

# **Grammar-Driven Generation of Domain-Specific Language Debuggers**

Hui Wu<sup>1</sup>, Jeff Gray<sup>1</sup>, and Marjan Mernik<sup>2</sup>

<sup>1</sup> Dept. of Computer and Information Sciences, University of Alabama at Birmingham  
Birmingham AL 35294-1170  
{wuh, gray} @ cis.uab.edu

<sup>2</sup> Faculty of Electrical Engineering and Computer Science, University of Maribor  
2000 Maribor, Slovenia  
marjan.mernik @ uni-mb.si

## ABSTRACT

Domain-Specific Languages (DSLs) assist a software developer (or end-user) in writing a program using idioms that are similar to the abstractions found in a specific problem domain. Tool support for DSLs is lacking when compared to the capabilities provided for standard General-Purpose Languages (GPLs), such as Java and C++. For example, support for debugging a program written in a DSL is often non-existent. The lack of a debugger at the proper abstraction level limits an end-user's ability to discover and locate faults in a DSL program. This paper describes a grammar-driven technique to build a debugging tool generation framework from existing DSL grammars. The DSL grammars are used to generate the hooks needed to interface with a supporting infrastructure constructed for an Integrated Development Environment (IDE) that assists in debugging a program written in a DSL. The contribution represents a coordinated approach to bring essential software tools (e.g., debuggers) to different types of DSLs (e.g., imperative, declarative, and hybrid). This approach hides from the end-users the accidental complexities associated with expanding the focus of a language environment to include debuggers. The research described in the paper addresses a long-term goal of empowering end-users with development tools for particular DSL problem domains at the proper level of abstraction without depending on a specific GPL.

**Keywords** *Debugging, DSL, GPL, Grammars, Program Environments-Integrated Environments.*

## 1. INTRODUCTION

The advancement of end-user programming tools has empowered those who are not traditional programmers with an ability to create their own software solutions [55]. Experts who have a strong understanding of a problem domain, but no formal computer programming training, can write software applications to solve a specific need in their daily work tasks. The ability to create a software solution is no longer the privilege of a computer scientist – in some cases, training in traditional programming is not necessary (e.g., one of the most widely available end-user programming environments is the spreadsheet [14], which can be programmed by scripts that use arithmetic and statistical formulas of a specific

domain instead of using traditional programming language concepts [13]). It has been estimated that only a small fraction of software developers are actually professional developers (e.g., in the United States, Scaffidi et al. [51] estimate that there are approximately 2.75 million professional developers out of an estimated 80 million end-user programmers), with the vast majority of end-user developers building applications using tools such as spreadsheets, query systems, or interactive scripting websites.

In addition to offering advantages toward end-user programming, Domain-Specific Languages (DSLs) also offer benefits to traditional software developers by providing specialized languages to support activities within the software development process. Some of the more popular DSLs include the language used in the Unix `make` utility [44] and the language used to specify grammars in parser generators like YACC [39]. Other examples include the Very High Speed Integrated Circuit Hardware Description Language (VHDL) [4], which is a DSL to model a digital hardware system; Structured Query Language (SQL) is a DSL to access and manipulate databases [31]; Graphics Adaptor Language (GAL) is a DSL to specify video device drivers [56]; and Mawl is a DSL to specify form-based services in a device-independent manner [5].

## **1.1 The Benefits of Domain-Specific Languages**

To assist end-users in describing solutions to their work tasks, DSLs [58] have been suggested as an approach to remove the dependence on traditional General-Purpose Languages (GPLs), such as Java and C++. A DSL is a language tailored toward the specific needs of a particular problem domain to ease the development of software solutions for that domain [20], [41]. DSLs are often described as “little languages” [9], [19] that are designed to solve problems in particular domains.

The intent of DSLs is to assist end-users in writing more concise, descriptive, and platform-independent programs. This is enabled because the domain knowledge is specified at the appropriate level of abstraction, which is independent of the implementation platform [19]. The goal of providing suitable programming abstractions for end-users is also a key principle of intentional programming [53]. Modifications to DSL programs are easier to make and can be understood and validated by domain

experts who do not know how to program in a GPL. The empirical evidence suggests that the use of DSLs increases flexibility, productivity, reliability, and usability [36], [41], [57].

## **1.2 The Challenges of DSL Implementation**

There is a distinction between the end-user programmers that use a DSL and the language designers who specify the DSL and implement the required tools (e.g., the DSL compiler). The design and implementation of a DSL can be challenging and expensive [38]. The development of a DSL requires knowledge of programming language implementation, as well as domain knowledge. Building a debugger for each DSL from scratch can be time consuming, error-prone, and costly. It is difficult to build new source-level debuggers [63] for each new language of interest and for each supported platform because each debugger depends heavily on the underlying operating system's capabilities and lower level native code functionality [49]. The main goal of this paper is to show how the effort of developing DSL tools can be minimized by an automated grammar-driven tool generation approach that extends a popular Integrated Development Environment (IDE). At the experimental evaluation phase of this research, the research results reveal the reusable and non-reusable software components of the framework. Reusable components deal with the commonality across debugging and testing behavior, and non-reusable components handle the specificity across different DSLs and different test tools. A detailed analysis of the reusability of these components is described in Section 4.4.

Mernik et al. have categorized various DSL implementation patterns as: interpreter, compiler/application generator, pre-processor, embedding, extensible compiler/interpreter, commercial off-the-shelf, and hybrid [41]. The majority of the DSL implementation patterns translate a single DSL construct into several constructs in a target GPL. The most popular strategies for implementing a DSL, which are also adopted in this paper, are those DSL implementation patterns where DSL code is translated to GPL code, including: compiler, extensible compiler, and preprocessing (macro-processing, source-to-source, pipelining and lexical processing). Translating a DSL to an existing GPL is a popular implementation approach because the underlying tools of the converted GPL can be used to obtain an

executable application. It is very convenient to express new DSL constructs in terms of GPL constructs, and the well-developed GPL tools can be reused (e.g., compiler, profiler, and debugger).

Although direct reuse of the existing GPL tools offers several benefits, a specific GPL tool does not provide the proper abstractions that are needed by end-users who often lack knowledge about the underlying GPL (i.e., a GPL may be difficult to understand by those not trained as programmers because the conceptual solution expressed in a GPL is not close enough to the specific problem domain). 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 an end-user to debug the translated program at the GPL level [32], [60]. An approach that hides the underlying use of the GPL tools offers a level of transparency that can remove the accidental complexities that cause the abstraction mismatch between the DSL and GPL; such an approach is advocated by Van Wyk and Johnson, where they argue for the need to perform analysis at the DSL level, not the GPL level [62]. Although techniques for constructing DSL tools (e.g., editor and compiler) have been developed over the years, support for debuggers for DSLs have not been investigated deeply. This paper describes how automated tool generation can be used to overcome the lack of debugging tool support for end-user application programmers. As Robert Floyd noted in his Turing Award speech, there is no need for a “shiny new language” unless it supports the programming methods and paradigm used by programmers [27]. The same comment can apply to DSL tools; i.e., if supporting tools needed by an end-user programmer are not available, then the utility of a new DSL is diminished.

Among the DSL implementation approaches, the lower level GPL can be considered a base machine and the higher abstraction provided by the DSL represents a virtual machine for the particular domain. If the virtual machine is completely transparent, any state or sequence of states obtained by the base machine can also be realized in the virtual machine. If the virtual machine exhibits loss of transparency, there exists a subset of states obtained by the base machine that cannot be represented in the virtual machine [45]. However, these states should not be reachable in a valid implementation of the

virtual machine. In many cases, including pre-processor implementation of DSLs, loss of transparency can be considered a desired property (i.e., it is often not necessary or desirable for the DSL to have a one-to-one mapping of all of the features available in a GPL). To realize the objectives of grammar-driven generation of DSL tools, all of the specified tool behaviors of the virtual machine are made available through a mapping to the base machine. The mapping transformation from the virtual machine to the base machine is an essential part of our approach. Specifically, the reverse mapping is the distinct feature of our approach. The program behavior of a specific tool is dependent on the translation process from the DSL to a GPL, which is modeled as one line of the DSL code mapped to an interval with a first and last line of the GPL code, as defined in the DSL grammar specification.

### **1.3 Background: The Eclipse Debugging Perspective**

Eclipse is an open-source development platform for constructing customized Integrated Development Environments (IDEs) that can be used to create diverse applications [24]. To provide the necessary background of the basic tools and techniques mentioned throughout the paper, this sub-section offers a brief description of the Eclipse debugging platform. A key characteristic of Eclipse is the ability to serve as a tool integration platform that offers numerous extension points for customization through a plug-in architecture. Eclipse has been defined as a collection of “places-to-plug-things-into” (extension points) and “things-to-plug-in” (extensions) [28]. An Eclipse plug-in has the ability to integrate with other plug-ins to extend functionality. New functionality and features are implemented as layered plug-ins. One plug-in can extend the functionality of another plug-in by implementing the interface defined by the extension point of the other. Developers can provide new functionality to Eclipse by extending several existing extension points, and at the same time provide further development opportunities for others by publicizing new extension points.

Eclipse is capable of integrating new functionality from different developers while preserving a seamless user interface and consistent user experience. The Eclipse Plug-in Development Environment (PDE) [24] offers a powerful platform to integrate different language tools that support DSL

development. To assist in DSL debugger construction, the Eclipse Software Development Kit (SDK) provides the debugging perspective, which is a framework for building and integrating custom debuggers. As shown in Figure 1, the debug perspective defines a set of interfaces that model common debugging artifacts (e.g., threads, variables, and breakpoints) and navigation actions (e.g., stepping, suspending, resuming, and terminating) [59]. Although the debugging perspective does not provide a specific implementation of a debugger, it does offer a basic debugger user interface that can be adapted and extended with features specific to a particular language. The basic debugger user interface listens to the events from the debug model interface and updates the contents according to the information from the debug events. In Section 4, we describe how we extended the Eclipse debugging perspective to integrate with an interactive debugging framework that assists in debugging a program written in a DSL.



Figure 1. Screenshot of the Eclipse Debug Perspective

## 1.4 Paper Overview

The research described in this paper offers two key contributions. The first contribution provides an initial step toward empowering end-user developers with traditional software engineering debugging capabilities at the DSL level. To accomplish this objective, we introduce a grammar-driven DSL tool framework that generates a debugger automatically from DSL grammar specifications. The pre-existing base DSL grammars are transformed to generate the hooks needed to interface with a supporting plug-in infrastructure written for an IDE. The second contribution is a presentation of techniques for debugging a diverse set of DSLs. These two contributions represent further investigation into the idea of Grammarware [37], which comprises grammars and all grammar-dependent software. The paper presents

grammar-driven generation of DSL debuggers that considers grammars as first-class artifacts that demand an engineering approach to grammar evolution (e.g., the source code mappings described in this paper adopt an engineering approach to grammar augmentation, rather than ad hoc evolution of source code). The research approach presented in this paper augments the DSL grammar with additional hooks to the DSL debugging framework. The grammar-dependent lexer and parser propagate the modification through the DSL parsing phase where the generated information serves as input into our DSL debugger generation framework.

