Debugging Domain-Specific Languages

Hui Wu, Jeff Gray
University of Alabama at Birmingham
{wuh, gray} (at) cis.uab.edu

Marjan Mernik
University of Maribor, Slovenia
marjan.mernik (at) uni-mb.si

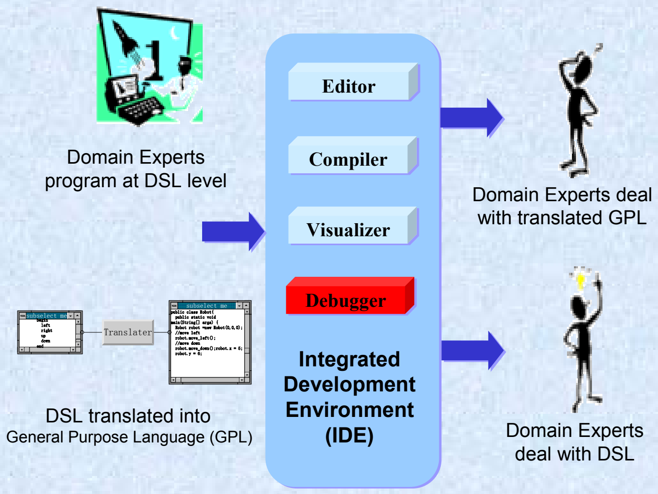
DDF
http://www.cis.uab.edu/wuh/DDF

Software Composition & Modeling Laboratory
SOFTCOM
Department of Computer and Information Sciences
University of Alabama at Birmingham

UAB

The DSL Debugger Framework (DDF) contributes 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. This poster presents the challenges of debugging with a DSL, as well as a case study describing two techniques for adding the debugging concern to a grammar.

Challenges with DSL Debugging

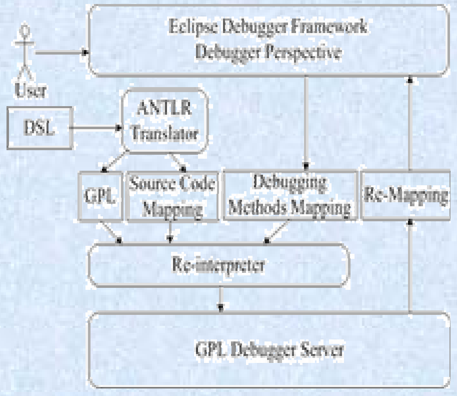


Domain Experts program at DSL level
Translator
DSL translated into General Purpose Language (GPL)
Editor
Compiler
Visualizer
Debugger
Integrated Development Environment (IDE)
Domain Experts deal with translated GPL
Domain Experts deal with DSL

Difficulties with DSL Debugger Construction

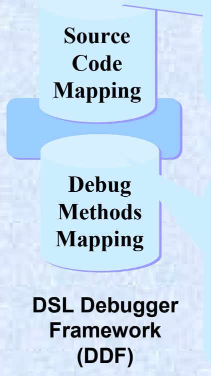
- A debugger is difficult to build because it depends heavily on the underlying operating system's capabilities and lower-level native code functionality
- Manual construction of the debugger for each new DSL can be time-consuming, expensive, and error-prone
- Domain experts lack knowledge about the underlying GPL
- 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 underlying GPL can be messy and human unreadable
- One line of GPL can be translated into tens of lines of GPL code

Architecture Overview of DSL Debugger Framework



- Reuse existing GPL standalone command line debugger
- Generative Programming techniques are used to synthesize debugger mapping code from DSL grammar specification
- Utilize the debugging methods mapping knowledge and source code mapping information to generate the DSL debugger in Eclipse
- Map the GPL debugger output messages back to the DSL level through the wrapper interface

Mapping Technique to Generate Debugger



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.

Debugging: A Crosscutting Grammar Concern

```
…
command
  :( RIGHT
    {         dsllinenumber=dsllinenumber+1;
              fileio.print("                    //move right");
              fileio.print("                    x=x+1;");
              gplbeginline=fileio.getLinenumber();
              fileio.print("                    time=time+1;");
              gplendline=fileio.getLinenumber();
              fileio.print(" ");
              filemap.print("  mapping.add(new Map("+dsllinenumber+", \"Robot.java\","+gplbeginline+","+gplendline+"));");
    }
  |LEFT
   {          dsllinenumber=dsllinenumber+1;
              fileio.print("                    //move left");
              fileio.print("                    x=x-1;");
              gplbeginline=fileio.getLinenumber();
              fileio.print("                    time=time+1;");
              gplendline=fileio.getLinenumber();
              fileio.print(" ");
              filemap.print("  mapping.add(new Map("+dsllinenumber+", \"Robot.java\","+gplbeginline+","+gplendline+"));");
   }
|UP
  {           dsllinenumber=dsllinenumber+1;
              fileio.print("                    //move up");
              fileio.print("                    y=y+1;");
              gplbeginline=fileio.getLinenumber();
              fileio.print("                    time=time+1;");
              gplendline=fileio.getLinenumber();
              fileio.print(" ");
              filemap.print("  mapping.add(new Map("+dsllinenumber+", \"Robot.java\","+gplbeginline+","+gplendline+"));");
}
…
```

To define this mapping, additional semantic actions inside each grammar production are defined. These mapping aspects crosscut the entire DSL grammar specification. Manually specifying these aspects can be expensive and error-prone, which compromises the advantages of using a DSL.

