GRAMMAR-DRIVEN GENERATION OF DOMAIN-SPECIFIC LANGUAGE TESTING TOOLS USING ASPECTS

by

HUI WU

JEFF GRAY, COMMITTEE CHAIR
BARRETT R. BRYANT
MARJAN MERNIK
MIKHAIL AUGUSTON
CHENGCUI ZHANG
BRIAN TOONE

A DISSERTATION

Submitted to the graduate faculty of The University of Alabama at Birmingham,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

BIRMINGHAM, ALABAMA

2007

GRAMMAR-DRIVEN GENERATION OF DOMAIN-SPECIFIC LANGUAGE TESTING TOOLS USING ASPECTS

HUI WU

COMPUTER AND INFORMATION SCIENCES

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. Testing 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 and unit testing a program written in a DSL is often non-existent. The lack of a debugger and unit test engine at the proper abstraction level limits an end-user's ability to discover and locate faults in a DSL program. This dissertation describes a grammar-driven technique to build a debugging and unit testing tool generation framework by adaptations to existing DSL grammars. This approach leverages existing GPL testing tools to indirectly exercise the end-user's debug and test intentions at the DSL level. The adaptations to DSL grammars represent the hooks needed to interface with a supporting infrastructure constructed for an Integrated Development Environment (IDE) that assists in debugging and unit testing a program written in a DSL. The contribution represents a coordinated approach to bring essential software tools (e.g., debuggers and test engines) 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 testing tools. During the testing tool generation process, crosscutting concerns

were observed in representations of DSL grammars. To address these particular crosscutting concerns, an investigation into the principles of aspect-oriented programming applied to grammars has been conducted. A domain-specific aspect language, called AspectG, has been designed and implemented, which is focused within the domain of language specification. This dissertation outlines the challenges and issues that exist when designing aspect languages that assist in modularizing crosscutting concerns in grammars. The research described in the dissertation 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.

DEDICATION

This work is dedicated to my wife, Chenxia – It is your love that keeps me going, and it is your sacrifice that makes this and all things possible

My grandmother – It is your wish and prayer that made my dream come true

My parents, Wenbin, Farong, and Xiaojiang, and my brother, Bo –

It is your love that has helped me grow

Mei Li and Selby Moody, for their unconditional love and support.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Dr. Jeff Gray, who gave me numerous opportunities, profound erudition, inspirational advice, and prepared me with knowledge, courage, and endurance to fight through one of the most challenging periods of my life, yet one of my most memorable times as well. I am grateful for the encouragement and compassion that he offered in leading me along the right track, letting me experience new frontiers, and allowing me to complete this work. His perspectives on teaching and research have been inspirational. I thank him for the trust and confidence that he has had in me.

I would like to genuinely thank Dr. Marjan Mernik, whose collaborations with UAB were a starting and turning point of this research work. I am thankful for his expertise, advice, consistency, and support. As one of my committee members, he was central in directing me to the area of Domain-Specific Languages. His academic research attitude and relationships with fellow students and researchers are models that I shall follow.

I also appreciate the contributions of my other committee members. Dr. Barrett Bryant helped my understanding of programming languages and taught me the skills needed to complete this work. My discussions with Dr. Mikhail Auguston helped me to look at the issues of debugger implementation from a totally different angle, which made

it more challenging and interesting. Dr. Chengcui Zhang and Dr. Brian Toone, I appreciate your great discussions on my work on various occasions.

To my fellow comrades, Fei Cao, Wei Zhao, Song Zhou, Yuehua Lin, Shih-hsi Liu, Suman Roychoudhury, Faizan Javed, Robert Tairas, Yonghui Chen, Ying Sun, Francisco Hernandez, Ying Liu, Zhijie Guan, Xin Chen, Haisong Li, Jing Zhang, and Carl Wu, I cherish our friendship and wonderful time together. Thank you for your help, encouragement, and support during the past years, especially during the difficult times when I had doubts about my future!

This dissertation research would not have been completed had it not been for the support from the Computer and Information Sciences Department. Mrs. Janet Tatum, Mrs. Kathy Baier, Mr. Fran Fabrizio, Dr. Anthony Skjellum, Dr. Warren Jones, Dr. Kevin Reilly, Dr. Alan Sprague, and Mr. Bruce Williams, I am grateful for your care and help.

Most importantly I thank God for giving me this opportunity to experience this wonderful journey.

I acknowledge financial support from IBM. The work presented in this dissertation was supported in part by the IBM Innovation Grant.

TABLE OF CONTENTS

Page

ABSTRACT iii

DEDICATION v

ACKNOWLEDGEMENTS vi

LIST OF TABLES xii

LIST OF FIGURES xiii

LIST OF ABBREVIATIONS xvi

CHAPTER

1 INTRODUCTION 1

1.1 The Benefits of Domain-Specific Languages 2
1.2 The Challenges of DSL Implementation 4
1.3 The Need for End-User DSL Testing Tools 6
1.4 Research Objectives and Contributions 8
1.5 Overview 12

2 BACKGROUND 14

2.1 Eclipse Plug-In Based Software Development 14
2.1.1 Eclipse Debugging Perspective 15
2.1.2 Eclipse JUnit Test Perspective 17
2.2 Program Transformation with the Design Maintenance System 19
2.3 Categories of Domain-Specific Languages 20
2.3.1 Imperative Domain-Specific Languages 20
2.3.2 Declarative Domain-Specific Languages 22
2.3.3 Hybrid Domain-Specific Languages 25
2.4 Aspect-Oriented Programming (AOP) 28
2.5 Syntax-Directed Translation 30

3 DSL DEBUGGING FRAMEWORK (DDF) 33

TABLE OF CONTENTS (Continued)

Page

CHAPTER

3.1 DDF Architecture Overview....33
3.2 Source Code Mapping....37
3.3 Debugging Methods Mapping....40
3.4 Debugging Results Mapping....44
3.4.1 Debugging Results Mapping Process....44
3.4.2 Debugging Results Mapping Example....45
3.4.3 Crosscutting Grammar Concerns....46
3.5 Illustrative Examples....47
3.5.1 Generation of an Imperative DSL Debugger....47
3.5.2 Generation of a Declarative DSL Debugger....51
3.5.3 Generation of a Hybrid DSL Debugger....59
3.6 Case Study Evaluation....62
3.6.1 Generalization of DDF Usage....63
3.7 Related Work in the Area of Domain-Specific Language Debuggers....67
3.7.1 Khepera....68
3.7.2 JSR-045....69
3.7.3 ANTLR Studio....70
3.7.4 TIDE....71
3.8 Summary....72

4 DSL UNIT TESTING FRAMEWORK (DUTF)....75

4.1 DUTF Architecture Overview....75
4.2 Source Code Mapping....79
4.3 Test Cases Mapping....79
4.4 Testing Results Mapping....83
4.5 Illustrative Examples....85
4.5.1 Generation of an Imperative DSL Unit Test Engine....85
4.5.2 Generation of a Declarative DSL Unit Test Engine....88
4.5.3 Generation of a Hybrid DSL Unit Test Engine....91
4.6 Case Study Evaluation....92
4.6.1 Generalization of DUTF Usage....92
4.7 Related Work of Domain-Specific Language Unit Test Engines....96
4.7.1 ASF+SDF....97
4.7.2 JST....97
4.7.3 LISA....98
4.7.4 SmartTools....99
4.7.5 Other Related Testing Tools....100
4.8 Summary....100

TABLE OF CONTENTS (Continued)

Page

CHAPTER

5 ASPECTG: WEAVING ASPECTS INTO DSL GRAMMARS 102

5.1 AspectG Design Challenges 102
5.2 AspectG Overview.....104
5.3 Weaving at the Generated GPL Code Level.....107
5.4 Weaving at the DSL Grammar Level 110
5.4.1 AspectG Specification 114
5.4.2 AspectG Implementation 118
5.5 Illustrative Examples 122
5.6 Related Work in the Area of Aspect-Oriented Grammars 128
5.6.1 AspectLISA 129
5.6.2 AspectASF 129
5.7 Summary 130

6 FUTURE WORK.....133

6.1 DSL Profiler Platform.....133
6.2 Application of Different IDE Platforms and GPLs 135
6.3 Adaptation of DDF and DUTF to Address more Complex DSLs.....136
6.4 Debugging Behavior through Event Grammars 137
6.5 Extending the Role of Aspects in Grammars 138

7 CONCLUSIONS.....140

LIST OF REFERENCES.....143

APPENDIX

A DOMAIN-SPECIFIC LANGUAGE GRAMMAR SPECIFICATIONS 152

A.1 Robot DSL Grammar Specification.....153

A.2 FDL Grammar Specification 154

A.3 SWUL Grammar Specification.....156

A.4 Hybrid Robot DSL Grammar Specification 158

B ASPECTJ CODE FOR POST-ANTLR GRAMMAR WEAVING 160

TABLE OF CONTENTS (Continued)

Page

CHAPTER

C ASPECTG GRAMMAR SPECIFICATION 163

D PARLANSE TRANSFORMATION FUNCTIONS FOR ASPECTG 166

D.1 After Weaving Function 167

D.2 Middle Weaving Function 169

LIST OF TABLES

Table *Page*

3-1 Generality Analysis of DDF 64

3-2 JDB, GDB, and Cordbg Basic Debugging Commands Comparison 66

3-3 The DDF Adaptation for JDB, GDB, and Cordbg 67

4-1 Generality Analysis of DUTF....93

4-2 JUnit and NUnit Basic Unit Test Actions Comparison....95

5-1 Usage Analysis for AspectG....131

LIST OF FIGURES

Figure *Page*

1-1 The Need for DSL Testing Tools 6

1-2 Matrix of DSL Tools and Language Categories....8

2-1 Screenshot of the Eclipse Debug Perspective 17

2-2 Screenshot of the JUnit Eclipse Plug-in 19

2-3 Robot DSL Sample Code 22

2-4 Car Features Specified in FDL and List of Possible Car Configurations....24

2-5 Robot Language Syntax Specification in BNF Format 25

2-6 SWUL Sample Code 27

2-7 Hybrid Robot DSL....28

3-1 DSL Debugging Framework (DDF)....35

3-2 Debugger Generation Overview....36

3-3 Robot DSL Source Code Mapping....38

3-4 Part of Robot DSL Grammar Specification....39

3-5 Part of SWUL Grammar Specification....40

3-6 Mapping of Debugging Actions between DSL and GPL 41

3-7 DSL Debugging Step Over Algorithm 42

3-8 Debugging Result Mapping for the **`INIT`** Production of the Robot Grammar....46

3-9 **`stepinto`** function in **`DSLDebugTarget`**....49

3-10 Screenshot of Debugging Session on Robot Language 51
3-11 Screenshot of Debugging Session on Car Program 53
3-12 Action and Goto table of Robot Language LR-parsing 56
3-13 Screenshot of Debugging Session on Robot BNF 57
3-14 Screenshot of another Version of Debugging Session on Robot BNF 58
3-15 Screenshot of Debugging Session on SWUL Program 60
3-16 Screenshot of Debugging Session on Hybrid Robot Program 62
4-1 DSL Unit Testing Framework (DUTF) 77
4-2 DSL Unit Test Engine Generation Process 78
4-3 Robot Language Test Cases Mapping 81
4-4 FDL Test Cases Mapping 83
4-5 Screenshot of Unit Testing Session on Robot Language 86
4-6 Correct and Incorrect Knight Methods 87
4-7 **`handleDoubleClick`** Method in **`TestResultView`** Class 88
4-8 Screenshot of Unit Testing Success Session on Car FDL 89
4-9 Screenshot of Unit Testing Failure Session on Car FDL 90
4-10 Comparison of JUnit and NUnit Assertion Usage 96
5-1 Robot DSL Specification in ANTLR 105
5-2 Part of Robot DSL Specification with Additional Debug Information 106
5-3 Post-ANTLR Processing (AspectJ Approach) 107
5-4 Fragment of DSL Line Mapping Aspect in AspectJ 108
5-5 Fragment of DSL Last Line Tracking Aspect in AspectJ 109
5-6 Steps to Weave Debugging Aspects into an ANTLR Grammar 112