The remainder of this paper is organized as follows: Section 2 presents an overview of the research contributions described in the paper; Section 3 introduces the debugging tool generation framework we have implemented, including the techniques and algorithms involved; Section 4 illustrates the instantiation of the debugging framework on three types of DSLs; Section 5 discusses related work in two categories; Section 6 offers a summary of the research and describes several limitations that point toward future work.

## **2. CATEGORIES OF DSL DEBUGGERS**

An overview of the research objectives described in the paper is shown in Figure 2, which illustrates a DSL tool framework that can generate various testing tools (e.g., debuggers, test engines, and profilers) for different categories of DSLs (e.g., imperative, declarative, and hybrid). In this paper we solely focus on DSL debuggers. The topic of DSL profilers is an area of future work. We have performed initial research into the generation of DSL unit testing tools, but that is out of the scope of this paper.

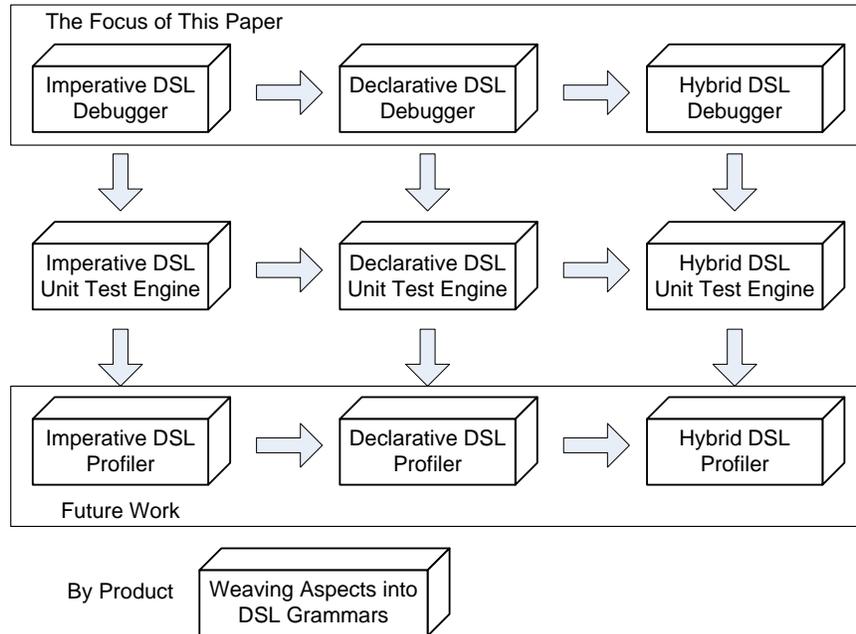


Figure 2. Matrix of DSL Tools and Language Categories

The automation provided by generative programming [18] offers an extensible mechanism as an alternative to manual tool construction. This approach conforms to the software development paradigm of step-wise refinement applied to language tool construction [8]. To realize this generative approach, we have implemented the DSL Debugger Framework (DDF), which reuses the existing debugging support in Eclipse and Java. A by-product of this research (not presented in this paper) defines an aspect language that can weave crosscutting tool concerns directly into a DSL grammar. More details about our use of aspects to weave debugging and testing concerns into grammars are presented in [61] and demonstrated through videos at the project web site [23]. The following sub-sections describe a research matrix (see Figure 2) focused on debuggers for different categories of DSLs.

## 2.1 DSL Debuggers

The tools that are generated from our grammar-driven approach are DSL debuggers applied to three types of DSLs. A debugger enables programmers to inspect and discover the errors in their programs efficiently during program execution. Zellweger categorizes GPL debuggers into two classes: expected behavior debugger and truthful behavior debugger [63]. An expected behavior debugger hides the program optimization and transformation from the programmers and “always responds exactly as it

would for an un-optimized version of the same program” [63]. A truthful behavior debugger “displays how optimizations have changed the program portion under consideration or it admits that it cannot give a correct response” [63]. As categorized by Auguston [6], the behavioral models of higher level debugging mechanisms can be specified (e.g., debugging queries, path expressions, assertion checkers, and event tracers) to generate new categories of debuggers (e.g., algorithmic debuggers, declarative debuggers, and event-based debuggers [7]). Our work described in this paper represents expected behavior debuggers that perform typical debugging tasks on DSL programs (e.g., set a break point, stop at a break point, display variable values, and step through the program), which enables a better understanding of the run-time behavior of a DSL program.

## **2.2 DSL Categories**

The horizontal direction of Figure 2 focuses on facilitating the construction of the same software tool (e.g., debugger) across different categories of DSLs (e.g., imperative DSL, declarative DSL, and hybrid DSL). This section introduces the specific differences among the categories of DSLs described in the paper.

### *2.2.1 Imperative DSL*

An imperative programming language is based on the von Neumann concept that is centered on assignment expressions and control flow statements [52], which allows a program to change the content of cells in memory. In an imperative language, the state change of variable values is a central feature of interest. Therefore, for imperative languages, debuggers are designed around capabilities to examine the value of variables at run-time.

```

...
15     knight:
16         position(+0,+1);
17         position(+0,+1);
18         position(+1,+0);
19     knight:
20     ...
21     Init position(0,0);
22     left;
23     down;
24     knight;
25     Set position(5,6);
26     up;
27     right;
28     Print position;
...

```

Figure 3. Robot DSL Sample Code

For the purpose of this paper, a simple imperative language for representing robot control was adopted from a previous case study [42], [60]. A simple language like the Robot DSL is used so that the discussion is not hindered by the complexities of the DSL itself. In this paper, the Robot language has been extended by adding user-defined functions. This extension is useful to demonstrate the **Step Into** functionality of the DSL Debugging Framework (DDF). The Robot DSL consists of four primitive moves that control robot movement: up, down, right, and left. Users can define other moves (e.g., knight). Every move increases or decreases the position of the robot along the **x** or **y** coordinates. Additional Robot DSL statements are: initial statement, set statement, and print statement. Figure 3 represents sample code written in the Robot DSL – lines 15 to 19 represent the user-defined function named **knight**; line 21 sets the robot’s initial position to  $\langle 0, 0 \rangle$ ; line 24 invokes **knight**; line 25 forces  $\langle 5, 6 \rangle$  as the robot’s new current position; and line 28 prints the robot’s current position.

### 2.2.2 Declarative DSL

A declarative programming language is based on declarations that state the relationship between inputs and outputs. Declarative programs consist of declarations rather than assignment or control flow statements. The declaration semantics have a precise interpretation that is closer to the problem domain. Such programs do not state how to solve a problem, but rather describe the essence of a problem and let the language environment determine how to obtain a result [52]. Instead of assessing the value of

individual variables, a declarative DSL debugger needs to evaluate the relationships between each declaration, which are often represented as data structures with symbolic logic.

Car features in FDL	All possible car configurations
<b>feature 1:</b> Car: all (carbody, Transmission, Engine, Horsepower, opt(pullsTrailer))	<b>1:</b> (carbody, pullsTrailer, manual, highPower, gasoline, electric)
<b>feature 2:</b> Transmission: oneof (automatic, manual)	<b>2:</b> (carbody, pullsTrailer, manual, highPower, electric)
<b>feature 3:</b> Engine: moreof (electric, gasoline)	<b>3:</b> (carbody, pullsTrailer, manual, highPower, gasoline)
<b>feature 4:</b> Horsepower: oneof (lowPower, mediumPower, highPower)	<b>4:</b> (carbody, pullsTrailer, automatic, highPower, gasoline, electric)
<b>feature 5:</b> include pullsTrailer	<b>5:</b> (carbody, pullsTrailer, automatic, highPower, electric)
<b>feature 6:</b> pullsTrailer requires highPower	<b>6:</b> (carbody, pullsTrailer, automatic, highPower, gasoline)

Figure 4. Car Features Specified in FDL and List of Possible Car Configurations (adapted from [21])

As an example of a declarative DSL, the Feature Description Language (FDL) is used in this paper to specify the legal configuration of an automobile product line [21]. The FDL is a textual language that describes a feature diagram [18], which represents a hierarchical decomposition of domain features arranged by composition rules (e.g., mandatory, alternative, or optional). FDL can be used to analyze all possible features in the development of families of related systems. The left side of Figure 4 is an example specification written in FDL to describe a simple car. According to the first feature, a **Car** consists of four mandatory features: **carbody**, **Transmission**, **Engine**, and **Horsepower**. As shown at the end of feature 1, feature **pullsTrailer** is an optional feature (i.e., a car can either have a **pullsTrailer** or not). If the first character of a feature is a lowercase character, the feature represents a primitive that cannot be expanded further (e.g., the **carbody** feature). The right side of Figure 4 enumerates all of the possible legal configurations that result from the features defined on the left side of the figure.

```

1      START ::=
2          begin
3          COMMANDS
4          end
5          ;
6      COMMANDS ::= =
7          COMMAND
8          COMMANDS
9          | epsilon
10         ;
11     COMMAND ::= =
12         left
13         | right
14         | up
15         | down
16         ;

```

Figure 5. Robot Language Syntax Specification in BNF Format

As another example, Backus–Naur Form (BNF) is a declarative DSL for formally describing the syntax of a language and used to express context-free grammars [1]. BNF is a widely used grammar notation to verify the instances of a language, to analyze the language features, and to generate the lexer and parser or other language tools. The syntax specification of the Robot language in BNF is shown in Figure 5, where the uppercase symbols represent non-terminals and the lowercase symbols represent terminals. Context-free productions are specified using terminals and non-terminals in this Robot BNF (e.g., **START ::= begin COMMANDS end**). This grammar is a slight simplification of the Robot language described in Section 2.2.1.

### 2.2.3 Hybrid DSL

Bravenboer and Visser have investigated the concrete syntax for languages that assimilate *embedded* DSL code into the surrounding GPL code to provide the appropriate notation for expressing domain composition [12]. Conversely, some DSLs embed GPL code within the DSL program. We call such examples *hybrid* DSLs. The linguistic extension provided by the GPL is used frequently in many DSLs and is named the *piggyback DSL design pattern* [41], [54]. The piggyback pattern is widely adopted in DSLs for tools like parser generators, such as Yet Another Compiler-Compiler (YACC), Bison [39], ANother Tool for Language Recognition (ANTLR) [46] or the Constructor of Useful Parsers (CUP) [17]. The semantic actions in the grammar specification used by a parser generator are described in GPL code

(e.g., Java and C++), which are surrounded by DSL constructs corresponding to the grammar of the language. For the hybrid DSL case studies described in this paper, we apply our framework to two different types of hybrid DSLs. One case is when the GPL notation is considered the host language and the DSL is embedded in the surrounding GPL code; the opposite case is when the DSL is the host language and the GPL is embedded in the surrounding DSL code. To support end-user programming, DSL debuggers should be able to debug hybrid DSLs by switching language modes between two different language domains.