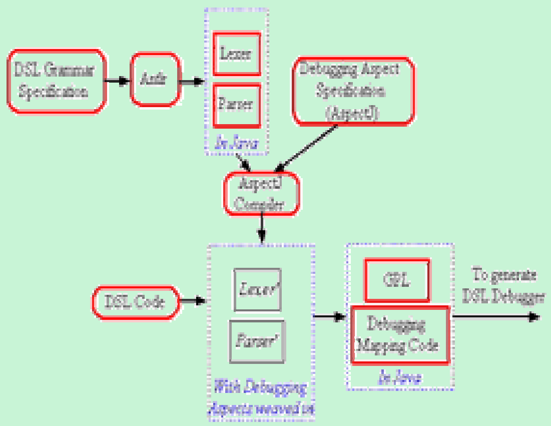
A Case Study with Two Different Approaches

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. Following is the 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
```

With the DDF, the Robot DSL debugger can be generated automatically from the DSL grammar provided that an explicit mapping is specified between the DSL and the translated GPL.

First Approach: Weaving at the Generated Code Level



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 a seamless aspect-oriented extension to Java that assists in modular implementation of numerous crosscutting concerns. In the figure above, 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.

Debugging Aspect in AspectJ

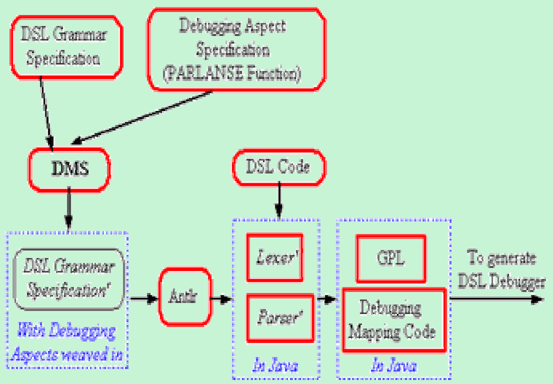
```
…
after(int commandname):
     call(void antlr.Parser.match(int))
     && args(commandname)
       { match(commandname); }

after():
     call (void P.command());
       {dsllinenumber=dsllinenumber+1;}
…
```

Disadvantages of this 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.

Second Approach: Weaving at the DSL Grammar Level



DSL Grammar Specification
Debugging Aspect Specification (PARLANSE Function)
DMS
DSL Code
DSL Grammar Specification'
With Debugging Aspects weaved in
Antlr
Lexer'
Parser'
In Java
GPL
Debugging Mapping Code
In Java
To generate DSL Debugger

The second approach involves the use of a mature program transformation system to weave the debugging concern into the *grammar*. In our work, we use the Design Maintenance System (DMS) from Semantic Designs. A debugging aspect is specified as a DMS PARLANSE function, 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). Before the grammar is even sent to 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.

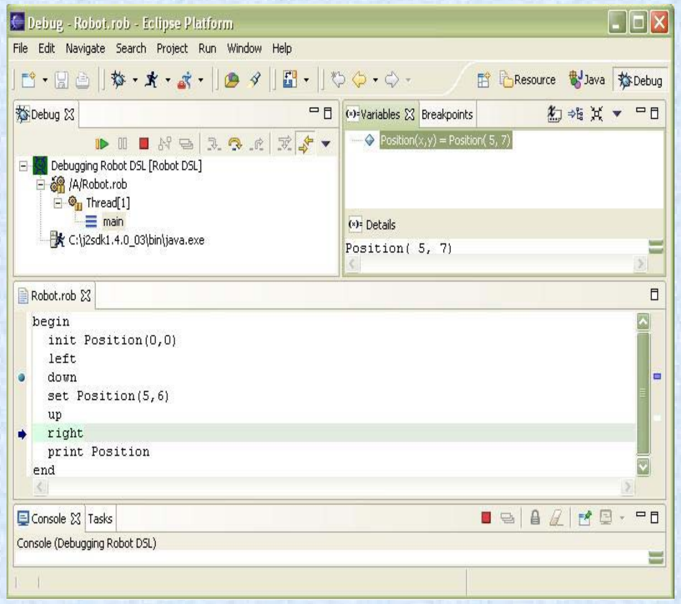
12 Steps to Weave Debugging Aspects into DSL Grammar Level Using a Program Transformation Engine (DMS)

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)

Advantages of this Approach

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.

DSL Debugger Perspective in Eclipse



Debugging Robot DSL [Robot DSL]
/A/Robot.rob
Thread[1]
main
C:\j2sdk1.4.0_03\bin\java.exe

Position(x,y) = Position(5, 7)

Position(5, 7)

```
begin
  init Position(0,0)
  left
  down
  set Position(5,6)
  up
  right
  print Position
end
```

Console (Debugging Robot DSL)

Key References:

- The AspectJ web site (http://aspectj.org).
- *ANTLR - ANother Tool for Language Recognition*, available from http://www.antlr.org.
- 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
- D. Wright and B. Freeman-Benson, "How to Write an Eclipse Debugger," *Eclipse Corner*, Fall 2004, http://www.eclipse.org/articles/index.html
- 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.