5-7 Pre-ANTLR Processing (DMS Approach) 113
5-8 AspectG Pointcut Model 115
5-9 Generalized Algorithm for AspectG Weaving 119
5-10 Part of **`after`** function in PARLANSE 121
5-11 DSL Line Number Counter Aspect in AspectG 122
5-12 Low-level Rule Transformation Generated from AspectG 124
5-13 Applied Weaving of "After" Transformation Rule on the Robot Grammar 125
5-14 GPL Line Number Counter Aspect in AspectG 126
5-15 Low-level Rule Transformation Generated from AspectG 127
5-16 Applied Weaving of "Middle" Transformation Rule on the Robot Grammar 128
6-1 DSL Profiler Framework (DPF) 135

LIST OF ABBREVIATIONS

ANTLR	Another Tool for Language Recognition
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
API	Application Program Interface
ASF	Algebraic Specification Formalism
AST	Abstract Syntax Tree
BNF	Backus–Naur Form
CUP	Constructor of Useful Parsers
DDF	DSL Debugging Framework
DMS	Design Maintenance System
DOM	Document Object Model
DPF	DSL Profiling Framework
DSAL	Domain-Specific Aspect Language
DSLs	Domain-Specific Languages
DTTS	DSL Testing Tool Studio
DUTF	DSL Unit Testing Framework
EUSES	End-Users Shaping Effective Software Consortium
FDL	Feature Description Language
GAL	Graphics Adaptor Language

GDB	GNU Project Debugger
GPAL	General-Purpose Aspect Language
GPLs	General-Purpose Languages
IDE	Integrated Development Environment
JPDA	Java Platform Debugger Architecture
JPM	Join Point Model
JTS	Jakarta Tool Suite
LISA	Language Implementation System based on Attribute Grammars
OOP	Object-Oriented Programming
PARLANSE	Parallel Language for Symbolic Expression
PDE	Plug-in Development Environment
PTE	Program Transformation Engine
RSL	Rule Specification Language
SDF	Syntax Definition Formalism
SDK	Software Development Kit
SQL	Structured Query Language
SSCLI	Shared Source Common Language Infrastructure
SST	Surface Syntax Tree
SWUL	Swing User Interface Language
TDD	Test-Driven Development
TIDE	ToolBus Integrated Debugging Environment
VHDL	Very High Speed Integrated Circuit Hardware Description Language

WYSIWT	What You See Is What You Test
XP	Extreme Programming
YACC	Yet Another Compiler-Compiler

CHAPTER 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 [Sutcliffe and Mehandjiev, 2004]. Those 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 [Burnett *et al.*, 2005], which can be programmed by scripts that use arithmetic and statistical formulas of a specific domain instead of using traditional programming language concepts [Burnett *et al.*, 2003]). 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* estimate that there are approximately 2.75 million professional developers out of an estimated 80 million end-user programmers [Scaffidi *et al.*, 2005]), with the vast majority of end-user developers building applications using tools such as spreadsheets, query systems, or interactive scripting websites.

End-user programmers are more likely to introduce software errors than professional programmers because they lack software development training and proper tool support [Harrison, 2005]. As observed in many industry studies, individual examples of software errors have cost the economy millions of dollars in recent cases [Hilzenrath, 2003; Schmitt, 2005]. According to a 2002 study, it was estimated that software failures collectively contribute to over $60 billion in unreported losses per year [Tassey, 2002; Crissey, 2004]. Without the availability of standard software development tools, the final products of end-user programming can be dangerous [Harrison, 2005]. The proper programming tools (e.g., editor, compiler, test engine, and debugger) are needed for end-users to ensure the integrity of the products they develop. With a large pool of end-user developers, and the rising cost of software failures, it is imperative that end-users are provided with tools that allow them to detect and find software errors at an abstraction level that is familiar to them.

1.1 The Benefits of Domain-Specific Languages

To assist end-users in describing solutions to their work tasks, Domain-Specific Languages (DSLs) [Wile and Ramming, 1999] have been promoted as an approach to remove the dependence on traditional General-Purpose Languages (GPLs), such as Java and C++. A DSL is a, "programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain" [van Deursen *et al.*, 2000]. A DSL is a programming language tailored toward the specific needs of a particular problem domain to ease the development of software solutions for that domain [van Deursen *et al.*, 2000;

Mernik *et al.*, 2005]. DSLs are often described as "little languages" [Bentley, 1986; van Deursen and Klint, 1998] that are designed to solve problems in particular domains. DSLs hide the lower level programming language details such as complex data structures, complicated algorithms, and tedious GPL syntax from DSL programmers. DSLs have also been shown to assist in software maintenance whereby end-users can directly use the DSLs to make required routine modifications [Bentley, 1986]. 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 [van Deursen and Klint, 1998]. The goal of providing suitable programming abstractions for end-users is also a key principle of intentional programming [Simonyi *et al.*, 2006]. Modifications to DSL programs are easier to make and can be understood and validated by domain experts who are not familiar with a GPL, or do not know how to program in a GPL. The empirical evidence suggests that the use of DSLs increases flexibility, productivity, reliability, and usability [Kieburtz *et al.*, 1996; Wile, 2004; Mernik *et al.*, 2005] so that DSLs can shorten the application development time and reduce the development cost significantly.

DSLs describe problems at a level familiar to domain experts. Without dealing with generated GPL code, domain experts can concentrate their time and effort on utilizing their domain knowledge to develop solutions without concern for how to express, interpret, and solve the problem in an unfamiliar notation. The declarative and concise nature of some of DSLs makes them easy to understand by eliminating the

abstraction leaks common in representing domain concepts using lower level language abstractions (e.g., GPLs).

Some of the more popular DSLs include the language used in the Unix make utility [Oram and Talbott, 1991] and the language used to specify grammars in parser generators like YACC (Yet Another Compiler-Compiler) [Johnson, 1975]. Other examples include the Very High Speed Integrated Circuit Hardware Description Language (VHDL) [VASG, 2007], which is a DSL to model a digital hardware system; Structured Query Language (SQL) is a DSL to access and manipulate databases [Groff and Weinberg, 2002]; Graphics Adaptor Language (GAL) is a DSL to specify video device drivers [Thibault *et al.*, 1999]; CSounds is a DSL used to create audio files [CSounds, 2007], and Mawl is a DSL to specify form-based services in a device-independent manner [Atkins *et al.*, 1999].

1.2 The Challenge 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. The development of DSLs requires knowledge of programming language implementation, as well as domain knowledge. Building a test engine and debugger for each DSL from scratch can be time consuming, error-prone, and costly. It is difficult to build new testing tools for each new language of interest and for each supported platform because each language tool depends heavily on the underlying operating system's capabilities and lower level native code functionality [Rosenberg, 1996]. The goal of this dissertation is to

show how the cost of developing DSL tools can be minimized by an automated grammar-driven tool generation approach that extends a popular Integrated Development Environment (IDE).

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 [Mernik *et al.*, 2005]. The majority of the DSL implementation patterns translate a single DSL construct into several constructs in a target GPL. The most popular strategy for implementing a DSL, which is also adopted in this dissertation, is a pre-processor serving as a compiler and application generator that performs a source-to-source transformation (i.e., the DSL source code is translated into the source code of an existing GPL [Mernik *et al.*, 2005]). 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, debugger, unit test engine, and profiler). The higher level abstractions of a particular domain are built into the translator that synthesizes a DSL program into a GPL program. The technique to implement such DSL translators can vary. In this dissertation, the pre-processor implementation pattern is chosen where a DSL is translated using a syntax-directed approach by the translators rather than a complete compiler, which introduces new maintenance issues (e.g., if the language definition changes, the translator has to be modified accordingly [van Deursen and Klint, 1998]).

1.3 The Need for End-User DSL Testing Tools

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 - please see Figure 1-1). Usually, domain experts describe a problem at the DSL level where the DSL applications are translated into GPL code so that the actual computations or tasks can be performed. Even though some DSL programming tools can be generated (e.g., editor, parser, and visualizer), the DSL development environments available to domain experts are limited (e.g., lack of DSL testing tool support). Domain experts prefer to develop their DSL applications at the DSL level during the different software development phases instead of being forced to test and debug their applications at the generated GPL level.

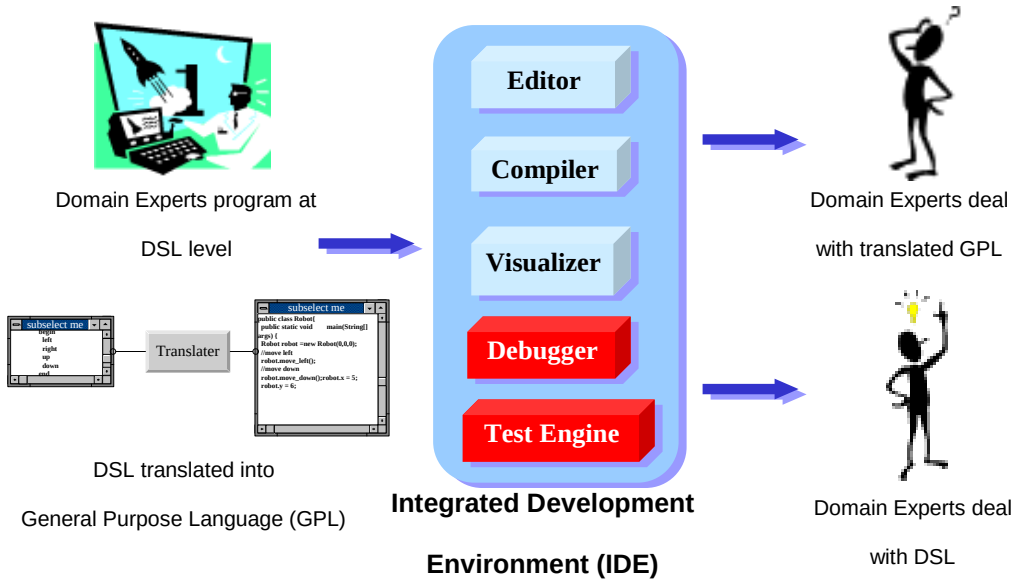


Figure 1-1. The Need for DSL Testing Tools

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 and test the translated program at the GPL level [Wu *et al*., 2007]. 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, who argue for the need to perform analysis at the DSL level, not at the GPL level [Van Wyk and Johnson, 2007]. Although techniques for constructing DSL tools (e.g., editor and compiler) have been developed over the years, support for debuggers and test engines for DSLs have not been investigated deeply. This dissertation describes how automated tool generation can be used to overcome the lack of testing 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 [Floyd, 1979]. The same comment can apply to DSL tools; i.e., if sufficient 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 be realized also 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 [Parnas and Siewiorek, 1975]. 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 the 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.

1.4 Research Objectives and Contributions

This dissertation introduces a DSL tool framework that can automatically generate various testing tools (e.g., debuggers, test engines, and profilers) for different categories of DSLs (e.g., imperative, declarative, and hybrid). Figure 1-2 illustrates a research matrix along the vertical direction (representing the various DSL testing tools) and the horizontal direction (representing the classes of DSL languages to be supported).

The vertical direction of Figure 1-2 corresponds to the vector representing the various testing tools applied to the same type of DSL. The testing tools that have been generated from the grammar-driven approach include several DSL debuggers and unit test engines. As discussed at the end of the dissertation, future work includes generation of profilers for DSLs.