When developing graphical user interfaces for Java, traditional Swing [40] user interface code is intertwined together so that it is hard to determine the actual structure of the end result of the visual representation. Figure 6 is an example of a hybrid DSL called the Swing User-interface Language (SWUL), which was first introduced as an example DSL by Bravenboer and Visser [12]. SWUL is a hybrid DSL that assists in constructing a Java Swing user interface in a more comprehensive and structured way. The SWUL program is embedded into a Java program. The SWUL program is assimilated into the surrounding Java code through translation into pure Java Swing code. Ideally, programmers should be able to debug through the SWUL code between two different language notations (e.g., Java and SWUL) rather than the generated Java code. SWUL provides syntax for a block module that can describe the user interface and add concrete syntax into the whole DSL program. As an example, Lines 6 to 18 of Figure 6 show SWUL code surrounded by Java code. In this specific case, the SWUL code specifies a JFrame layout containing one JLabel in the middle (Line 9) and two JButtons (Line 11 to 14). As this example illustrates, a hybrid DSL's ability to escape to a GPL provides a simple language extension mechanism.

```
1.  import javax.swing.*;
2.  import java.awt.*;
3.
4.  public class WelcomeSwing {
5.      public static void main(String[] ps) {
6.          JFrame frame = frame {
7.              title = "Welcome!"
8.              content = panel of border layout {
9.                  center = label { text = "Hello World" }
10.                 south = panel of grid layout {
11.                     row = { button {
12.                         text = "cancel" }
13.                         button {
14.                             text = "ok" }
15.                     }
16.                 }
17.             }
18.         };
19.         frame.pack();
20.         frame.setVisible(true);
21.     }
22. }
```

Figure 6. SWUL Sample Code

Figure 7 is another example of a hybrid DSL that represents an extension of the Robot DSL from Section 2.2.1. The original Robot DSL does not provide native constructs to handle I/O operation, user-interface, and random number generation. A hybrid version of the Robot DSL provides syntax for a block module that can add Java code fragments into the DSL program, which can be used to implement the functionality not provided in the original language. As an example, line 13 of Figure 7 introduces a new random move that requests from the user the boundaries for a random number generator that produces random coordinates and lines 13 to 27 represent the method definition of the random move (lines 14 to 26 specify the semantics of random as written in Java). Line 35 is the code in the main part of the DSL that calls random. A hybrid DSL’s ability to escape to a GPL provides a simple language extension mechanism. DSL design is often an iterative process guided by user feedback. Frequent escape to the GPL level may suggest that a new construct should be added in the next version of the DSL to support a commonly needed feature.

```

...
13  random:
14  {
15    String answer;
16    int max;
17    JOptionPane myGUI = new JOptionPane();
18    Random rand = new Random();
19    answer = myGUI.showInputDialog("Please enter the upper bound of the ...
20    max = Integer.parseInt(answer);
21    x = rand.nextInt(max);
22    answer = myGUI.showInputDialog("Please enter the upper bound of the ...
23    max = Integer.parseInt(answer);
24    y = rand.nextInt(max);
25    myGUI.showMessageDialog(null, "Generated Position(" + x + "," + y + ")");
26  }
27  random:
28  ...
29  Init position(0,0);
30  left;
31  down;
32  knight;
33  Set position(5,6);
34  up;
35  random;
...

```

Figure 7. Hybrid Robot DSL

### 3 DSL DEBUGGING FRAMEWORK (DDF)

The DSL Debugging Framework (DDF) provides a grammar-driven technique for reusing an existing GPL debugger in conjunction with the debugging interface. An illustrative overview of the DDF is shown in Figure 8. A key technique of the DDF is a mapping process that records the correspondence between the DSL and the generated GPL. The translator generates GPL code and mapping information from the DSL source. The DDF requires mapping information that depends on both the source language (DSL) and the target language (GPL). The mapping components comprise the source code mapping, debugging methods mapping, and debugging results mapping components (middle of Figure 8). The results from these first two mapping processes are reinterpreted into the GPL debugger server as debugging commands, along with parameters provided to the translated GPL code. Important information that is captured in the mapping are: (1) the representation of the source-level language; (2) a function that defines how values in the DSL are represented on the target GPL, and (3) a specification that states how such values should be displayed to the end-user in the debugging perspective [50]. The *source code mapping component* uses the generated mapping information to determine which line of the DSL code is

mapped to the corresponding segment of GPL code. Source code mapping indicates the location of the GPL code segment corresponding to a single line of code in the DSL. The *debugging methods mapping component* receives the end-user's debugging commands from the DSL debugging 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 debugging server responds to the debugging commands sent from the re-interpreter component. The debug result at the GPL level is sent back to the debugging perspective by the *debugging results mapping component*, which is a wrapper interface to convert the GPL debugging result messages back into the DSL level. Because the messages from the GPL debugger are command-line outputs, which know nothing of the DSL or the debug perspective, it is necessary to remap the results to the end-user perspective. As a result, the DDF enables the end-user to interact directly with the debugging perspective at the DSL level.

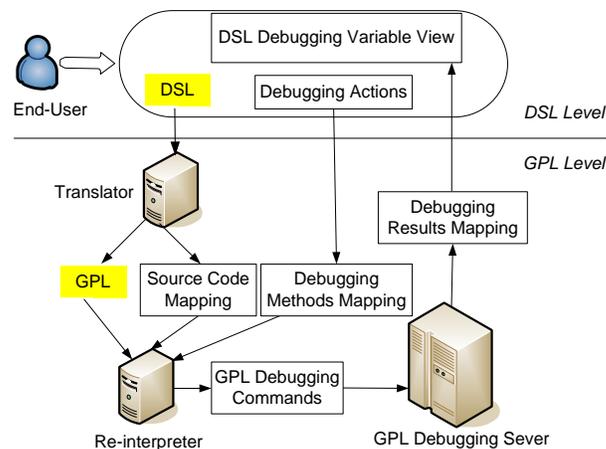


Figure 8. The DSL Debugging Framework (DDF)

### 3.1 Source Code Mapping

As a side-effect of the source-to-source translation process of the DSL processor, the source code mapping information is generated when a DSL source file is translated into an equivalent GPL representation. The translation rules are defined in the DSL grammar. During the translation process, the base grammar of the DSL is augmented with additional semantic actions that generate the source code

mapping needed to create the DSL debugger. The mapping contains the following information, which is stored in a vector: (1) the DSL line number; (2) the translated GPL file name; (3) the line number of the first line of the corresponding code segment in GPL; (4) the line number of the last line of the corresponding code segment in the GPL; (5) the function name of the current DSL line location, and (6) the statement type at the current DSL line location. The statement types can be **functiondefinition**, **functioncall**, or **none**.

A **functiondefinition** consists of **functionhead**, **functionbody**, and **functionend**, where: **functionhead** is the beginning of a function (line 3 on the left side of Figure 9 is the **functionhead** of knight); **functionbody** is the actual definition of a function (lines 4 to 6 on the left side of Figure 9 represent the **functionbody** of knight); **functionend** is the end of a function (line 7 on the left side of Figure 9 is the **functionend** of knight). A **functioncall** is the name of the function being called from another location of a program. The statement type for a built-in method of a Robot program is set to **none**. For example, the mapping information at Robot DSL line 13 in Figure 9 is {13, "Robot.java", 20, 21, "main", "none"}. This vector indicates that line 13 of the Robot DSL is translated into lines 20 to 21 in Robot.java, designating the “**Set position()**” method call inside of the **main** function. For each line of the Robot DSL code, there is corresponding mapping information specified in the same format. Although the examples presented in this section are tied to Java and the simple Robot DSL, the source code mapping and interaction with the GPL debugger and debug platform can be separated from any specific DSL and GPL. The Eclipse debug perspective is GPL-independent. Thus, the DDF can be used with any GPL that has an existing debugger.

<pre> ... 3 knight: 4   position(+0,+1); 5   position(+0,+1); 6   position(+1,+0); 7 knight: 8 ... 9   Init position(0,0); 10  left; 11  down; 12  knight; 13  Set position(5,6); 14  up; 15  right; 16  Print position; ... </pre>	<pre> ... 6 public static void move_knight(){ 7   x=x+0; 8   y=y+1; 9   x=x+0; 10  y=y+1; 11  x=x+1; 12  y=y+0;} 13 public static void main(String[] args) { 14   x=0; 15   y=0; ... 18   move_knight(); ... 20   x = 5; 21   y = 6; ... 26   System.out.println("x coordinate= "+x+" "+ 27                       "y coordinate= " + y);} ... </pre>
---	--

a) Robot DSL

b) Generated Java

Figure 9. Robot DSL Source Code Mapping

### 3.2 Debugging Methods Mapping

The traditional debugging activities of a GPL include setting or clearing a breakpoint, stepping over, stepping into, terminating a debug session, and resuming execution [49]. These debug actions are also suitable for end-users debugging a DSL program. All of the debugging mapping knowledge is pre-defined within the algorithms in the DDF. These algorithms are designed in a general manner to work with many cases of the different types of DSLs defined within this paper (i.e., imperative and hybrid). However, several minor adjustments to the algorithms may be needed in some cases, such as particular features within declarative DSLs. The quantitative measurement of such adaptation is presented in Section 4.4. The specifics of the required debugging method mapping are illustrated in Figure 10, with the type of mappings named in the first column, the DSL debugging actions specified in column two, and the respective GPL debugging actions in column three.

In Figure 10, the second row indicates that DSL line number **n<sub>i</sub>** is mapped to a segment of GPL code from line number **m<sub>i</sub>** to **m<sub>j</sub>** (as shown in column 3). Among the debugging actions, **step** is the most useful and complicated action. Except for the **Step Over** and **Step Into** actions, the

remaining GPL debugging actions have straightforward mappings (i.e., the same debugging action requested on line  $n_i$  of the DSL is mapped to the same action on line  $m_i$  of the generated GPL).

