug DSL programs at the level of the domain, rather than the level of the GPL.

With respect to current and future work, we are validating the approach using several DSLs of varying complexity. We also realize the aspect-oriented nature of the augmentations that were made to the existing Robot grammar in order to add debugging support. The code that was added to support the debugging features actually cross-cuts all of the productions of the DSL grammar. In Figure 3, in order to get the Robot DSL line number, every production of the terminal command has embedded code, such as (Figure 3, line 12) `dsllinenumber=dsllinenumber+1;`. For a larger and complex DSL grammar with hundreds of terminal productions, the addition of the debugging concern to the grammar would be very time consuming. We are currently working toward an aspect-oriented approach that would weave the debugging code into *grammars* (rather than source code).

More information about this research, including complete descriptions of example DSL grammars and a video, is available at the project web site: http://www.cis.uab.edu/wuh/DDF

About the Authors

Hui Wu is a Ph.D. student in the CIS Department at UAB. Jeff Gray is an Assistant Professor at UAB, and Marjan Mernik is an Associate Professor at the University of Maribor (Slovenia).

References

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

[2] R. E. Faith, *Debugging Programs After Structure-Changing Transformation*, Doctoral Dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, 1998.

[3] P. Henriques, M. Varanda Pereira, M. Mernik, M. Lenic, E. Avdicausevic, and V. Zumer, "Automatic Generation of Language-based Tools," *Electronic Notes in Theoretical Computer Science*, Vol, 65, No. 3, 2002.

[4] jdb - *The Java Debugger*, available from http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jdb.html

[5] M. Mernik, J. Heering, and A. Sloane, "When and How to Develop Domain-Specific Languages," *CWI Technical Report*, SEN-E0309, 2003. http://ftp.cwi.nl/CWIreports/SEN/SEN-E0309.pdf

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

[7] A. van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," *ACM SIGPLAN Notices*, June 2000, pp. 26-36.

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