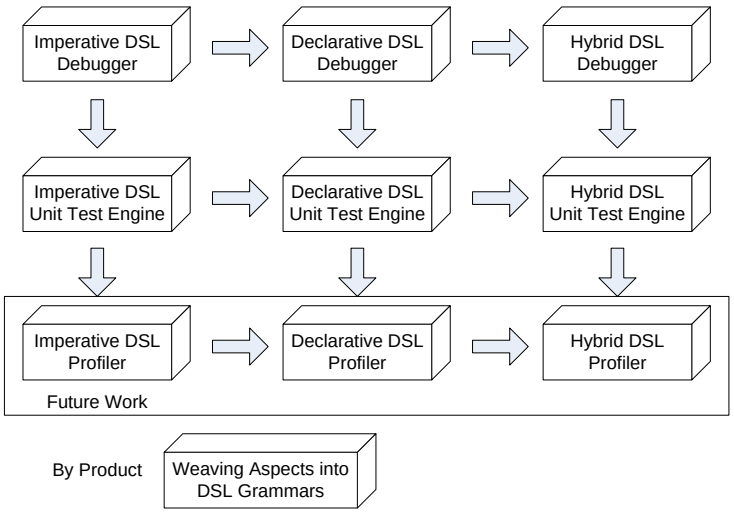


Figure 1-2. Matrix of DSL Tools and Language Categories

A debugger enables programmers to inspect and discover the errors in their programs during program execution. Zellweger categorizes debuggers into two classes: expected behavior debugger and truthful behavior debugger [Zellweger, 1984]. 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" [Zellweger, 1984]. A truthful behavior debugger "displays how optimizations have changed the program portion under consideration or it admits that it cannot give a correct response" [Zellweger, 1984]. As categorized by [Auguston, 1995], 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 [Auguston, 1998]). The work described in this dissertation represents expected behavior debuggers that perform typical debugging tasks on DSL programs (e.g., set a break point, stop at the break point, display variable values, and step

through the program), which enables a better understanding of the run-time behavior of a DSL program.

A unit test engine is a development tool used to determine the correctness of a set of modules (e.g., classes, methods, or functions) by executing source code against specified test cases. Each unit test case is tested separately in an automated fashion using a test engine. The test results help a programmer identify and fix the errors in their program. To provide tool support consistent with accepted software engineering practice, a DSL unit test engine provides end-users the ability to discover the existence of software errors, and DSL debuggers can further help end-users to locate the errors in the DSL code.

A profiler is a meta-program that gathers information about another program's performance measurements by recording and computing over run-time event traces from hardware (e.g., timer triggers) or software (e.g., function call and OS scheduling) [Auguston, 1998]. After execution of a program, a profiler displays summary information of recorded event traces and their corresponding occurrences in the program. A DSL profiler is helpful to determine performance bottlenecks during the execution of a DSL.

The contributions described in this dissertation can be summarized by the following two objectives:

- **Assist in generation of DSL testing tools using a generative framework**

  Even though the individual tools can be implemented separately for each DSL category, the core research contribution concerns the investigation into a generalized method that enables construction of a matrix of DSL testing tools as a type of software factory [Greenfield *et al.*, 2005]. The matrix of tools and languages can be considered as a family of systems that is a domain-specific

product-line architecture, where a set of different products with common characteristics adapt to a set of distinct features [Clements and Northrop, 2002]. The automation provided by generative programming [Czarnecki and Eisenecker, 2000] offers an extensible mechanism as an alternative to manual tool construction by transforming higher level specifications to lower level equivalent program applications. This approach also conforms to the software development paradigm of step-wise refinement on language tool construction [Batory *et al.*, 2004]. To realize this generative approach, the DSL Debugger Framework (DDF) and DSL Unit Test Framework (DUTF) have been implemented, which reuse the existing testing support in Eclipse and Java.

- **Raise Aspect-Oriented (AO) concepts to a higher level of abstraction: Aspects for language specification and grammar weaving**

  As a by-product of this research, an aspect language (described in Chapter 5) can weave crosscutting tool concerns directly into a DSL grammar as an aid toward the rapid generation of new DSL testing tools. Chapter 5 also discusses the details of the investigation into aspect-oriented programming [Kiczales *et al.*, 1997] to assist in modularizing the DSL tool concerns (e.g., debugging and testing) from a base grammar. Although there have been other efforts that explore AO on different software artifacts at various lifecycle stages (e.g., source code and models), the work described in this dissertation represents one of the first occurrences in the research literature of an actual aspect-oriented weaver that is focused on language specification and

grammar weaving, rather than topics that are applicable to traditional programming language source code.

The research described in this dissertation offers three key contributions. The first contribution provides an initial step toward empowering end-user developers with traditional software engineering testing capabilities at the DSL level. To accomplish this objective, a grammar-driven DSL tool framework has been developed that generates testing tools (e.g., debugger and testing engine) automatically from DSL grammar specifications. The 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 testing and debugging a diverse set of DSLs. Different types of DSLs have different language characteristics that require specific features. The third contribution is the exploration of a technique for better separation of concerns in Grammarware [Klint *et al.*, 2005], which comprises grammars and all grammar-dependent software (e.g., lexer, parser). A key benefit is the ability to explore numerous scenarios by considering crosscutting grammar concerns as aspects that can be used to generate DSL testing tools.

1.5 Overview

The remainder of this dissertation is structured as follows: Chapter 2 introduces the necessary background information to provide the reader with a better understanding of other sections of the dissertation. The first part addresses Eclipse plug-in development, which is used to build the front-end of the framework. A program transformation engine is used to build the back-end of the framework, which performs the actual transformation

on the grammar file. The last part of Chapter 2 introduces categories of DSLs including examples of different types of DSLs that have been used during this research.

Chapters 3 and 4 begin by describing an overview of the architecture of DDF and DUTF. These chapters also provide details about the implementation of DDF and DUTF, including the construction of the different components, the algorithms involved, experimentation results, and generalized usage of the DDF and DUTF. Other related software engineering practices are also discussed in these two chapters. Each of these two chapters provide experimental evaluation of the contribution and offer a discussion of the importance of considering software engineering factors while developing tools for DSLs.

Chapter 5 introduces an investigation of aspects applied to grammars to enable better separation of concerns during the testing tool generation process. The chapter begins by describing an overview of the grammar weaving process. The rest of Chapter 5 details the implementation of AspectG, including the weaving process, the algorithms involved, and experimentation results. Other related approaches to aspect-oriented grammar weaving are discussed in this chapter, such as AspectLISA and AspectASF.

Chapter 6 describes several existing limitations that serve as a focus of future extensions of this work. Chapter 7 offers a concluding summary of the research contributions. Appendix A provides the specification of the DSLs presented in this dissertation; Appendix B provides the AspectJ code for the post-ANTLR grammar weaving approach; Appendix C provides the specification of AspectG in ANTLR notation; and Appendix D provides the PARLANSE transformation functions for AspectG implementation.

CHAPTER 2

BACKGROUND

This thesis presents research that unites the descriptive power provided by the Eclipse debugging perspective (Section 2.1.1) and the JUnit testing engine (Section 2.1.2), in conjunction with the invasive modification capabilities of a mature program transformation system (Section 2.2). To provide the necessary background of the basic tools and techniques mentioned throughout the dissertation, this background chapter offers a brief description of these concepts and tools. This chapter also includes the description of three different types of DSLs in Section 2.3, which are used as case studies throughout the remaining chapters of the dissertation. A survey of software engineering techniques and practices used in this research are also outlined (e.g., Aspect-Oriented Programming (AOP) in Section 2.4 and syntax-directed translation in Section 2.5).

2.1 Eclipse Plug-In Based Software Development

Eclipse is an open-source development platform for constructing customized Integrated Development Environments (IDEs) that can be used to create diverse applications [Eclipse, 2007]. A key characteristic of Eclipse is the ability to serve as a tool integration platform that offers numerous extension points for software feature customizations through a plug-in architecture. As a tool integration framework, Eclipse

has been defined as “a collection of places-to-plug-things-into (extension points) and things-to-plug-in (extensions)” [Gamma and Beck, 2003]. 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) offers a powerful platform to develop and integrate different language tools that support DSL development [Eclipse, 2007].

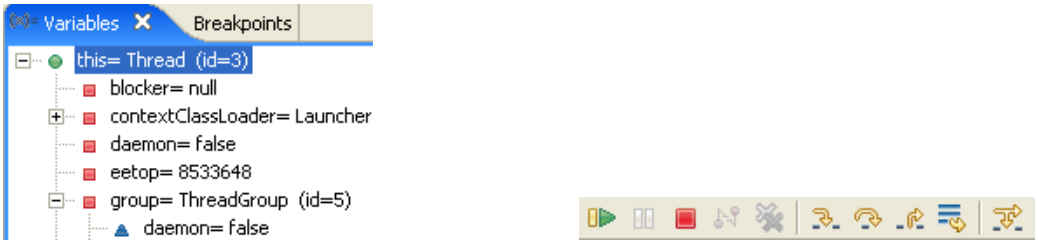
2.1.1 Eclipse Debugging Perspective

To assist in construction of new debugger interfaces, the Eclipse Software Development Kit (SDK) provides the debugging perspective, which is a framework for building and integrating debuggers. As shown in Figure 2-1, the debug perspective defines a set of interfaces that model common debugging artifacts (e.g., threads, variables, and breakpoints) and debugging navigation actions (e.g., stepping, suspending, resuming, and terminating) [Wright and Freeman-Benson, 2004]. The debug perspective appears when programmers select the debugging mode for program execution. 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 debugging perspective consists of a language editor, a variable view, and a debugger view. The Eclipse debugger perspective consists of a console view in an initial layout that is designed to perform basic debugging functions on the source code in the editor.

The language editor is the place where the targeted source code resides. The breakpoints and program pointer appear on the left frame of the editor. A breakpoint is the location where a programmer wants the program to stop during the execution. A program pointer is the current execution point of the running program. The variable view is in the upper-right corner of the IDE and displays the local variable values. The debugger view is in the upper-left corner of the IDE and shows the current execution status of the running program such as threads and function names. The top part of the debugger view lists different debugging actions that programmers can invoke. Depending on the current program behavior and logic, some debugging actions may not be applicable and will appear as disabled in the debugging interface. The editors, views, and perspectives can be modified, extended, and rearranged according to a user's specific needs. 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.

Chapter 3 describes an extension to the Eclipse debugging perspective to integrate with an interactive debugging framework that assists in debugging a program written in a DSL.



a) Variable Inspector b) Navigation Toolbar

Figure 2-1. Screenshot of the Eclipse Debug Perspective

2.1.2 Eclipse JUnit Test Perspective

Unit testing is a testing approach [Zhu *et al.*, 1997] that isolates the individual units of program source code and validates the correctness of each unit against its requirements [Unit Testing, 2007]. JUnit is a popular unit testing tool for constructing automated Java test cases that are easy to write, composable, and isolated [JUnit, 2007]. JUnit is not only a stand-alone tool but also is adapted to the Eclipse plug-in development environment. A JUnit plug-in for Eclipse provides a framework for automating functional unit testing on Java programs with integrated JUnit support. JUnit generates a skeleton of unit test code according to the tester's specification. The software developer needs to specify the expected value, the tested variable, the tested module of the source code, and the test method of the test cases. JUnit provides a set of rich testing methods (e.g., **`assertEquals`**, **`assertNotNull`**, **`assertFalse`**, and **`assertSame`**) and reports the results (shown in Figure 2-2) as: the total number of passed or failed test cases; the true expected value and current expected value of the failed test cases; the name and location of the passed and failed test cases; and the total execution time of all the test cases. The test results can be traced back to the source code locations of the tested program. The test cases are displayed in a hierarchical tree structure that defines the

relationship among test cases. There are several benefits that JUnit offers in terms of test automation and reuse that are provided by a common fixture that encapsulates all testing. As noted in the JUnit documentation, "JUnit also provides a common structure to all tests that programmers can set up a test fixture, run some code against the fixture, check test results, and then clean up the fixture. This means that each test will run with a fresh fixture and the results of one test can't influence the result of another. This supports the goal of maximizing the value of the tests" [JUnit, 2007]. A collection of related test cases is called a test suite. When one of the test cases fails, the entire test suite is declared a failure. Template methods are used to separate and parameterize the fixture (e.g., set up and tear down), which make JUnit test cases easy to write. In each test case, assertion predicates assess the expected outcome against the actual outcome after executing a program unit. The results of the tests are reported to programmers in either a graphical summary or plain text.