Mapping	DSL	GPL
Source Code $n_i$ maps to $m_i$ to $m_j$	Line Number: $n_1$ $n_2$ $n_{\dots}$ $n_i$ $n_{i+1}$ $n_{\dots}$ $n_j$ $n_{j+1}$ $n_{\dots}$	Line Number: $m_1$ $m_2$ $m_{\dots}$ $m_i$ $m_{i+1}$ $m_{\dots}$ $m_j$ $m_{j+1}$ $m_{\dots}$
Breakpoint	Set breakpoint at $n_i$	Set breakpoint at $m_i$
Step Over	Step over line at $n_i$	Step Over algorithm
Step Into	Step into line at $n_i$	Step Into algorithm
Terminate	Terminate at line $n_i$	Terminate at line $m_i$
Resume	Resume at line $n_i$	Resume at line $m_i$

Figure 10. Mapping of Debugging Actions between DSL and GPL

Because there is an abstraction mismatch between the DSL and the generated GPL code, the step debugging actions cannot be mapped directly. In order to simulate an end-user's step through action (e.g., to examine the values of DSL variables), the interaction with the underlying GPL must be coordinated to match the view of the end-user. To enable such coordination, a series of GPL debugging commands are issued to perform equivalent debugging tasks at the DSL code level. During a stepping action within the DDF, the DSL Debugging Step Over Algorithm is invoked (see Figure 11). This algorithm requires information from each line of the DSL program (e.g., function types and function names generated from the DSL grammar). This information will help the algorithm to distinguish whether the current DSL program statement is a function call or a function definition. This algorithm is responsible for matching the language abstraction gap between the DSL and GPL at the source code level. End-users can perform a **Step Over** action at either the main function level or within individual function definitions. A **Step Into** action may also be performed at the function call level if a corresponding function definition exists. The **Step Into** action is disabled if the current statement has no function definition.

```

1  if (function name equals "main") {
2      if (dsl_line_number < last line number of DSL code) {
3          set breakpoint at gpl_line_number corresponding to dsl_line_number+1
4          call cont()
5      }
6      else {
7          call cont()
8          step over last line of DSL code, debugging session terminated
9      }
10     current dsl_line_number increased by one;
11 }
12 else {
13     get function_type from mapping information base
14     if (function_type equals "functionbody") {
15         current dsl_line_number increased by one
16         for all the statements corresponding to this one line of DSL code {
17             call step()
18         }
19     } else if (function_type equals "functionend") {
20         call step()
21         assign current dsl_line_number as previous_dsl_line_number + 1
22     }
23 }

```

Figure 11. DSL Debugging Step Over Algorithm

In Figure 11, according to the function name of the current line of the DSL source code, the first condition (see line 1) is used to determine where the **Step Over** action is taking place (e.g., at the main function level or at the user-defined function definition level). The **dsl\_line\_number** is the current execution position at the DSL level. The **gpl\_line\_number** is the current execution position at the GPL level. If the current program pointer is within the main function level, the DDF sets up a breakpoint at the GPL level at location **gpl\_line\_number**, which is the beginning GPL line of the corresponding DSL line. The **cont** method is a sub-routine that continues execution of the debugged application until the debug session is stopped at a breakpoint or terminated. Line 10 increases the current DSL line number after the **Step Over** action is completed.

When the current program pointer is at the function definition level, the step over action performs differently. If the current DSL statement's function type is **functionbody**, unlike the situation in the main program, the function definition may cascade another function definition where the source code mapping information is not sufficient to determine the line number of the intended program execution location. Therefore, a different strategy is used in this case. Stepping over one line of DSL is equivalent to performing an iteration of steps through many lines of GPL code, because one line of DSL code corresponds to a sequence of GPL code. The number of iterations (line 16) can be computed by

subtracting the `beginning_line_number+1` from the `ending_line_number`. Each iteration performs the GPL `step()` sub-routine on the GPL code, which only advances execution to the next line. When the user steps through the last line of a function (indicated by function type `functionend` in line 19), the algorithm invokes the GPL `step()` method only once, which moves the program pointer out of the function definition and back to the next line of the GPL code before the DSL `Step Over` action. To synchronize the line number at the DSL code level, the current program pointer is moved to line `previous_dsl_line_number+1`, which is the next line before the DSL `Step Over` at the function definition in the DSL. The variable called `previous_dsl_line_number` is a temporary counter that stores the line number before the user executes the DSL `Step Over` action on a function definition. All of the corresponding GPL line numbers, function name, and types come from the source code mapping information (e.g., line 13). Although this step over algorithm is generalized to be used in most case studies described in this paper, the different meanings of step into and step over for a declarative DSL (e.g., a language grammar) require minor adjustments in this algorithm to handle new requirements, which are described in Section 4.2.

Note that the choice of the GPL debugger depends on the kind of GPL code generated from the DSL implementation. In the Robot language example, the generated GPL code is Java, which influenced the choice to use the jdb. If the generated GPL is C/C++, Fortran, or Pascal, the GNU Project Debugger (gdb) [29] can be selected as the new GPL debugging server. The gdb also provides the stand-alone command-line debugging services for several GPLs (e.g., C/C++, Fortran, and Pascal) other than Java. If the generated GPL is C#, VB.Net, managed C++, or J# we can choose the Microsoft Runtime Debugger (called Cordbg.exe). The Cordbg is the debugger for Microsoft Visual Studio .Net environment supported languages.

### 3.3 Debugging Results Mapping

The debugging results from the GPL debugger are returned in the GPL context (i.e., GPL variable names and results), which is not at the correct level of abstraction for end-users. Thus, the debugging results from the GPL debugger must be mapped back to the DSL debugging perspective so that end-users can understand the meaning of the results. The one-to-many mapping between the DSL and GPL can be captured by augmenting the base DSL grammar with additional code that describes the mapping in specific grammar productions.

#### *3.3.1 Debugging Results Mapping Process*

The DDF captures the debugging results by reading the output of the GPL debugging server's response to the sequence of GPL debugging commands. These debugging results sometimes are meaningless for DSL programmers unless they get proper interpretation. Also, the debugging results from the GPL debugging server may contain many symbols that are not needed in the DSL context (e.g., command prompt symbols, spaces, tabs, and newlines). The first step towards debugging results mapping is to sanitize the raw GPL debugging results. A clean up function is implemented to handle the first step of the reverse mapping. The second step is to retrieve the necessary information from the sanitized results and compose them into the format that the IDE debugging perspective can display properly. The objective is to allow the results of the GPL debugging server to be displayed to the end-user in the proper context of the DSL. In some cases, the debugging result mappings may exist within the DSL compiler such that the DDF can directly use the interface functions that the compiler provides to reveal variable values.

### 3.3.2 Debugging Results Mapping Example

```
1 | INIT var:VARIABLES LPAREN init_num1:NUMBER COMMA init_num2:NUMBER RPAREN
2 | {
3 |     dsllineline=dsllineline+1;
4 |     fileio.print("x="+init_num1.getText()+"");
5 |     fileresult.print("x_coordinate=print x");
6 |     gplbeginline=fileio.getLineNumber();
7 |     fileio.print("y="+init_num2.getText()+"");
8 |     fileresult.print("y_coordinate=print y");
9 |     fileio.print("time=0"+"");
10 |    gplendline=fileio.getLineNumber();
11 |    filemap.print("mapping.add(new Map("+dsllineline+", \"Robot.java\", "+gplbeginline +
12 |                "+gplendline+", "+\"\""+funcname+\"\""+", "+\"\""+funcall+\"\""+"));");
13 |    fileresult.print(var=var.getText()+"(x_coordinate, y_coordinate)");
14 | }
```

Figure 12. Debugging Result Mapping for the `INIT` Production of the Robot Grammar

Figure 12 shows the specification of the `INIT` statement from the Robot DSL grammar. Line 13 is an addition to the base grammar that adds the results mapping information. This specific mapping is added into the DSL grammar to enable the reinterpreting of the raw GPL data returned from jdb. This mapping assists in reconstituting the DSL variable value (i.e., this specific line number indicates that the variable in the DSL is composed of two variables named `x_coordinate` and `y_coordinate`). The left side of the assignment is the DSL variable name and the right side of the assignment corresponds to the presentation format of variables at the DSL level. In order to obtain the variable value, Lines 5 and 8 indicate the commands to query the values of GPL variables `x` and `y`, and assign them to `x_coordinate` and `y_coordinate`, which are used to construct the value of the DSL variable. For example, the command to get the value of `x_coordinate` from jdb is “`print x.`” The debugger variable view retrieves the result mapping from jdb and displays the variable values at the DSL level.

### 3.3.3 Crosscutting Grammar Concerns

A crosscutting concern emerges from the addition of the explicit mapping in each of the grammar productions. For example, in Figure 12 there are many lines that are not part of the original grammar and are concerned solely with the debug mapping (lines 3, 5, 6, 8, 10, 11, 12, 13). Similar debug mapping statements in the semantic actions are repeated in *every* terminal production. The manual addition of the same mapping code in each grammar production results in much redundancy. Although the Robot DSL is simple, 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. Our previous work demonstrated how an aspect language for grammars can assist in separating the various tool-concerns for a specific grammar [61]. Using a program transformation technique, we developed an aspect-oriented language called AspectG to weave aspects into language grammars [47].

## 4 EXAMPLE GENERATION OF DSL DEBUGGERS

This section illustrates the application of the DDF on three different types of DSLs through five examples; i.e., the Robot language, FDL, BNF, SWUL, and the hybrid Robot language.

### 4.1 Generation of an Imperative DSL Debugger

This sub-section describes an imperative debugger for the Robot DSL (introduced in Section 2.2.1) that was generated by DDF from automated additions made to the base Robot grammar. The front-end of the process begins with the ANTLR generation of a lexer and parser for the Robot language. In addition to the lexer and parser, a mapping is needed to link the Robot language to the generated Java code. The mapping is specified as additional semantic actions in the Robot grammar definition. The lexer, parser, and mapping generator form the building blocks for the front-end of the DDF.

The back-end of the DDF consists of the stand-alone jdb [33] and the Eclipse debugger perspective. While adapting the architecture of the Eclipse debug platform, DDF generates an implementation of the debug model interfaces (e.g., **ILaunch**, **IDebugElement**, **IDebugTarget**, and **IBreakPoint**) to establish an Eclipse debugging perspective for the Robot DSL. The Eclipse debug model is an event-driven design that intercepts all debugging events. Most of the debugger event listeners are implemented as interfaces without an implementation – it is the responsibility of a plug-in to extend and adapt the interfaces to correspond to a specific behavior for each debugger. The Eclipse debugging perspective listens for events and uses the event information to update the user interface to show the current state of the debugged program [59].

The DDF has a debugger re-interpreter that marshals requests between the specific debug model interfaces and jdb. The debugger re-interpreter obtains a sequence of debugging commands from the **DSLDebugTarget** and queries the underlying command-line debugger (in this case, the jdb). The **DSLDebugTarget** class represents the debugging process and virtual machine, and communicates with the debugger re-interpreter. The **DSLDebugElement** interface generalizes different artifacts in a program (e.g., debug target statement, variable values, and process threads). When an end-user launches a debugging session, the user's activity is re-interpreted and sent as a **run** command to the debugger interpreter. The result returned from the jdb is stored in a variable called **resultReader**, which is re-mapped back into the debugging perspective at the DSL level. If a GPL other than Java is used (e.g., C++), the underlying GPL debugger can be changed easily to gdb. In such a case, the only adaptation needed to the DDF is a modification to the commands issued by the debugger re-interpreter. The front-end of DDF, including the implementation of **DSLDebugTarget**, would remain unchanged. The back-end of the DDF is modularized so that the concern of the debugging GUI is separated from the back-end specifics of the underlying GPL debugger.

```
1  protected void stepinto() throws DebugException {
2      Map map;
3      dslmarker = dsllinenum;
4      map = (Map) mapping.mapping.get(dsllinenum - 1);
5      sendRequest("step");
6      String functioncall = map.getFunctioncall();
7      for (int i = 1; i < mapping.size(); i++) {
8          map = (Map) mapping.mapping.get(i);
9          if (functioncall.equals(map.getFunctionname())) {
10             dsllinenum = i + 1;
11             break;
12         }
13     }
14     dsllinenum = dsllinenum + 1;
15 }
```

Figure 13. **stepinto** function in **DSLDebugTarget**

Figure 13 shows the **stepinto** method defined in the **DSLDebugTarget** class. When a debugging event (e.g., step into) is triggered by an end-user, the **DSLDebugTarget** sends a **step** command to the debugging re-interpreter through the source code mapping generated from the DSL grammar addition (line 8). The current position of the DSL line number (lines 10 and 14) and DSL function definition (line

6) are updated after the **stepinto** action is performed. Global variable **dslmarker** (line 3) keeps track of the DSL line number before the program pointer enters the function definition code segment.

The result returned from the jdb is in terms of the generated Java code, which is at the wrong abstraction level for most end-users. The variables view in the debug perspective provides a suitable place to display the variable values during the debug session. In order to display the variable values in terms of the Robot language, the variable view must map values from the Java state (as returned from the jdb) to the equivalent DSL variables in the debug perspective. 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 this Robot language example, only two variables were used in the generated code (i.e., integers **x** and **y**). Within the debugging results mapping, the **DSLDebugTarget** obtains the result from the jdb by issuing “**print x**” and “**print y**” commands to query the state of these two variables. However, the Robot DSL represents these two Java variables as a single **position** variable, which is a composition of the **x** and **y** variables in the generated Java. The **DSLDebugTarget** class re-interprets the raw data returned from jdb and reconstitutes the **position** value as the format obtained from the debug result mapping. The **position** value is then passed to the variables view of the debug perspective at the DSL level.

Figure 14 represents a screenshot of the debugging session on a Robot program. The lower-half of the figure is the Robot DSL editor, which indicates the location of the current program execution point (i.e., the highlight over the **position (+0, +1)** statement in the knight method) and the breakpoint (i.e., the bullet on the left side of the editor over the call to **down**). On the upper-right corner of the figure, the variable view for the Robot DSL indicates that the current robot position at this point in the execution is  $\langle -1, 0 \rangle$ , which is the result of moving the robot left after the initial position. The upper-left corner of the figure shows the debugging view of the Robot DSL, which includes the available debugging actions (e.g., resume, stop, step over, and step into). The debugging view also displays several properties of this debugging session (e.g., the name of this session, the current debugging function name, and the current debugging target program name).

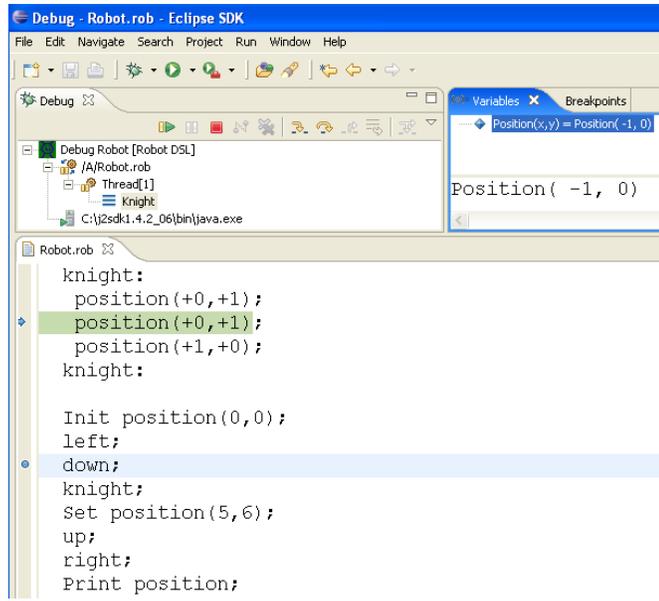


Figure 14. Screenshot of Debugging Session on Robot Language

## 4.2 Generation of a Declarative DSL Debugger

In addition to generating a debugger for an imperative DSL like the Robot language, we also used the DDF to generate a declarative DSL debugger for the FDL and BNF (described in Section 2.2.2). For different declarative DSLs the intention of the debugging activity needs to be defined because the end-user needs are more diverse when a declarative DSL is used. For example, in FDL the end-users are interested in the combination of legitimate features at a certain execution point; for BNF, end-users are interested in the snapshot (e.g., current state, current token, and current action) of the parsing iteration steps at a certain execution point. Also, the reverse mapping of variables may be different among declarative DSLs. In the case of FDL, the debugging actions (e.g., step into, step over, and set a breakpoint) are the same as an imperative DSL and the only modification to DDF was isolated in the component that mapped the variable results back into the Eclipse debugging perspective. In other declarative DSLs, the variable mapping from the GPL to DSL may be different from the imperative DSL case (e.g., in the Robot imperative DSL, the position variable is mapped to two integers x and y in Java; in the declarative FDL, features are mapped into feature specification objects in Java).

### 4.2.1 FDL Debugger

In the particular instance of the FDL debugger, the variables view in the debug perspective must represent all of the features at any point in time within the execution state. Considering as an example the **Car** specification from Figure 4, stepping over each feature causes the resulting configuration to expand or contract (see the expanded variable view of the configuration in Figure 15). The Eclipse debugging variable view is not able to display a Java object directly. A function called, `getVariableValue()`, in the DDF is used to retrieve the attributes from the object and translate the object into a primitive **String** type that can be displayed in the DSL variables view. The source code mapping information is passed to the retrieve method so that the GPL variables are matched to variables in an FDL feature list. This specific variable remapping (`var= var.getText()+remap(feature);`) is added into the DSL grammar to enable the reinterpretation of the raw GPL data returned from jdb. This mapping assists in reconstituting the DSL variable value. The function `remap()` is used to sanitize the raw debugging results from jdb so that the well-formatted string can be displayed in the Eclipse debugging variable view. The left side of the assignment is the DSL variable name (i.e., **carbody**, **Transmission**, **Engine**, and **Horsepower**) and the right side of the assignment corresponds to the presentation format of variables at the DSL level. In order to obtain the value of the variable (i.e., feature), (`feature="print var"+listnumber+".toString()";`) is needed to query the values of Java variable **feature**, and assigned to **var**, which is used to construct the value of the DSL variable.

Figure 15 is a screenshot of a debugging session on the **Car** specification introduced in Figure 4. The lower-half of the figure contains the FDL editor, which indicates the breakpoints and current feature rule under evaluation. The upper-right of the figure contains the variable view for the **Car** program, which shows the current state of the feature configuration at the point of evaluating the **Horsepower** feature. Figure 15 captures the instance of the **Car** feature diagram after evaluating three feature rules (e.g., **Car**, **Transmission**, and **Engine**) where the program pointer stops. In this example, the

program execution point has stepped over two features (e.g., **Transmission** and **Engine**) from the location of the original breakpoint (e.g., **Transmission** feature). The detailed view enables an end-user to see all of the rule combinations that contribute to the current state of a particular configuration. The rest of the debugging perspective (e.g., stepping over rules) is similar to the Robot DSL debugger in Section 4.1.

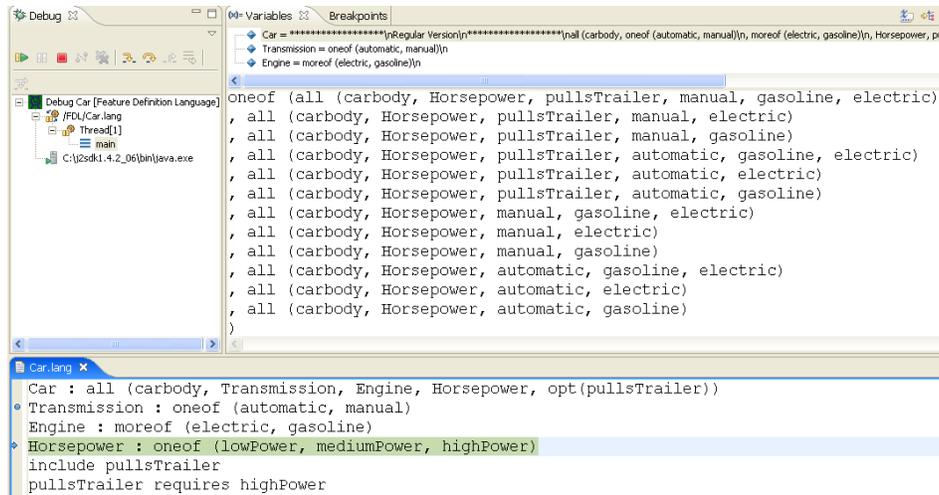


Figure 15. Screenshot of Debugging Session on Car Program

#### 4.2.2 BNF Debugger

In addition to generating a debugger for a declarative DSL like the FDL, we also used the DDF to generate a declarative DSL debugger for the Robot BNF (described in Section 2.2.2).

The Language Implementation System based on Attribute Grammars (LISA) [42] is a system to generate a parser, compiler, interpreter, and other language-based tools (e.g., finite state automata, visualization editor) based on the language specification. The example in this section uses LISA to generate the parser for the Robot BNF. LISA follows the standard BNF notation for defining the syntax of the Robot language. Considering the simple Robot BNF as an example, stepping over a non-terminal causes the generated Robot language parser to iterate through the input tokens by looking up the parsing control table and taking appropriate actions (e.g., shift, reduce, accept, and signal an error). This generated parser code is implemented using the classic table-driven LR(1) parsing algorithm described in

[1]. Compared to previous case studies, the more complex table-driven data structure is generated in Java that is equivalent to the BNF. A parsing control table is pre-constructed based on the grammar specification of the Robot language. LISA generates Java code from this description. The grammar in BNF is mapped to the Action and Goto functions of an LR-parsing table. The complete table contents and video demos are available at [23].

In order to generate a debugger for BNF, the modifications to DDF were isolated in the component that mapped the variable results back into the Eclipse debugging perspective and step over debugging action, which are different from the FDL case. For different problem domains, the end-users have varying interests while debugging the different DSLs. The language design experts are interested in the snapshot of each parsing step based on the parsing control table. In the grammar debugger, the variables view in the debug perspective must represent the current parsing state, current input token, current parsing action, current stack, and current input token position. Even within the same DSL, different users may want to turn off the level of detail reported in the variable view (e.g., in the BNF DSL, the user can turn on and off the detailed view of specific variables). Considering as an example the Robot language specification from Figure 5, stepping over the **COMMANDS** production causes the current status of the parsing process to execute the **COMMANDS** definition. The modified Eclipse debugging variable view is able to display the current parsing state, current input token, current action, current stack, and current input token position. All of this information is retrieved from the jdb by the generated parser. A function in the DDF is used to retrieve the information from the parsing control table object (in Java) and translate the object's attributes into a primitive String type that can be displayed and understood by language design experts through the DSL variables view.