In its current form, JUnit is focused solely on Java and is not applicable to general testing of DSL programs. In Chapter 4, we describe how our mapping framework enables unit testing of DSL programs using JUnit as the underlying unit test engine.

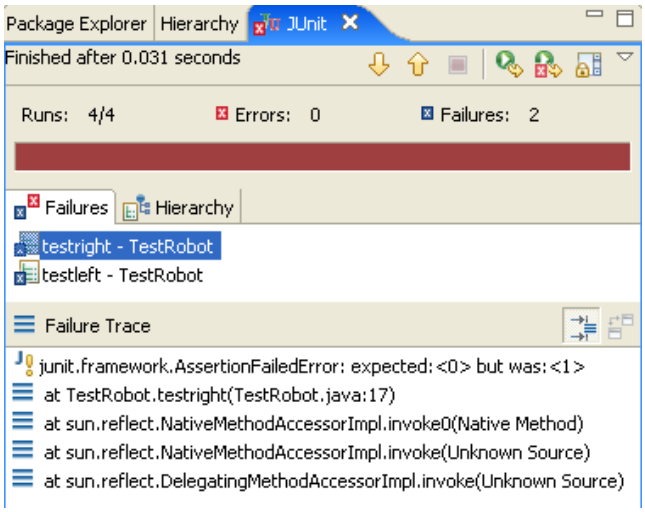


Figure 2-2. Screenshot of the JUnit Eclipse Plug-in

2.2 Program Transformation with the Design Maintenance System

A program transformation engine facilitates the transformation of a source program into a new program representation. In some cases, the original behavior of the source program needs to be preserved (e.g., code refactoring and code optimization). Typically, a program transformation is based on transformation rules that specify pattern matching on an abstract syntax tree (AST). A transformation rule also defines the rewrite action needed for those parts of the AST that are matched by the rule's pattern.

The Design Maintenance System (DMS) is a transformation and re-engineering toolkit developed by Semantic Designs [Baxter *et al.*, 2004]. In addition to DMS, there are many other popular program transformation systems (e.g., ASF+SDF [van den Brand *et al.*, 2002], Stratego [Visser, 2001], and TXL [Cordy, 2006]). DMS is a commercially available product and provides lower level transformation functions such as parsing, AST generation and manipulation, pretty printing, powerful pattern matching, and source translation capabilities. DMS also provides pre-constructed domains for several dozen

GPLs (e.g., Java, C++, and Object Pascal). In DMS, a language domain contains a lexer, parser, and pretty printer, as well as additional language tools such as type analysis tools.

In addition to the available parsers, the underlying rewriting engine of DMS offers the machinery needed to perform invasive software transformations on legacy code [Aßmann, 2003]. DSL language developers (not the actual end-users) can create a new language domain using the source-to-source transformation functionalities of DMS. For the purpose of the research presented in this dissertation, DMS is used as the underlying transformation engine to support the implementation of an aspect language for grammars (called AspectG) to weave crosscutting testing tool concerns into DSL grammars. In Chapter 5, we describe aspects applied to grammars in order to generate new DSL testing tools.

2.3 Categories of Domain-Specific Languages

To demonstrate the generality of the approach described in the dissertation, three different categories of DSLs were considered. The horizontal direction of Figure 1-2 focuses on facilitating the construction of the same software tool (e.g., debugger and unit test engine) across these three different categories of DSLs (e.g., imperative DSL, declarative DSL, and hybrid DSL). This sub-section introduces the definitions and specific differences among the categories of DSLs described throughout this dissertation.

2.3.1 Imperative Domain-Specific Languages

An imperative programming language is based on the von Neumann concept that is centered on assignment expressions and control flow statements [Sebesta, 2007], 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, testing tools are designed around capabilities to examine the value of variables at run-time.

For the purpose of this research, a simple imperative language for representing robot control was adopted from previous case studies [Wu *et al.*, 2004; Mernik and Žumer, 2005]. 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 dissertation, the Robot language has been extended by adding user-defined function definitions and function calls. 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 coordinate. Additional Robot DSL statements are: initial statement, set statement, and print statement. Figure 2-3 represents sample code written in the Robot DSL – lines 15 to 19 define knight; line 21 sets the robot's initial position to <0, 0>; line 24 invokes knight; line 25 forces <5, 6> as the robot's new current position; and line 28 prints the robot's current position. The complete Robot language grammar written in ANTLR is contained in Appendix A.1.

```
…
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 2-3. Robot DSL Sample Code

2.3.2 Declarative Domain-Specific Languages

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 declarative 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 [Sebesta, 2007]. Instead of assessing the value of individual variables, a declarative DSL testing tool needs to evaluate the relationships between each declaration, which are represented as data structures with symbolic logic.

As an example of a declarative DSL, the Feature Description Language (FDL) is used in this dissertation to specify the legal configuration of an automobile product line [van Deursen and Klint, 2002]. The FDL is a textual language that describes a feature diagram [Czarnecki and Eisenecker, 2000], 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 upper part of Figure 2-4 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 is atomic and cannot be expanded further (e.g., the **carbody** feature). If the first character of a feature is an uppercase character, the feature is composite, which may consist of other composite or primitive features (e.g., the **Transmission** feature consists of two primitive features, **automatic** and **manual**). In feature 2 of Figure 2-4, the **oneof** composition logic operator states that **Transmission** can be either **automatic** or **manual**, but not both. In feature 3, the **moreof** composition logic operator specifies that the **Engine** can be either **electric** or **gasoline**, or both. In constraint 1, all cars are required to have a **pullsTrailer**. In constraint 2, only **highPower** cars are associated with the **pullsTrailer** feature. The combination of constraints 1 and 2 imply that all cars in this product line must be **highPower**. The lower part of Figure 2-4 enumerates all of the possible legal configurations that result from the features defined on the upper part of the figure. The complete FDL grammar written in ANTLR is contained in Appendix A.2.

Car features in FDL

```
feature 1: Car: all (carbody, Transmission, Engine,
                     Horsepower, opt(pullsTrailer))
feature 2: Transmission: oneof (automatic, manual)
feature 3: Engine: moreof (electric, gasoline)
feature 4: Horsepower: oneof (lowPower, mediumPower, highPower)
constraint 1: include pullsTrailer
constraint 2: pullsTrailer requires highPower
```

All possible car configurations

```
1:(carbody, pullsTrailer, manual, highPower, gasoline, electric)
2:(carbody, pullsTrailer, manual, highPower, electric)
3:(carbody, pullsTrailer, manual, highPower, gasoline)
4:(carbody, pullsTrailer, automatic, highPower, gasoline, electric)
5:(carbody, pullsTrailer, automatic, highPower, electric)
6:(carbody, pullsTrailer, automatic, highPower, gasoline)
```

Figure 2-4. Car Features Specified in FDL and List of Possible Car Configurations (adapted from [van Deursen and Klint, 2002])

As another example, Backus–Naur Form (BNF) is a declarative DSL for formally describing the syntax of a language using a context-free grammar [Aho *et al.*, 2007]. 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 notation is shown in Figure 2-5, 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 2.3.1.

```
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 2-5. Robot Language Syntax Specification in BNF Format

2.3.3 Hybrid Domain-Specific Languages

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 [Bravenboer and Visser, 2004]. 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 [Spinellis, 2001; Mernik *et al.*, 2005]. The piggyback pattern is widely adopted in DSLs for tools like parser generators, such as Yet Another Compiler-Compiler (YACC) [Johnson, 1975], Bison [Bison, 2007], ANother Tool for Language Recognition (ANTLR) [ANTLR, 2007] or the Constructor of Useful Parsers (CUP) [CUP, 2007].

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 [Loy *et al.*, 2002] 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 2-6 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 [Bravenboer and Visser, 2004]. 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 and 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 2-6 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 GPL that has an embedded DSL provides an ability to remove the accidental complexities of library usage. The complete SWUL grammar written in ANTLR is contained in Appendix A.3.

```
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 2-6. SWUL Sample Code (SWUL code in italics)

Figure 2-7 is another example of a hybrid DSL that represents an extension of the Robot DSL from Section 2.3.1. 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 2-7 introduces a new random move that requests from the user the boundaries for a random number generator that produces random coordinates. 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 a GPL may suggest that a new construct should be added in the next version of

the DSL to support a commonly needed feature. The complete Hybrid Robot DSL grammar written in ANTLR is contained in Appendix A.4.

```
…
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 2-7. Hybrid Robot DSL (Java code in italics)

2.4 Aspect-Oriented Programming (AOP)

Programmers often encounter a situation where identical or similar functionality is spread over an application's code base and is difficult to modularize. The nature of certain business processes and real-world problems may force some concern dimensions to be scattered across different modules and tangled within a single module, hampering the proper separation of concerns [Dijkstra, 1976]. Even though the Object-Oriented Programming (OOP) paradigm supports modularization and reusability through encapsulation, inheritance, and polymorphism, a new language construct is needed for identifying, encapsulating, and manipulating the separation of concerns to complement traditional programming languages. Capturing scattered and tangled code is modularized as *aspects*, which are concerns of interest that are specified in a single location that is

modularized. Aspect-Oriented Programming (AOP) assists software engineers in modularizing and decomposing crosscutting concerns into a more manageable fashion, which has been shown to improve the comprehensibility, changeability, and maintainability of the whole software system.

AOP techniques for general-purpose programming languages have been developed (e.g., AspectJ [Kiczales *et al.*, 2001], [AspectC#, 2007], and [AspectC, 2007]). These languages represent general-purpose aspect languages (GPALs) applied to GPL domains (e.g., Java, C#, and C). A GPAL is an aspect language that provides general constructs that improve modularization of a broad range of crosscutting concerns bounded within a specific GPL. Crosscutting concerns emerge not only in GPL source code, but also in various representations of software artifacts (e.g., models [Gray *et al.*, 2001], non-functional software requirements [Duclos *et al.*, 2002], and programming language grammars [Wu *et al.*, 2004]). However, most research and development efforts have been devoted toward bringing AOP support to programming languages, rather than artifacts from other phases of software development. Just recently, the research trend has shifted towards describing specific crosscutting concerns (e.g., model evolution, language extension, and tool generation) that provide language constructs tailored to the particular representation of such concerns; such languages are called domain-specific aspect languages (DSALs). A DSAL that addresses tool generation concerns observed in language grammars is described in Chapter 5.

There are three main language components of a typical aspect-oriented language: join points, pointcuts, and advice. A join point indicates the location in the program where a specific crosscutting concerns appears. This location can be either a static

location of a particular segment of source code, or it can be a dynamic program execution point. A pointcut is a set of join points and is specified by designators that are declarative keywords indicating characteristics that identify the essence of a set of common join points. Advice is a set of behavior that is attached to specific join points. Advice can be attached to pointcuts with specific behavior that represent methods or operations written in a GPL notation.

2.5 Syntax-Directed Translation

Popular parser and lexer generators (e.g., ANTLR, CUP, and YACC) aid programming language designers in constructing new programming languages by translating a language specification into a lexer and parser [Parr, 2007]. These tools provide an extensible framework and allow walking and manipulating ASTs for building language compilers based on various technologies (e.g., Visitor Pattern [Gamma *et al.*, 1995] and program transformation). Syntax-directed translation is a grammar-oriented compiling technique where an input-output mapping is based on a context-free grammar that specifies the syntactic structure of the input [Aho *et al.*, 2007]. Embedded within the right-side of each grammar production is a set of semantic rules for computation associated with the grammar symbols appearing in that production.

A DSL grammar is often defined using a standard language specification notation, such as BNF. Based on the DSL grammar written in BNF, language design tools can generate the language lexer and parser for a DSL. Also, by modifying the semantic specification of a DSL's BNF definition, the additional mapping information of the

translation from the DSL code to the generated GPL code can be generated by the syntax-directed translation process of the modified parser.

Throughout this dissertation, ANTLR [ANTLR, 2007] is used as the language construction tool to define the various DSLs that are discussed. ANTLR is a parser generator that provides a framework for constructing various programming language related tools (e.g., recognizers, compilers, and translators) from grammatical specifications. The ANTLR specification language is based on EBNF notation and enables syntax-directed generation of a lexer and parser. The tokens comprising the lexical part of the grammar for the new language are defined using named regular expressions. The parser representing the syntax and semantic parts of the language specification is defined as a subclass of the grammar specification and encapsulates semantic rules within each grammar production. The semantic actions within each production rule are written in a GPL (e.g., Java, C#, C++, or Python).

The Eclipse PDE provides a platform for creating a new domain-specific language environment, which allows the developer to extend the basic features that are suitable for a particular DSL through Eclipse's plug-in extension mechanism. Within the Eclipse platform, crosscutting concerns emerged from the DSL testing tool generation process. A program transformation approach to implement an aspect-oriented weaver assists in modularizing crosscutting concerns at the language grammar level. The framework developed to support the research has been implemented as Eclipse plug-ins (i.e., the DSL debugging framework discussed in Chapter 3 and the DSL unit testing framework discussed in Chapter 4). All of the DSL debuggers and unit test engines

presented in this dissertation were developed in Eclipse. The aspect language discussed in Chapter 5 is developed using a program transformation engine (i.e., DMS).

CHAPTER 3

DSL DEBUGGING FRAMEWORK (DDF)

This chapter presents a technique to build a debugging tool generation framework from existing DSL grammars. It utilizes existing GPL debuggers and a plug-in software development environment to simulate the end-user's debugging intention. By augmentation of original DSL grammars, hooks are generated to interface with the IDE. Five case studies presented in this chapter illustrate how the DSL debuggers are generated by DDF. This chapter introduces an approach that can provide debugging tool support for DSLs at a higher abstraction level. In addition, to demonstrate the benefits of this approach, experimental evaluation is discussed, including generality analysis and experimental results. At the end of this chapter, related work and a concluding discussion are also presented.

3.1 DDF Architecture Overview

The DSL Debugging Framework (DDF) provides a grammar-driven technique for reusing an existing GPL debugger in conjunction with the debugging interface available in Eclipse. An illustrative overview of the DDF is shown in Figure 3-1. A key technique of the DDF is a mapping process that records the correspondence between the DSL and the generated GPL. The ANTLR 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 3-1). The results from these first two mapping processes are re-interpreted 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 [Ryu and Ramsey, 2005]. 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 debugger perspective at the DSL level to determine what type of debugging commands need to be issued to a command-line debugger at the GPL level. The semantic actions associated with the debugger use syntax-directed translation and additional semantic functions in the grammar specification to generate the mapping information.

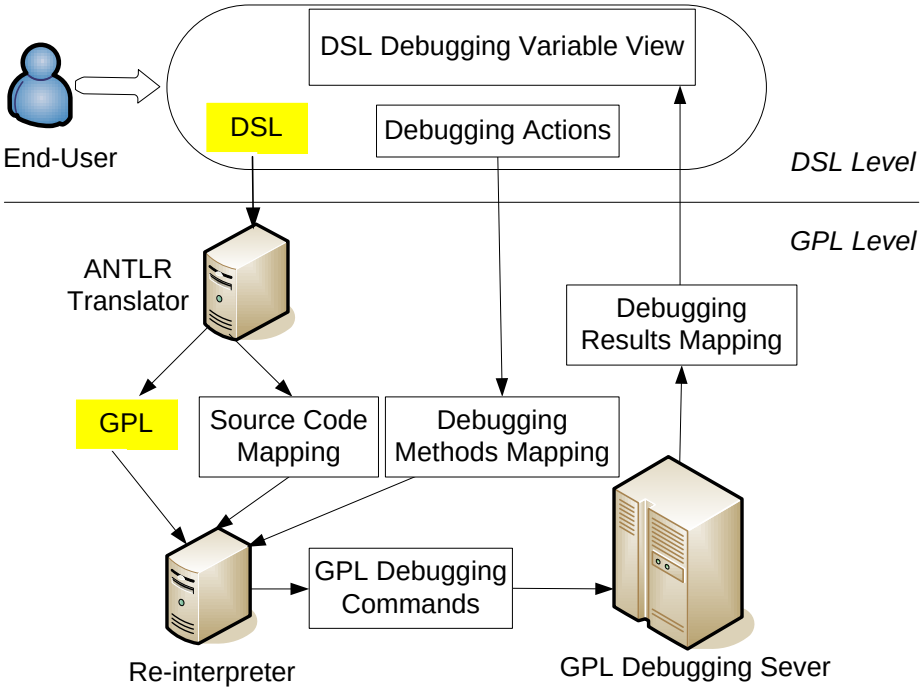


Figure 3-1. DSL Debugging Framework (DDF)

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 Eclipse debugging perspective by the *debugging results mapping component*, which is a wrapper interface to convert the GPL debugging result messages back into a form to be displayed at the DSL level. Because the messages from the GPL debugger are command-line outputs, which know nothing of the DSL or the Eclipse 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 3-2 illustrates the Robot DSL debugger generation process. In Figure 3-2, with the mapping generator embedded inside the grammar, the lexer and parser generated by ANTLR (step 1) takes the Robot DSL as input (step 2). ANTLR not only translates the Robot DSL program into the corresponding Robot.java, but also generates the

Mapping.java file (step 3). The mapping file represents a data structure that records all of the mapping information about which line of the Robot DSL code is mapped to the corresponding segment of Robot.java code. It indicates the location of the Robot.java code segment. Interestingly, the mapping information crosscuts the grammar in such a way that an aspect emerges within the grammar definition (please see Chapter 5) [Wu *et al.*, 2005].

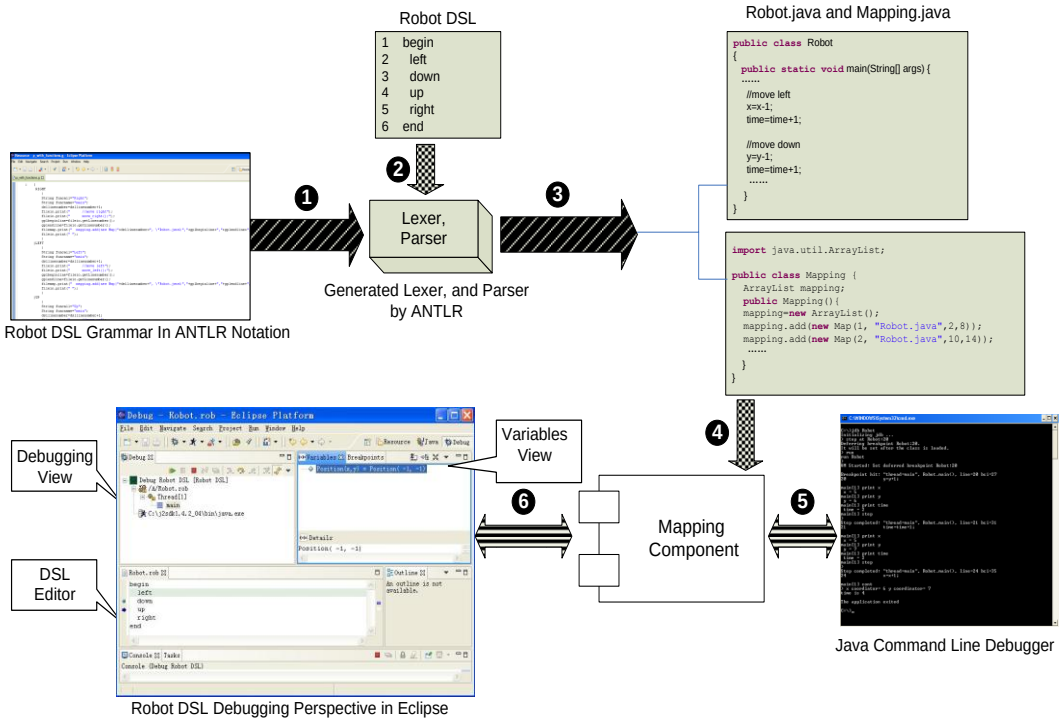


Figure 3-2. Debugger Generation Overview

The mapping component interacts and bridges the differences between the Eclipse debugger platform and the JDB (Java Debugger) (step 4). There are two round-trip mapping processes involved (step 5 and step 6) between the Robot DSL debugging perspective in Eclipse and JDB. A user issues debugging commands from Eclipse that are interpreted into a series of JDB commands against the Robot.java code. Based on the pre-

defined debugging mapping knowledge, the mapping component determines the sequence of debugging commands that need to be issued to the JDB at the GPL level.

3.2 Source Code Mapping

As a side-effect of the source-to-source translation process of the DSL pre-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 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 3-3 is the **`functionhead`** of knight); **`functionbody`** is the actual definition of a function (lines 4 to 6 on the left side of Figure 3-3 represent the **`functionbody`** of knight); **`functionend`** is the end of a function (line 7 on the left side of Figure 3-3 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 3-3 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 debugger perspective is independent of any GPL. Thus, the DDF can be used with any GPL that has an existing debugger.

```
…
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;
…
```

a) Robot DSL

```
…
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);}
…
```

b) Generated Java

Figure 3-3. Robot DSL Source Code Mapping

Variable mapping implicitly exists within the DSL compiler specified during the syntax-direct translation in the semantics specification. Figure 3-4 is part of the Robot DSL grammar specification that specifies the semantic actions taken on the implicit

position variable. This part of the grammar translates line 4 of the Robot DSL in Figure 3-3a into lines 7 and 8 of generated Robot.java in Figure 3-3b. The Robot DSL variable **position** is mapped to **x** and **y** variables in Robot.java. The translation of the **position** variable represents a one-to-many variable mapping, where one DSL variable is mapped to two or more GPL variables. These forward (i.e., from DSL to GPL) variable mappings are used implicitly by the DDF for generating the DSL debuggers.

```
Functionbody
 :(VARIABLES LPAREN op1:OP func_num1 :NUMBER COMMA op2:OP func_num2:NUMBER RPAREN
    {   funcall="functionbody";
        dsllinenumber=dsllinenumber+1;
        fileio.print(" x=x"+op1.getText()+func_num1.getText()+";");
        gplbeginline=fileio.getLinenumber();
        fileio.print(" y=y"+op2.getText()+func_num2.getText()+";");
        fileio.print(" time=time+1;");
        gplendline=fileio.getLinenumber();
         filemap.print("mapping.add(new
         Map("+dsllinenumber+",\"Robot.java\","+gplbeginline+","+gplendline+","+
              "\""+funcname+"\""+","+"\""+funcall+"\""+"));");
    }
 )
 ;
```

Figure 3-4. Part of Robot DSL Grammar Specification

Figure 3-5 is part of the SWUL grammar specification that specifies the semantic actions taken on JFrame variable assignments. This grammar fragment translates one SWUL statement (i.e., "**JFrame frame = frame;**") into 4 lines of the generated WelcomeSwing.java statements (e.g., lines 7, 9, 10, and 11) as indicated in Figure 3-5. The SWUL variable **frame** is mapped to the frame variables in WelcomeSwing.java. The complete DSL grammar specifications are available at Appendix A.3.

```
5   JFRAME IDENTIFIER ASSIGN IDENTIFIER LCURLY
6        {
7         fileio.print("jFrame_0 = new JFrame();");
8         jframename="jFrame_0";
9         fileio.print("JFrame frame="+jframename+";");
10        fileio.print("  frame.setSize(200,150);");
11        fileio.print("    frame.setVisible(true);");
12        }
13  frame    RCURLY   SEMI
14       )
15      ;
```

Figure 3-5. Part of SWUL Grammar Specification

3.3 Debugging Methods Mapping

The traditional debugging activities of a GPL include setting or clearing a break-point, stepping over, stepping into, terminating a debug session, and resuming execution [Rosenberg, 1996]. 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 most cases of the different types of DSLs defined in Chapter 2 (i.e., imperative, declarative, 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 3.6. The specifics of the debugging methods mapping are illustrated in Figure 3-6, 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 3-6, the second row indicates that DSL line number *n_i* is mapped to a segment of GPL code from line number *m_i* to *m_j*, 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_... n_i n_i+1 n_... n_j n_j+1 n_...	Line Number: m_1 m_2 m_... m_i m_i+1 m_... m_j m_j+1 m_...
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 3-6. 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. When an end-user steps through the DSL code to examine the values of DSL variables, the underlying GPL debugger acts differently to simulate the step through debugging action at the DSL level. During a stepping action within the DDF, the DSL debugging **`Step Over`** algorithm is invoked (see Figure 3-7). This algorithm requires information about the function types and function names that were generated from the DSL grammar. 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 function call 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 3-7. DSL Debugging Step Over Algorithm

In Figure 3-7, 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 code 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 another 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 cast 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 dissertation, the different meanings of **`Step Into`** and **`Step Over`** for a declarative DSL require minor adjustments in this algorithm to handle new requirements, which are described in Section 3.5.2.