Figure 16 is a screenshot of a debugging session on the Robot BNF. The lower-left half of the figure contains the grammar editor, which also indicates the breakpoints and current program pointer. The lower-right half of the figure contains the input language editor which contains a sample Robot language program. The upper-right of the figure contains the variable view of the BNF debugger, which

shows the current status of parsing. In Figure 16, the grammar program execution point has stopped at Line 3 where the breakpoint is set. After the step over debugging action, the current program pointer moves to Line 4 from Line 3, which indicates that the **COMMANDS** non-terminal has been executed. An end-user can click the **CurrentOperation** variable in the variable view and a detailed expansion view is provided (e.g., the current parse action is **shift10**, which means the parser will shift the number 10 state into the current parsing stack). In this case, the current parsing state is 4; current token is “end” located at row 4 and column 1 in the input editor; current parsing state stack contains 0, 2, and 4. The detailed view shows variable values that may be of interest that can help language designers to assess the parsing process according to the language syntax specification in BNF. The rest of the debugging perspective (e.g., step into rule) allows language designers to obtain the parsing status of the non-terminal definition that was stepped into. Another version of a BNF debugger can hide the bottom-up parser implementation details from end-users who are not familiar with this parsing technique. In such case, the debugging perspective only shows the current token, ConsumedTokensTrace, current token location, and ProductionTrace. The detailed video demonstrations can be found at the project website [23].

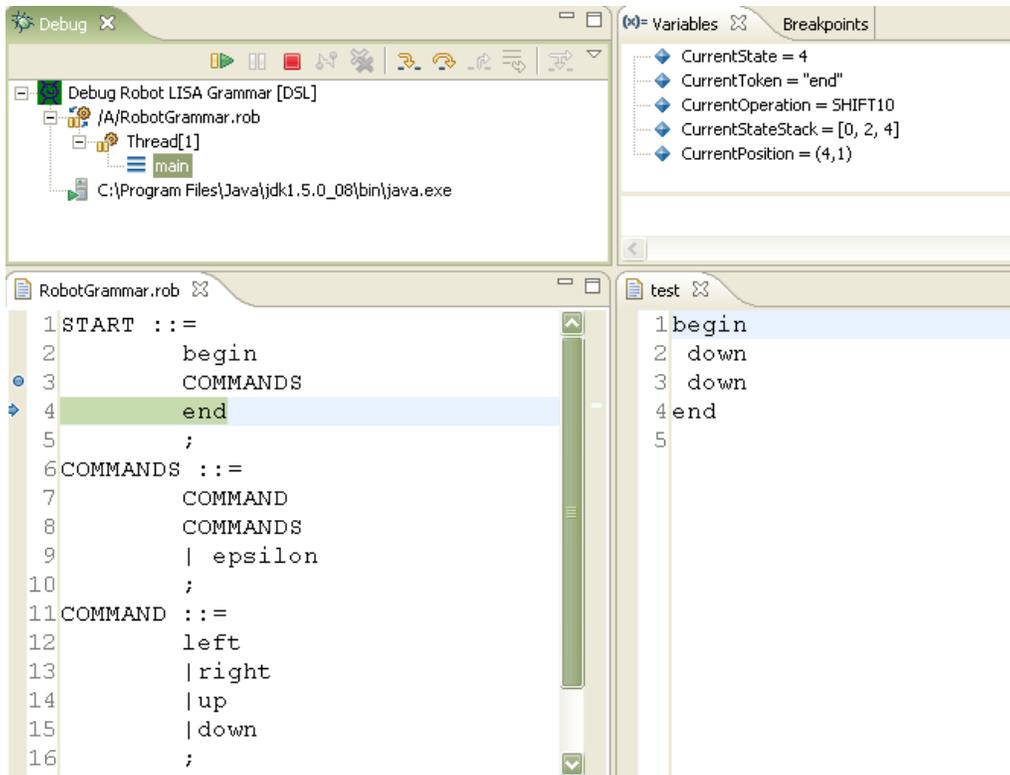


Figure 16. Screenshot of Debugging Session on Robot BNF

### 4.3 Generation of a Hybrid DSL Debugger

This section uses the SWUL and Robot hybrid DSLs (as described in Section 2.2.3) to illustrate the generation process of hybrid DSL debuggers from DSL grammars.

#### 4.3.1 SWUL Debugger

In the SWUL debugger case, we use the prefix “dsl” in the function type to distinguish the embedded SWUL code surrounding the Java statements. For example, the mapping: `{9, "WelcomeSwing.java", 22, 24, "main", "label", "dslfunctionbody"}`, means that line 9 is a DSL statement that is mapped to one segment of code (e.g., from line 22 to 24) of the generated Java code in WelcomeSwing.java. The function type is also used to determine the mode in which the variables will be displayed (i.e., the DDF will switch between a Java variable view and a SWUL variable view depending on the function type of the currently executed line of code). If the function type of a specific line of the SWUL program is not prefixed with “dsl,” the DDF variable view

will show the Java variable values (i.e., with the “dsl” prefix, the DSL variable values will be shown). The variable name is also determined by the sixth field of this mapping information (e.g., label).

Within a hybrid debugger, the debug perspective must be able to display both DSL variables and Java core variables based on the current mode of the debugger. In jdb, the **locals** command is used to display the values of the local Java variables for the current stack frame. In the debugger interpreter, a method called **debug\_locals** directly sends the **locals** command to the jdb. The **DSLDebugTarget** method within DDF obtains the debug result of the **locals** command and displays the Java variables in the debug perspective.

In order to generate the hybrid SWUL DSL debugger, a modification was needed to the **Step Over** action (i.e., an addition was made to the **step** method discussed in Section 4.1). When debugging the Java part of a hybrid DSL (e.g., the code in Figure 17 concerned with packing the frame, line 19), there is a one-to-one correspondence between the code in the DSL and the generated GPL. In this mode, there is no mismatch between the DSL code and the generated GPL (Java) code. In such cases, the **step** method was modified in the case of a hybrid DSL by setting the debug actions in the DSL (e.g., set a breakpoint, step over/into) to correspond to the same line of code in the GPL.

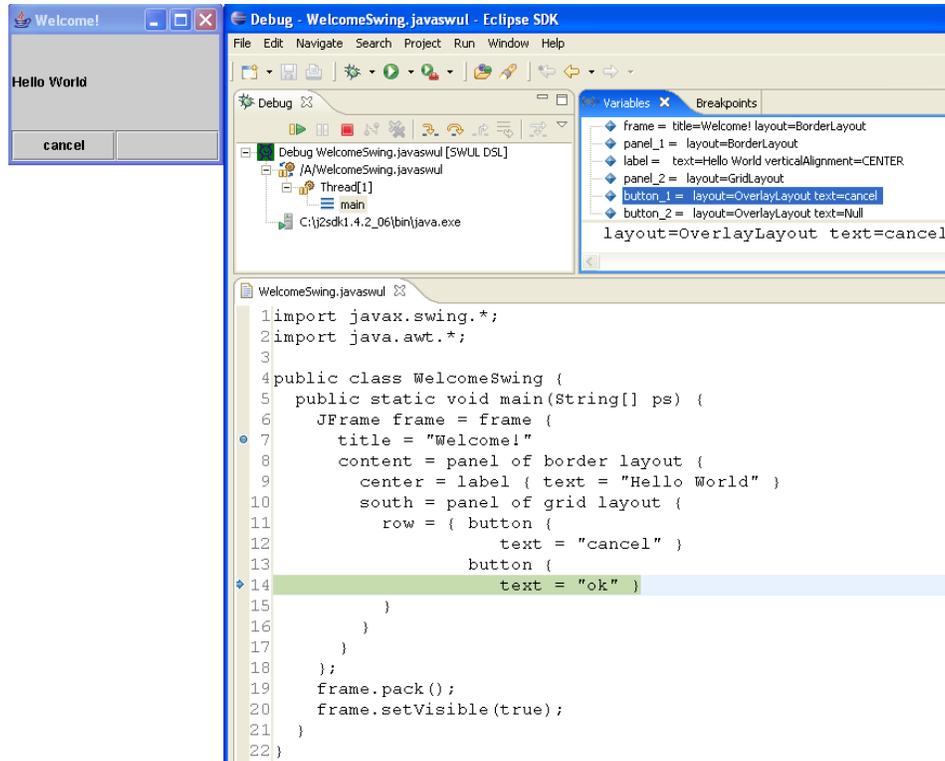


Figure 17. Screenshot of Debugging Session on SWUL Program

Figure 17 is a screenshot of the debugging session for a hybrid SWUL program that uses an escape block to obtain structural description of user interfaces. As the program pointer steps through the embedded DSL code in the SWUL program, the user interface Swing graphic representation window on the upper-left corner evolves according to the current program execution state. In the variables view of the debugging perspective (shown in the upper-right of Figure 17), the individual components (e.g., frame, label, and button) and their associated attributes (e.g., location and contents) are displayed.

#### 4.3.2 Robot Hybrid Debugger

We also use the Robot hybrid DSL (as described in Section 2.2.3) to illustrate the generation process of a hybrid DSL debugger from the DSL grammar. We use the prefix “gpl” in the function type to distinguish the embedded Java code surrounding the Robot DSL statements. For example, the mapping: `{10, "Robot.java", 20, 20, "Random", "gplfunctionbody"}`, means the functionbody type on line 10 of the Java code in the Robot language is mapped to one single line (e.g.,

line 20) of Robot.java. The function type is also used to determine the mode in which the variables will be displayed (i.e., the DDF will switch between a Java variable view and a DSL variable view depending on the function type of the currently executed line of code).

In order to display both DSL variables and Java core variables based on the current mode of the debugger, the `getVariableValue()` method within DDF was extended to handle the `jdb locals` command to retrieve all the local variables of a running Java program. To generate the hybrid Robot DSL debugger, an extension was needed to the **Step Over** action (i.e., an addition was made to the step method discussed in Section 3.2). When debugging the Java part of a hybrid DSL (e.g., the code in Figure 18 concerned with generating a random position), there is a one-to-one correspondence between the code in the Robot hybrid program and the generated Java program. In summary, when debugging the Java part of the hybrid DSL, the `jdb`'s debugging results are passed back to the debug perspective as a one-to-one mapping.

Figure 18 is a screenshot of the debugging session for a hybrid Robot DSL program that uses an escape block to obtain random coordinates for the robot position. As the program pointer steps through the embedded Java code in the Robot DSL program, the input dialog window asks the user to enter the range of values for the random number generator. In the variable view of the debugging perspective (shown in the top-right of Figure 18), there are two sets of views (e.g., Robot variables and Java variables) available to the user. At this point in the debugging session, the two abstraction levels complement each other to provide the user with more precise information about the execution behavior of the hybrid DSL program.

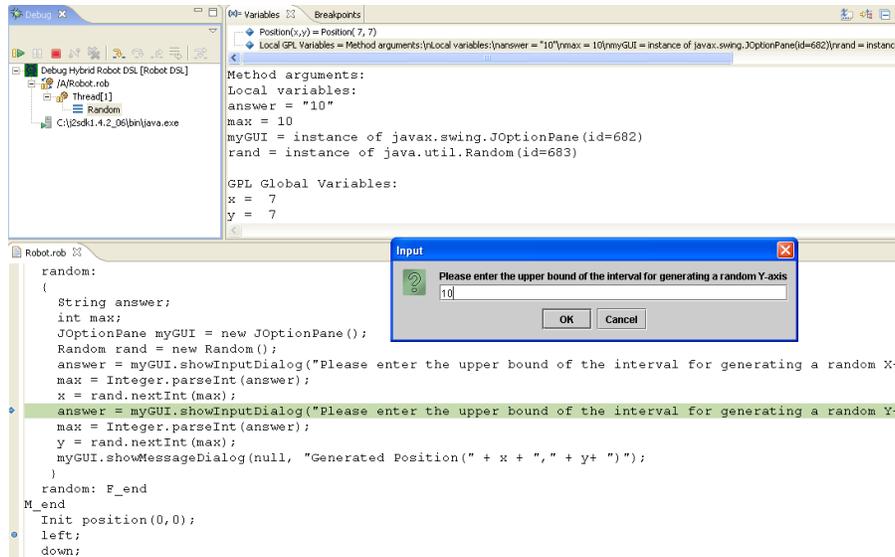


Figure 18. Screenshot of Debugging Session on Hybrid Robot Program

#### 4.4 Case Study Evaluation

Various styles of DSLs served as test cases to determine the generality of our grammar-driven approach by comparing the generation of different DSL debuggers. A total of five different debuggers for three types of DSLs were generated using DDF. Several quantitative measurements were observed to analyze the amount of effort required to generate new debuggers. In particular, this section addresses the question, “How many of the generic software components from DDF can be reused without modification or small changes, as compared between different types of DSLs?” To address the level of effort required to adapt a debugger, we measured the amount of code that was written for each new debugger. Within DDF there are 19 software components. Among these components, there are 3,429 lines of code that are generalized and reused in all of the debuggers. On average, through our use of AspectG in modifying grammars to hook into the DDF, we have observed that less than 150 lines of code are needed for each new debugger generation. In Figure 19, column 5 shows the number of additional statements that need to be added into the original DSL grammars for each of the five different case studies. The comparison presented in Figure 19 indicates that the amount of code needed to generate a DSL debugger is relatively small when using DDF.

<i>DSL Category</i>	<i>DSL Name</i>	<i>Number of Specific Methods</i>	<i>Customized Lines of Code</i>	<i>Additional Statements in Modified Grammars</i>
<i>Imperative DSL</i>	Robot Language	2	69	97
<i>Declarative DSL</i>	FDL	3	89	22
	BNF	4	261	31
<i>Hybrid DSL</i>	Hybrid Robot Language	4	117	117
	SWUL	5	152	66

Figure 19. Generality Analysis on DDF

## 5 RELATED WORK

This section provides an overview of related work in the area of DSL debuggers.

### 5.1 Khepera

Khepera is a toolkit for the rapid implementation and long-term maintenance of DSLs [25]. The Khepera system provides program transformation functions that can translate from one high-level language to another. Khepera provides debug tracking information transparently and supports transformation replay and navigation, as well as debugger queries. The research application of Khepera developed by Faith emphasizes debugging support to optimize translated code (i.e., one-to-many, many-to-one, and many-to-many source-to-source transformation), which focuses on the optimized target code. This is different from the assumption of our work because the source-to-source translation is restricted to one-to-many transformation between a DSL and a corresponding GPL. When dealing with composed transformations, Khepera stores every step of the transformation information (including translation and optimization) into a database. Faith also developed several algorithms (e.g., a tuple logging algorithm) to track changes to the AST throughout the transformation process [26]. With the assistance of the rich transformation information provided in the database, the transformation can be replayed. However, the execution time is dramatically hampered, which increases the overall cost of building a DSL debugger using Khepera.

Because the approach adopted by Khepera provides a rich database of transformation information, it may not scale to larger programs due to performance issues. Optimization in a DSL implementation is not considered a necessary step in many cases because the optimization is a complicated and time-consuming task compared to the other parts of language implementation. Our DDF provides DSL developers a framework to construct a debugger for a DSL without going through the type of transformation algorithms and database used by Khepera. Because we adopt a pre-processor piggyback approach, much of the optimization in our framework is passed on to the compiler for the targeted GPL.

## **5.2 JSR-045**

JSR-045 represents the “Java Specification Request for Debugging Support for Other Languages.” This specification establishes standardized debugging tools for correlating Java bytecode to the source code of languages other than Java [35]. Although JSR-045 has a source code line number mapping mechanism similar to our approach, JSR-045 does not have a debugging result mapping process that maps the variables in Java back to DSL variables. When using an implementation of the JSR-045, the variables are all in the context of Java, rather than in the domain of the end-user perspective represented by the DSL. Furthermore, the JSR-045 mapping is not rich enough to capture the special attributes of each DSL line statement (e.g., function name and function type). Another limitation of the JSR-045 is that it only handles Java Virtual Machine byte code as its target language. JSR-045 expresses the debugging support to DSLs (e.g., JSP and SQLJ) which have to be translated into Java Virtual Machine byte code. Therefore, JSR-045 is tied to the Java Platform Debugger Architecture (JPDA) [34]. Because JSR-045 is Java bytecode-specific, it cannot be adapted to other GPLs (e.g., C++ and Fortran). Our approach simply uses the available GPL command-line debuggers (e.g., jdb [33]), which can be replaced easily by other command-line debuggers (e.g., gdb) if the target GPL changes.

## **5.3 ANTLR Studio**

ANTLR Studio is an Eclipse plug-in for ANTLR [3]. It provides ANTLR language developers a grammar editor with syntax highlighting, auto indenting, and syntax auto completion functions. ANTLR Studio

uses JSR-045 to implement its debugger functionality. As the execution pointer moves through each production rule, the “Text Consumed View” displays the recently parsed text, and the “AST View” displays the AST tree of the recently parsed text characters. The user can step into the semantic action of the grammar, which is written in Java. As such, ANTLR Studio is an example of a hybrid debugger. While stepping through a semantic action, the variable view displays the current state of all Java variables in the semantic code. A debugging action called “Step to Next Rule” is created for the specific syntax and semantic meaning of an ANTLR grammar so that users can step to the next token in situations when a single grammar rule consists of several tokens. The variables displayed in the variable view of ANTLR Studio are in the Java context. In comparison, debuggers created with the DDF are able to display the variables in the context of the end-users domain. ANTLR Studio appears to be Java bytecode-specific because of its dependence on JSR-045. It also is tied to ANTLR and does not offer a capability to debug grammars in other notations (e.g., Yacc). ANTLR Studio has many rich debugging features that solely target ANTLR grammar specifications; as such, it offers richer functionality than could be offered by a DDF-generated debugger. However, the DDF is more generic and adaptable to different DSLs as its target language. The tradeoff is richer functionality for a specific debugging context, versus opportunities for retargeting a debugger to a different context.

## **5.4 TIDE**

Olivier designed the ToolBus Integrated Debugging Environment (TIDE) [43], which is implemented within the ASF+SDF Meta-Environment. Using TIDE, an additional framework has to be constructed to provide an interactive debugger for the user [10], [11]. ToolBus represents middleware written in C to enable component-based communication among tool environments [16]. TIDE provides an interface during the execution of a program defined in ASF+SDF. Although TIDE claims to reduce the lines of code for implementing a full-fledged debugger, knowledge of the rewrite rules in the built-in library are required to construct a debugger for a new language. Our work adopts standard compiler implementation methods (e.g., lexer and parser) as the front-end of the DDF. Although TIDE uses gdb, in order to use

TIDE it is required to understand a precise language specification in ASF+SDF. For example, debugging events can happen at certain program locations, which have to be considered very thoroughly before such events are inserted into the ASF+SDF specification. DDF handles this issue differently by providing debugging knowledge through the property of each mapping location (i.e., it dynamically decides the appropriate debugging actions that can happen at each program location based on the user debugging actions).

## 6 CONCLUSION

As the number of end-user programmers rise substantially, the need for a full suite of development tools appropriate for the end-user's domain is increasingly justified. One of the limitations of end-user programming is the lack of debuggers to assist in the identification of errors. As computer software applications increase in number, cost, size, and complexity, the capabilities offered by debugging tools assume more significance because of greater economic risk [30]. DSLs are becoming more prevalent in general development and assist end-users in describing the essence of a problem in their domain. Although research on debugging GPLs has been investigated in depth over the past decades, the topic of debugging DSLs has been neglected. The goal of DDF is to provide a framework for debugger generation that will assist domain-experts and end-users in debugging DSL programs at an abstraction level that is familiar to them.

Developing DSL debuggers from scratch is a very expensive and demanding task. The DDF offers a generalized approach where DSL debuggers can be generated automatically with minimal additional effort by reusing existing GPL debuggers and IDEs. Our approach can be applied if DSL programs are translated to GPL programs (e.g., using compiler, extensible compiler, or preprocessor DSL implementation pattern), which provides an opportunity to reuse existing GPL debuggers and IDEs in order to debug DSL programs. The idea is to use GPL debugger commands and re-interpret them in the context of a DSL. The initial step of creating a debugger using the DDF involves a description of how the DSL source code is mapped to GPL code. This requires an existing DSL compiler/interpreter that is

specified in a language specification such as ANTLR. Section 3.1 presented the details of the DDF source code mapping, which is DSL-independent.

Re-interpretation of GPL debugging commands requires that each command (e.g., set a breakpoint, step into, and step over) be re-implemented for each specific DSL. In most cases, the algorithms presented in this paper are generalized sufficiently to be used with several particular patterns of different DSL categories. However, for each new DSL this step needs to be examined in consideration of specific intentions provided by the DSL. The implementation of a new DSL debugger requires several issues to be considered, such as, “What does it mean to set a breakpoint in this DSL, or to step over a line of DSL code?” Examples of the type of mapping needed to address such questions were offered in Section 3.2. A final step in the construction of a new DSL debugger is concerned with the description of a reverse mapping of the GPL debugging server results back into the DSL variable context. This step, called debugging results mapping, is described in Section 3.3. In many cases, we were able to reuse existing DSL compiler implementations to implement this mapping.

Our work makes a contribution in the area of Grammarware [37] by impacting the status of grammars, grammar transformations, and their relationship to tool plug-ins. The resulting contribution advances the capabilities of domain experts and end-user programmers by providing an adequate tool base for a software development lifecycle based on DSLs. This paper also demonstrated the potential for reusing existing GPL language tools through grammar-driven automation. We believe that automated software engineering applied to the adaptation of existing IDE interfaces will become a future trend of tool construction. A key enabler of such automation will be the application of aspect-oriented concepts to support a new generative approach for language tool construction [61].

## **6.1 Future Work**

This research has several implications for future work. In addition to DDF, as noted in Figure 2 the future research will investigate the generation of profilers from the DSL grammar. The same architecture depicted in Figure 7 applies to the DSL profiler structure (i.e., the DDF’s GPL debugging server will be

replaced by the GPL profiler, which monitors the run-time execution environment). The future research will use JFluid [22] as the underlying GPL profile server, which is a profiler for programs written in Java.