3.4 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.

3.4.1 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 can be understood properly. 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 method handles 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 results mapping may exist within the DSL compiler such that the DDF can directly use the interface functions that the compiler provides to reveal variable values.

3.4.2 Debugging Results Mapping Example

Figure 3-8 shows the specification of the **`INIT`** production 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 re-interpreting 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 retrieve the value of the **`x_coordinate`** from JDB is "**`print x`**." The debugger variable view retrieves the result mapping from JDB and displays the position values at the Robot DSL level.

```
1    | INIT var:VARIABLES LPAREN init_num1:NUMBER COMMA init_num2:NUMBER RPAREN
2     {
3       dsllinenumber=dsllinenumber+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("+dsllinenumber+",\"Robot.java\","+gplbeginline +
12      ","+gplendline+","+"\""+funcname+"\""+","+"\""+funcall+"\""+"));");
13      fileresult.print(var=var.getText()+"(x_coordinate,y_coordinate)");
14     }
```

Figure 3-8. Debugging Result Mapping for the **`INIT`** Production of the Robot Grammar

The debugging results mapping is stored in one central location (called **`fileresult`**) where the DDF framework can access this information while automatically generating a DSL debugger for a specific DSL (e.g., Robot language, FDL, BNF, and SWUL).

3.4.3 Crosscutting Grammar Concerns

A crosscutting concern emerges from the addition of the explicit mapping in each of the grammar productions. For example, in Figure 3-8 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. A contribution of this dissertation research is described in Chapter 5, which demonstrates how an aspect language for grammars can assist in separating the various testing tool concerns for a specific grammar [Wu *et al.*, 2005]. Using a program transformation technique, an aspect-oriented language was developed called AspectG that can weave aspects into DSL grammars. The detailed description of AspectG is provided in Chapter 5.

3.5 Illustrative Examples

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).

3.5.1 Generation of an Imperative DSL Debugger

This sub-section describes an imperative debugger for the Robot DSL introduced in Section 2.3.1 that is 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 Java command-line debugger [JDB, 2007] 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 [Wright and Freeman-Benson, 2004].

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 (i.e., 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 command to the debugger interpreter. The result returned from the JDB is stored in a variable called **`resultReader`**, which is then 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 (GNU Project Debugger). 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 the DDF, including the implementation of **DSLDebugTarget**, stays the same. The back-end of the DDF is modularized so that the concern of the debugging user interface is separated from the back-end specifics of the underlying GPL debugger.

Figure 3-9 shows the **stepinto** method defined in the **DSLDebugTarget** class. When a debugging event (e.g., **stepinto**) 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.

```
1    protected void stepinto() throws DebugException {
2      Map map;
3      dslrember = dsllinenumber;
4      map = (Map) mapping.mapping.get(dsllinenumber - 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                   dsllinenumber = i + 1;
11                   break;
12            }
13     }
14     dsllinenumber = dsllinenumber + 1;
15    }
```

Figure 3-9. **stepinto** function in **DSLDebugTarget**

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 of Eclipse 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

variables 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 3-10 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 <-1, 0>. 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).

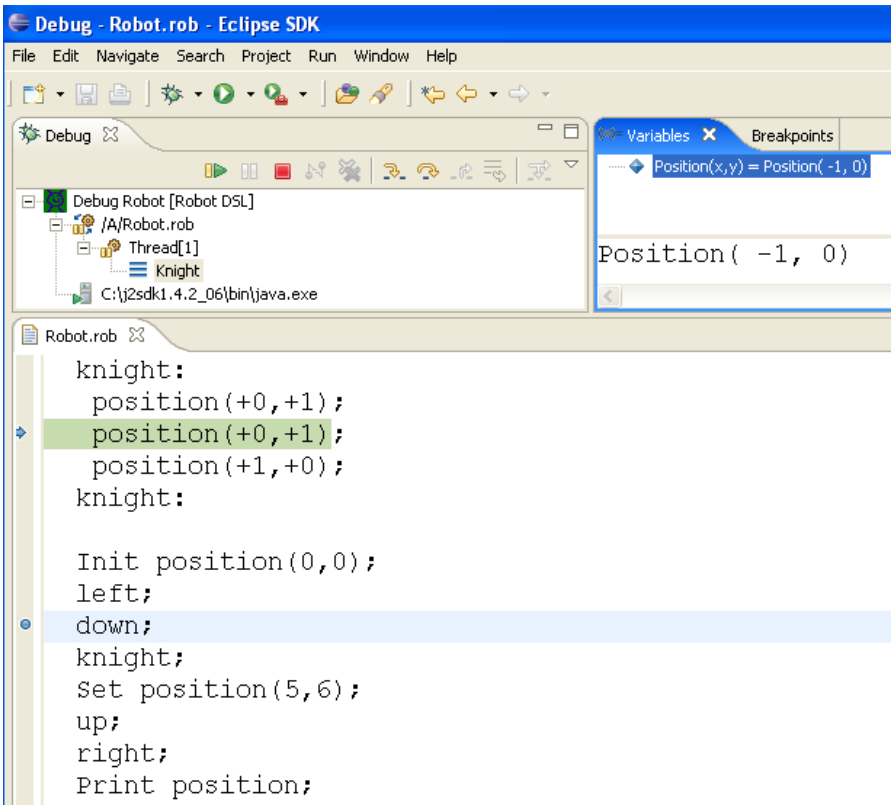


Figure 3-10. Screenshot of Debugging Session on Robot Language

3.5.2 Generation of a Declarative DSL Debugger

This section demonstrates the application of DDF to two separate declarative DSLs – FDL and BNF.

A Debugger for FDL

In addition to generating a debugger for an imperative DSL like the Robot language, the DDF can also generate a declarative DSL debugger for FDL (described in Section 2.3.2). The only modification to DDF is isolated in the component that maps the variable results back into the Eclipse debugging perspective. In the declarative DSL case, the variable mapping from the GPL to DSL is different from the imperative DSL case. In

this 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 2-4, stepping over each feature causes the resulting configuration to expand or contract (see the expanded variable view of the configuration in Figure 3-11).

The Eclipse debugging variable view is not able to display a Java object directly. A function 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. In the Car example, the car and feature variables are objects in the generated code. The **`DSLDebugTarget`** queries the state of the car value and the state of the feature values from the JDB. The state of the GPL variables is re-interpreted into a String representation so that the Eclipse variable view is able to display the current structure and contents of the Car feature diagram. This specific variable remapping (i.e., "**`variable = var.getText() + remap(feature);`**") is added into the DSL grammar to enable the re-interpretation of the raw GPL data returned from JDB. This mapping assists in reconstituting the DSL variable value. The function **`remap()`** is used to clear up 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 a variable (i.e., **`feature`**), (**`feature = "print var_name" +`**

`listnumber + ".toString()";`) is needed to query the values of Java variable **`feature`**, and assigned to **`variable`**, which are used to construct the value of the DSL variable.

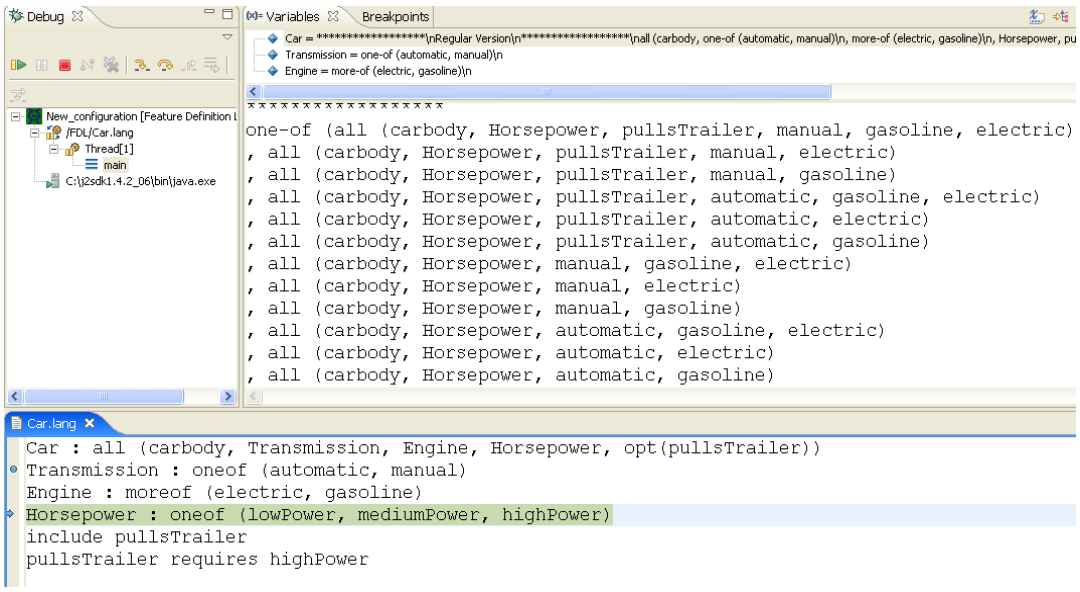


Figure 3-11. Screenshot of Debugging Session on Car Program

Figure 3-11 is a screenshot of a debugging session on the car specification introduced in Figure 2-4. 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 3-11 captures the instance of the **`Car`** feature diagram after evaluating three feature rules (e.g., **`Car`**, **`Transmission`**, and **`Engine`**). In this example, the program execution point has stopped at the fourth feature definition (**`Horsepower`**). An end-user can click a feature in the variable view and a detailed expansion view is provided (e.g., the

enumerated list of possible configurations in the middle of the figure). 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 detailed view shows all feature rule combinations that have been evaluated by the composition logic rules (e.g., **`one-of`**, **`more-of`**, and **`all`**). Notice that in the detailed view there are no **`Car`**, **`Transmission`**, and **`Engine`** composite features – they have all been expanded to their atomic parts from the evaluation of the first three feature definitions. The rest of the debugging perspective (e.g., stepping over rules) is similar to the Robot DSL debugger in Section 3.6.1.

A Debugger for BNF

In addition to generating a debugger for a declarative DSL like the FDL, the DDF can also generate a declarative DSL debugger for the Robot BNF (described in Section 2.3.2). The specific application of this debugger represents a tool integration demonstration with the Language Implementation System based on Attribute Grammars (LISA) [Mernik *et al.*, 2002] (see Section 4.7.3), which is a system to generate a parser, compiler, interpreter, and other language-based tools (e.g., finite state automata and visualization editor) from a language specification. LISA is used 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 [Aho *et al.*, 2007]. Compared to

the previous case studies in this dissertation, the more complex table-driven data structure is generated in Java that is equivalent to the syntax specification written in BNF based on the LR(1) parsing technique. 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 [DSL Testing Studio, 2007].

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. Considering as an example the Robot language specification from Figure 2-5, stepping over **`COMMANDS`** causes the current status of the parsing process to execute the **`COMMANDS`** definition. Figure 3-12 shows the Action and Goto functions of the Robot language parsing table for the Robot language. The action sub-table represents action functions of all the terminals of the Robot language and the special symbol "$" that is stored in the bottom of the stack. The Goto sub-table represents goto functions of all the non-terminals of the Robot language. The 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 DDF is used to retrieve the information from the parsing control table object (in Java) and translate is attributes into a primitive String type that can be displayed and understood by end-users through the DSL variables view.

	ACTION							GOTO		
State	begin	End	left	right	up	down	$	START	COMMANDS	COMMAND
0	s2							1		
1							accept			
2		r2	s6	s7	s8	s5			4	3
3		r2	s6	s7	s8	s5			9	3
4		s10								
5		r6	r6	r6	r6	r6				
6		r3	r3	r3	r3	r3				
7		r4	r4	r4	r4	r4				
8		r5	r5	r5	r5	r5				
9		r1								
10							r0			

Figure 3-12. Action and Goto table of Robot Language LR-parsing

Figure 3-13 is a screenshot of a debugging session of the Robot BNF. The lower-left of the figure contains the grammar editor, which also indicates the breakpoints and current program pointer. The lower-right of the figure contains the input language editor that 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 3-13, 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 a **`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 to the number 10 state into the current parsing stack). In this case, according to the action in row 4 and column **`end`** of the action field of Figure 3-12, the current operation is **`Shift10`**, meaning shift by pushing state 10 on to the stack, and remove **`end`** from the input. At this point, 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 all variable values 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.

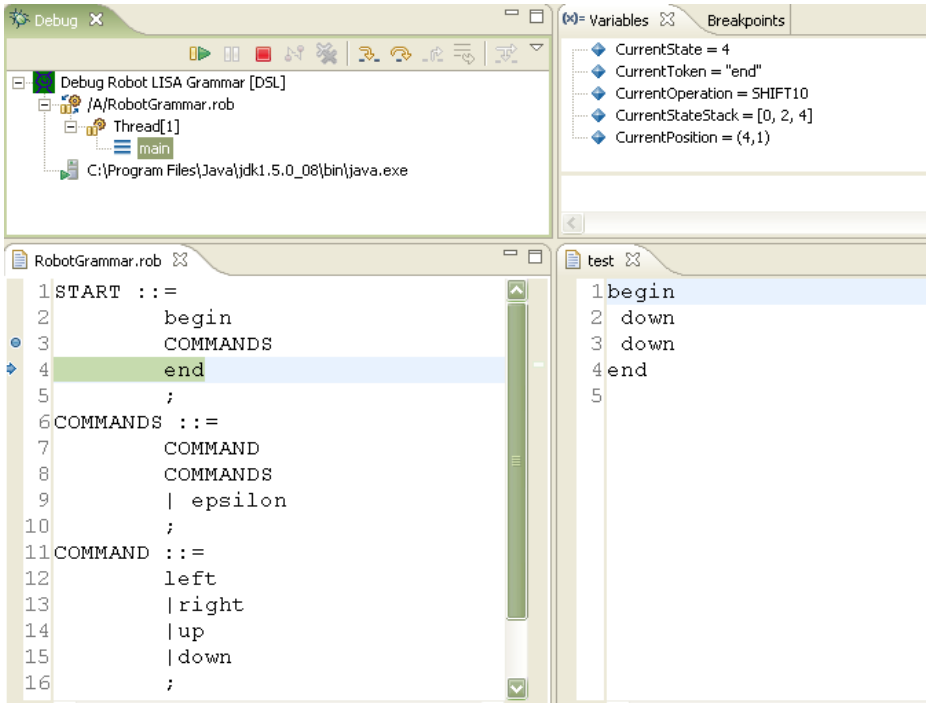


Figure 3-13. Screenshot of Debugging Session on Robot BNF

Another version of a BNF debugger can hide the bottom-up parser implementation details (e.g., current state, current operation, and current state stack) from end-users who are not familiar with this parsing technique. In such case, the debugging perspective only shows the current token, current token location, trace of consumed tokens, and a trace of productions. Figure 3-14 is a screenshot of this second version of the BNF debugger. In this figure, the current program execution point stopped at line 4 after the **`Step Over`** debugging action from line 3, which indicates that the **`COMMANDS`** non-terminal has been executed. A **`ProductionTrace`** variable in the variable view provides a detailed expansion view of the value of such tracing (e.g., a trace of the production flow history, which indicates all the grammar productions executed up to this execution point). The **`ConsumedTokensTrace`** view shows all input tokens that are consumed by the grammar productions up to this execution point, which can help language designers to validate the Robot language BNF syntax specification.

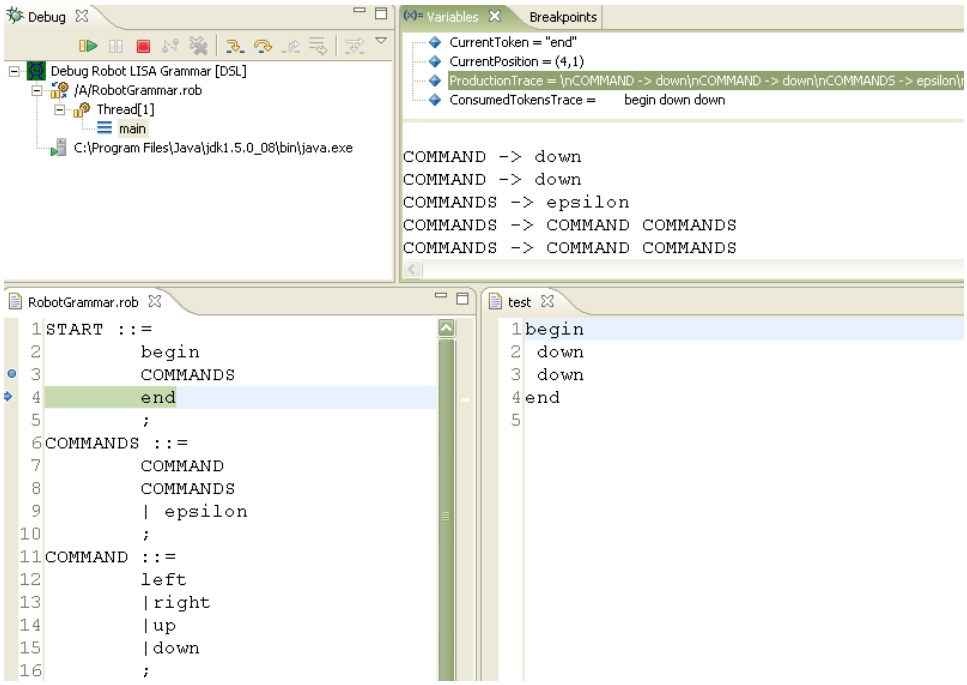


Figure 3-14. Screenshot of another Version of Debugging Session on Robot BNF

3.5.3 Generation of a Hybrid DSL Debugger

This section demonstrates the application of DDF to two separate hybrid DSLs – the SWUL and the hybrid Robot language.

A Debugger for SWUL

This section uses the SWUL hybrid DSL (as described in Section 2.3.3) to illustrate the generation process of creating a hybrid DSL debugger from a DSL grammar. An identifier “dsl” is used as a prefix 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, but without the prefix, the Java variable will be displayed). The variable name is also determined by the sixth field of this mapping information (i.e., **`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 retrieve 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.

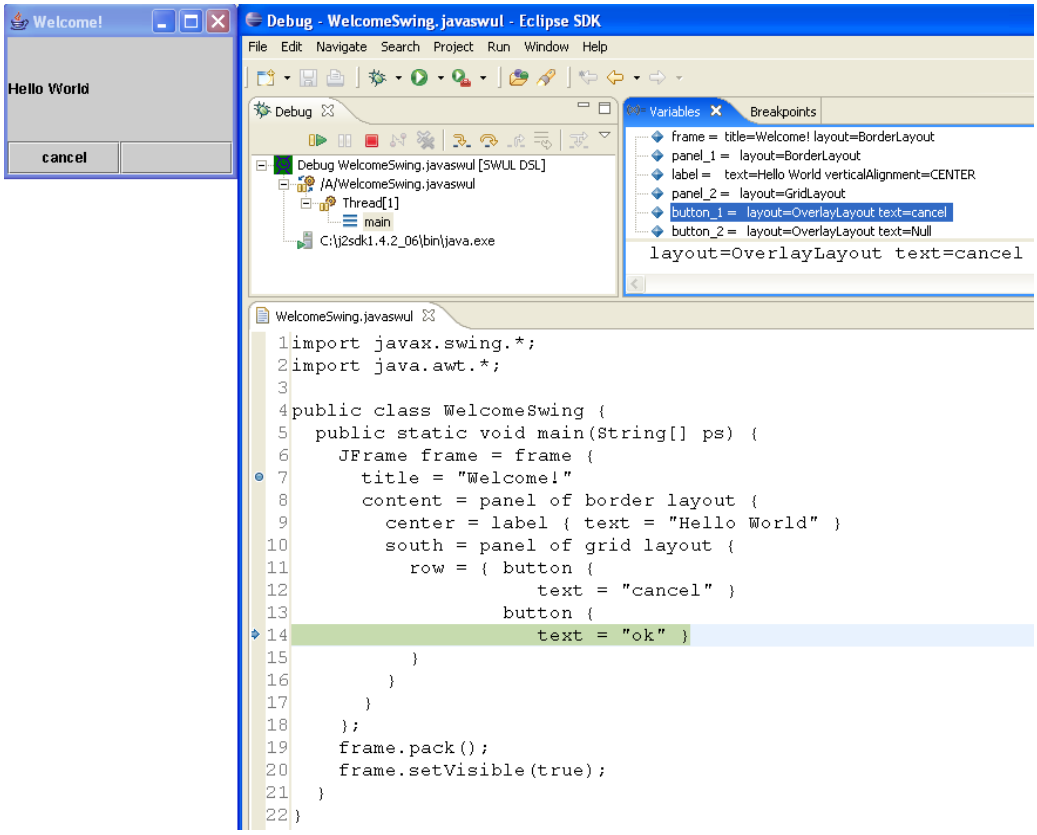


Figure 3-15. Screenshot of Debugging Session on SWUL Program

In order to generate the SWUL debugger, a modification was needed to the **`Step Over`** action (i.e., an addition is 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 3-15 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 code. The step method is 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. In summary, when debugging the Java part of a hybrid DSL, the JDB results are passed back to the debug perspective as a one-to-one mapping.

Figure 3-15 is a screenshot of the debugging session for a hybrid SWUL program that uses an escape block (e.g., lines 7 through 18) to obtain a structural description of user interfaces. As the program counter 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 variable view of the debugging perspective (shown in the upper-right of Figure 3-15), the individual components (e.g., frame, label, and button) and their associated attributes (e.g., location and contents) are displayed.

A Debugger for the Hybrid Robot Language

Figure 3-16 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. After the user enters the upper bound in the text box, the program pointer will stop at the very next line (i.e., the line that translates the value into the Java **`max`** variable). In the variable view of the debugging perspective (shown in the top-right of Figure 3-16), there are two sets of views available to the user. The top-most view shows the variable value of the **`position`** variable at the DSL level. The bottom-most variable

view displays the local variables of the embedded Java code, including all primitive variables (e.g., **`String answer`** and **`int max`**) and object variables (e.g., **`JOptionPane myGUI`** and **`Random rand`**). 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.