This paper focused on the situation when one line of DSL code is generated into a set of *consecutive* lines of code in a GPL. There is a corresponding problem that occurs when one line of DSL code is generated into *non-consecutive* lines of code to various locations in a GPL. This may occur if a DSL is implemented using a rewrite rule transformation system, where local-to-global transformation tangles the implementation [15]. We plan to continue the investigation of DDF to handle most of the DSL to GPL generation scenarios, including the non-consecutive case.

We are also interested in furthering this work to other IDEs other than Eclipse. For example, a future work will explore the requirements for providing the same set of DSL tools in Microsoft Visual Studio [48]. Adaptation of the DDF to IDEs other than Eclipse will offer a new research objective that seeks to generalize both the tool-independent and language-independent nature of the framework.

## ACKNOWLEDGMENT

This work was supported in part by an IBM Eclipse Innovation Grant (EIG). We thank the reviewers for their very helpful comments that aided us in preparing a revision of this paper.

## REFERENCES

- [1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Pearson/Addison-Wesley, 2007.
- [2] ANTLR - ANother Tool for Language Recognition, available at <http://www.antlr.org/>
- [3] ANTLR Studio, available at <http://www.antlrstudio.com>
- [4] P.J. Ashenden, *The Designer's Guide to VHDL*, Morgan-Kaufmann, 2002.
- [5] D. Atkins, T. Ball, G. Bruns, and K. Cox, "Mawl: A Domain-Specific Language for Form-Based Services," *IEEE Transactions on Software Engineering*, vol. 25, no. 3, pp. 334-346, May/June 1999.
- [6] M. Auguston, "Program Behavior Model Based on Event Grammar and its Application for Debugging Automation," *AADEBUG Workshop on Automated and Algorithmic Debugging*, Saint-Malo, France, pp. 277-291, May 1995.

- [7] M. Auguston, "Building Program Behavior Models," *ECAI Workshop on Spatial and Temporal Reasoning*, Brighton, England, pp. 19-26, August 1998.
- [8] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 355-371, June 2004.
- [9] J. Bentley, "Little Languages," *Communications of the ACM*, vol. 29, no. 8, pp. 711-721, August 1986.
- [10] M. van den Brand, J. Heering, P. Klint, and P. Olivier, "Compiling Language Definitions: The ASF+SDF Compiler," *ACM Transactions on Programming Languages and Systems*, vol. 24, no. 4, pp. 334-368, July 2002.
- [11] M. van den Brand, B. Cornelissen, P. Olivier, and J. Vinju, "TIDE: A Generic Debugging Framework," *Tool Demonstration at Fifth Workshop on Language Descriptions, Tools, and Applications*, Edinburgh, Scotland, June 2005.
- [12] M. Bravenboer and E. Visser, "Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions," *Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, pp. 365-383, October 2004.
- [13] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace, "End-User Software Engineering with Assertions in the Spreadsheet Paradigm," *International Conference on Software Engineering*, Portland, OR, pp. 93-105, May 2003.
- [14] M. Burnett, C. Cook, and G. Rothermel, "End-User Software Engineering," *Communications of the ACM*, vol. 48, no. 9, pp. 53-58, September 2005.
- [15] T. Cleenewerck and T. D'Hondt, "Disentangling the Implementation of Local-to-Global Transformations in a Rewrite Rule Transformation System," *Symposium for Applied Computing (SAC) – Programming Languages Track*, Santa Fe, NM, pp. 1398-1403, March 2005.
- [16] B. Cornelissen, *Using TIDE to Debug ASF+SDF on Multiple Levels*, Masters Thesis, Universiteit van Amsterdam, available at <http://www.cwi.nl/themes/sen1/twiki/pub/SEN1/TIDE/bas-scriptie.pdf>, December 2004.
- [17] CUP User Manual, available at <http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html>
- [18] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Techniques, and Applications*, Addison-Wesley, 2000.
- [19] A. van Deursen and P. Klint, "Little Languages: Little Maintenance?" *Journal of Software Maintenance*, vol. 10, no. 2, pp. 75-92, March 1998.

- [20] 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.
- [21] A. van Deursen and P. Klint, "Domain-Specific Language Design Requires Feature Descriptions," *Journal of Computing and Information Technology*, vol. 10, no. 1, pp. 1-17, 2002.
- [22] M. Dmitriev, "Design of JFluid: A Profiling Technology and Tool Based on Dynamic Bytecode Instrumentation," Sun Microsystems Technical Report, Mountain View, CA, available at <http://research.sun.com>, 2004.
- [23] Domain-Specific Language Debugging Framework , <http://www.cis.uab.edu/wuh/DDF>
- [24] Eclipse, available at <http://www.eclipse.org>
- [25] R. Faith, L. S. Nyland, and J. F. Prins, "Khepera: A System for Rapid Implementation of Domain-Specific Languages," *USENIX Conference on Domain-Specific Languages*, Berkeley, CA, pp. 243-255, October 1997.
- [26] R. Faith, *Debugging Programs after Structure-Changing Transformation*, Ph.D. Dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, available at <http://www.cs.unc.edu/~faith/faith-dissertation-1997.ps>, 1998.
- [27] R. Floyd, "The Paradigms of Programming," *Communications of the ACM*, vol. 22, no. 8, pp. 455-460, August 1979.
- [28] E. Gamma and K. Beck, *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*, Addison-Wesley, 2003.
- [29] gdb - The GNU Project Debugger, available at <http://sourceware.org/gdb/>.
- [30] D. Gelperin and B. Hetzel, "The Growth of Software Testing," *Communications of the ACM*, vol. 31, no. 6, pp. 687-695, June 1988.
- [31] J. Groff and P. Weinberg, *SQL: The Complete Reference*, McGraw-Hill, 2002.
- [32] P. Henriques, M. Pereira, M. Mernik, M. Lenič, J. Gray, and H. Wu, "Automatic Generation of Language-based Tools using LISA," *IEE Proceedings - Software*, vol. 152, no. 2, pp. 54-69, April 2005.
- [33] jdb - The Java Debugger, available at <http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jdb.html>
- [34] The Java Platform Debugger Architecture, available at <http://java.sun.com/products/jpda/index.jsp>
- [35] JSR 45: Debugging Support for Other Languages, available at <http://www.jcp.org/en/jsr/detail?id=045>

- [36] B. R. Kiebertz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith, and L. Walton, "A Software Engineering Experiment in Software Component Generation," *International Conference on Software Engineering*, Berlin, Germany, pp. 542-552, March 1996.
- [37] 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.
- [38] T. Kosar, P.E. Martínez-López, P. A. Barrientos, and M., *A Preliminary Study on Various Implementation Approaches of Domain-Specific Languages*, *Journal of Information and Software Technology*, to appear, 2007, doi:10.1016/j.infsof.2007.04.002.
- [39] The Lex & Yacc Page, available at <http://dinosaur.compilertools.net/>
- [40] M. Loy, R. Eckstein, D. Wood, J. Elliott, and B. Cole, *Java Swing*, O'Reilly, 2002.
- [41] 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.
- [42] M. Mernik and V. Žumer, "Incremental Programming Language Development," *Computer Languages, Systems and Structures*, no. 31, pp. 1-16, 2005.
- [43] P. Olivier, *A Framework for Debugging Heterogeneous Applications*, Ph.D. Dissertation, Universiteit van Amsterdam, available at <http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/TIDE>, 2000.
- [44] A. Oram and S. Talbot, *Managing Projects with make*, O'Reilly, 1991.
- [45] D. Parnas and D. P. Siewiorek, "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems," *Communications of the ACM*, vol. 18, no. 7, pp. 401-408, July 1975.
- [46] T. Parr, *The Definitive ANTLR Reference Building Domain-Specific Languages*, Pragmatic Bookshelf, 2007.
- [47] D. Rebernak, M. Mernik, H. Wu, and J. Gray, "Domain-Specific Aspect Languages for Modularizing Crosscutting Concerns in Grammars," *GPCE Workshop on Domain-Specific Aspect Languages*, Portland, OR, October 2006.
- [48] J. Robbins, *Debugging Applications for Microsoft.Net and Microsoft Windows*, Microsoft Press, 2003.
- [49] J. B. Rosenberg, *How Debuggers Work: Algorithms, Data Structures, and Architecture*, John Wiley and Sons, 1996.

- [50] S. Ryu and N. Ramsey, "Source-level Debugging for Multiple Languages with Modest Programming Effort," *International Conference on Compiler Construction*, Edinburgh, Scotland, pp. 10-26, April 2005.
- [51] C. Scaffidi, M. Shaw, and B. Myers, "Estimating the Numbers of End-Users and End-User Programmers," *Symposium on Visual Languages and Human-Centric Computing*, Dallas, TX, pp. 207-214, September 2005.
- [52] R. W. Sebesta, *Concepts of Programming Languages*, Addison-Wesley, 2007.
- [53] C. Simonyi, M. Christerson, and S. Clifford, "Intentional Software," *Object-Oriented Programming, Systems, Languages, and Applications (Onward Track)*, Portland, OR, pp. 451-464, October 2006.
- [54] D. Spinellis, "Notable Design Patterns for Domain-Specific Languages," *Journal of Systems and Software*, vol. 56, no. 1, pp. 91-99, February 2001.
- [55] A. Sutcliffe and N. Mehandjiev, "End-User Development: Tools that Empower Users to Create their Own Software Solutions," *Communications of the ACM*, vol. 47, no. 9, pp. 31-32, September 2004.
- [56] S. A. Thibault, R. Marlet, and C. Consel, "Domain-Specific Languages: From Design to Implementation Application to Video Device Drivers Generation," *IEEE Transactions on Software Engineering*, vol. 25, no. 3, pp. 363-377, May/June 1999.
- [57] D. S. Wile, "Lessons Learned from Real DSL Experiments," *Science of Computer Programming*, vol. 51, no. 3, pp. 265-290, June 2004.
- [58] D. S. Wile and J. C. Ramming, "Guest Editorial: Introduction to the Special Section 'Domain-Specific Languages (DSLs)'," *IEEE Transaction on Software Engineering*, vol. 25, no.3, pp. 289-290, May/June 1999.
- [59] 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.
- [60] H. Wu, J. Gray, and M. Mernik, "Debugging Domain-Specific Languages in Eclipse," *OOPSLA Eclipse Technology Exchange Poster Session*, Vancouver, Canada, October 2004, available at <http://www.cis.uab.edu/gray/Pubs/etx-2004.pdf>.
- [61] 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.

- [62] E. Van Wyk and E. Johnson “Extensible Compilers and Modular Language Extensions for Computational Geometry,” *The 40th Hawaii International Conference on Systems Science*, Big Island, HI, January 2007.
- [63] P. T. Zellweger, *Interactive Source-Level Debugging of Optimized Programs*, Ph.D. Dissertation, published as Technical report CSL-84-5, Department of Computer Science, University of California, Berkeley, May 1984.