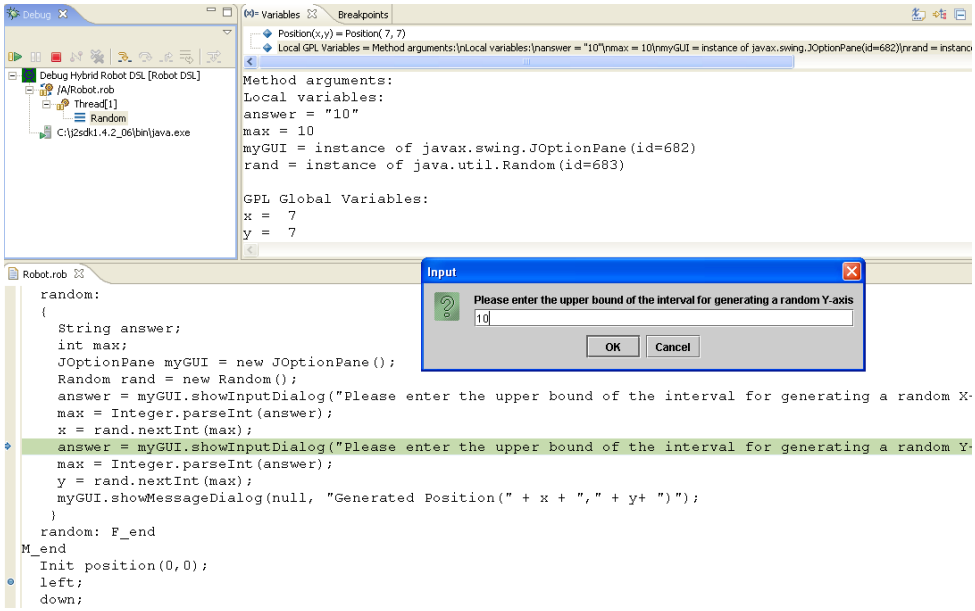


Figure 3-16. Screenshot of Debugging Session on Hybrid Robot Program

3.6 Case Study Evaluation

During the experimental evaluation phase of the DDF, it was observed that there are generic and specific parts in the debugger generation process [Wu, 2006]. The architecture of the DDF framework and the debugger generation processes are generic parts of the automated tool generation procedure that can be reused across different debuggers for these three categories of DSLs. The debugging action algorithms (e.g., **`Step Into`** and **`Step Over`**) are suited for most of the DSL cases in this research. These algorithms require minor modifications for the BNF debugger case. For different

types of DSLs, the two specific parts of the DDF are the source code mapping component and the debugging results mapping component, which are represented as several customized components in the DDF.

3.6.1 Generalization of DDF Usage

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, an important measurement is to assess the amount of code that is 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, it has been observed that less than 150 lines of code are needed for each new debugger generation. The comparison presented in Table 3-1 indicates that the amount of code needed to generate a DSL debugger is relatively small when using DDF.

DSL Category	*DSL Name*	*Number of Specific Functions or Classes*	*Customized Lines of Code*
Imperative DSL	Robot Language	2	69
Declarative DSL	FDL	3	89
	BNF	5	261
Hybrid DSL	Hybrid Robot Language	4	117
	SWUL	5	152

Table 3-1. Generality Analysis of DDF

In Table 3-1, column 3 shows the number of specific software components (e.g., functions or classes) needed to generate each DSL debugger. Across the five example case studies, among the 19 reusable software components there are two components (e.g., source code mapping component and debugging results mapping component) that need modification to adapt the specificity of different DSLs. Another component that cleans up the debugging results is needed for generating FDL, BNF, and SWUL debuggers. To generate the BNF debugger, DDF also needs to add two additional functions to handle its complicated **`Step Over`** and **`Step Into`** debugging actions. The two extra software components needed to generate Hybrid DSLs (e.g., Hybrid Robot language and SWUL) deal with the display of the local and global variables in the variables view.

The choice of the GPL debugger depends on the kind of GPL code generated from the DSL implementation. The GPL debugger performs the debugging actions on the generated GPL code. If DSLs are implemented through translation to different types of targeted GPLs other than Java, DDF has to change its underlying GPL debugger. Different GPL debuggers have their own supported GPLs and different interfaces with users. The re-interpreter is the one component that plays a specific role to adjust the

variability in this framework. If the underlying GPL debugger changes, the wrapper interface of the re-interpreter must be modified to adapt the differences among the specific GPL debuggers. This specificity only depends on the language types that the DSL program is translated to.

Currently, all the DSL examples used in this research are translated to Java, so the underlying GPL debugger is JDB. If the DSL programs are translated into C, C++, Objective-C, or Pascal, the GDB be can used, which provides general debugging support for various GPLs. The GDB "allows you to see what is going on 'inside' a program while it executes or what a program was doing at the moment it crashed" [GDB, 2007]. If the DSL programs are translated into C#, VB.Net, managed C++ and J# in .Net, then the Cordbg debugger can be used. This .Net debugger "helps tool vendors and application developers find and fix bugs in programs that target the .NET framework common language runtime. This tool uses the runtime Debug API to provide debugging services. Developers can examine the code to learn how to use the debugging services" [Cordbg, 2007].

The DDF provides software developers the freedom of choosing the underlying GPL debugger according to the generated GPL program. Table 3-2 shows the comparison of the three GPL debuggers (i.e., JDB, GDB, and Cordbg). The table lists only five of the basic commonly used debugging commands (e.g., set a breakpoint, step over, display value, terminate, and resume) which were used in the DDF. According to the comparison table in Table 3-2, there are several similarities among the syntax and semantics among these debuggers. The JDB is the simplest debugger among the three. Compared to JDB and GDB, Cordbg offers more sophisticated debugging functions and features (e.g.,

setting watchpoints, examining the complex data structures and native machine memory storage locations, and debugging programs with multiple processes).

Debugging Actions	***JDB (Java)***	***GDB (C++)***	***Cordbg (C#)***
Set a Breakpoint	stop at class: *line number*	break *filename*:*linenumber*	b[reak] [[*file*:] *line number*] | [[*class*::] *function* [:*offset*]]
Step Over	Step	step [*count*]	n[ext] [*count*]
Display Value	print class.staticfield	print *expr*	p[rint] [*variable name*]
Terminate	Exit	quit	exit
Resume	Cont	continue [*ignore-count*]	cont [*count*]

Table 3-2. JDB, GDB, and Cordbg Basic Debugging Commands Comparison

The manner in which DDF uses the underlying GPL debuggers is to call the **`sendRequest`** method and pass the actual GPL debugging commands as parameters. In order to make the DDF work with these three different GPL debuggers, the developers must change the parameters (i.e., different debugging command syntax) for **`sendRequest`** methods in the DDF as shown in Table 3-3. The breakpoint line number from the source code level debugging comes from a method called **`getGplbegin`**, which returns the correct result from a data structure that stores all of the mapping information. This approach decreases the cohesion with GPL debuggers and provides the developers with further extension opportunities to make the DDF more powerful and useful by utilizing the functionalities of the full-fledged GPL debuggers. Because the various GPL debuggers have different output formats, the **`remap`** sanitization method requires minor adjustments to handle the debugging results format differences.

JDB (Java)	`sendRequest("stop at Robot: " + map.getGplbegin());`
GDB (C++)	`sendRequest("break Robot.cpp: " + map.getGplbegin() );`
Cordbg (C#)	`sendRequest("break Robot.cs: " + map.getGplbegin() );`

Table 3-3. The DDF Adaptation for JDB, GDB, and Cordbg

For different types of IDEs (e.g., Eclipse or .Net), DDF has to change the way its components interact within different plug-in architectures in the IDE. The current focus has considered Eclipse as the target IDE, with the DDF plug-ins implemented in Java. A switch to Microsoft's Visual Studio .Net IDE would require changing the language to implement the plug-ins (e.g., C#, VB.Net, C++ and J#). The way plug-ins are implemented is different between IDEs. Furthermore, GPLs have different features for constructing plug-ins. These limitations to the generality of DDF are considered in Chapter 6, which addresses future work.

3.7 Related Work in the Area of Domain-Specific Language Debuggers

The End-Users Shaping Effective Software (EUSES) Consortium represents collaboration among several dozen researchers who aim to improve the software development capabilities provided to end-users [EUSES, 2007]. A contribution of EUSES is an investigation into the idea of "What You See Is What You Test" (WYSIWT) to help isolate faults in spreadsheets created by end-users [Ruthruff *et al.*, 2006]. More specific to the focus of this research, this section provides an overview of related work in the area of DSL debuggers (e.g., Khepera, JSR-045, ANTLR Studio, and TIDE).

3.7.1 Khepera

Khepera is a toolkit for the rapid implementation and long-term maintenance of DSLs [Faith *et al.*, 1997]. 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 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 in this dissertation 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 [Faith, 1998]. 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. The 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 a pre-processor piggyback approach is adopted in DDF, much of the optimization in our framework is passed on to the compiler for the targeted GPL.

3.7.2 JSR-045

JSR-045 represents the "Java Specification Request for Debugging Support for Other Languages," which establishes standardized debugging tools for correlating Java bytecode to the source code of languages other than Java [JSR 45, 2007]. Although JSR-045 has a source code line number mapping mechanism similar to DDF, JSR-045 does not have a test 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) [JPDA, 2007]. 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, 2007]), which can be replaced easily by other command-line debuggers (e.g., [GDB, 2007]) if the target GPL changes.

3.7.3 ANTLR Studio

ANTLR Studio is an Eclipse plug-in for ANTLR [ANTLR-Studio, 2007]. 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-user's 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.

3.7.4 TIDE

Olivier designed the ToolBus Integrated Debugging Environment (TIDE) [Olivier, 2000], which is implemented within the ASF+SDF Meta-Environment [van den Brand *et al.*, 2002; van den Brand *et al.*, 2005]. Using TIDE, an additional framework has to be constructed to provide an interactive debugger for the user. ToolBus represents middleware written in C to enable component-based communication among tool environments [Cornelissen, 2004]. 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. The DDF adopts standard compiler implementation methods (e.g., lexer and parser) as the front-end of the framework. 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).

3.8 Summary

As the number of end-user programmers rises substantially each year, 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 [Gelperin and Hetzel, 1988].

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. The lack of debugging support for DSLs forces domain experts to debug their DSL applications at the GPL level, which provides a semantic gap between the notations they expect and the tools that are provided. This may reduce the productivity and accuracy of the debugging process for end-users. To help address this problem, this dissertation introduced a grammar-driven framework that automates the generation of debugging tools for DSLs. This chapter presented five different examples of debuggers for three different DSL categories. With the proper training and availability of DSL debuggers, domain experts can solve their problems much more effectively [Wile, 2004]. More details about the research described in this chapter, including video demonstrations and complete examples, can be found at the project website [DSL Testing Studio, 2007]. As shown in Figure 1-2, the tool architecture

used in DDF has been applied to a similar framework to assist in the generation of DSL unit test engines, which is described in Chapter 4 of this dissertation.

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 pre-processor DSL implementation patterns), 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 or 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 chapter 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 3.3. 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 3.4. In many cases, it is possible to reuse an existing DSL compiler implementation to realize this mapping.

This dissertation makes a contribution in the area of Grammarware [Klint *et al.*, 2005] 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 chapter also demonstrated the potential for reusing existing GPL language tools through grammar-driven automation. 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 [Wu *et al.*, 2005], as discussed in Chapter 5.

CHAPTER 4

DSL UNIT TESTING FRAMEWORK (DUTF)

This chapter represents a grammar-driven approach that leverages an existing unit testing tool to generate unit test cases. These test cases directly exercise the end-user's test intention. This technique has been developed to build a unit test engine generation framework from existing DSL grammars, and applied on different DSLs taken from a variety of sources. Two case studies are presented to illustrate how unit test engines are generated by the DSL Unit Testing Framework (DUTF). It will be shown that this approach can provide unit testing tool support for DSLs at the proper abstraction level. To demonstrate the benefits of this approach, experimental evaluation is discussed, including generality analysis to assess how the framework can be used to generate test engines for diverse categories of DSLs. Related work and a concluding discussion are also presented at the end of this chapter.

4.1 DUTF Architecture Overview

As observed from traditional software development, unit testing supports early detection of program errors, and the complementary process of debugging helps to identify the specific location of the program fault to reduce the cost of software failures [Olan, 2003]. To complement the DDF, the DUTF assists in the construction of unit test

cases for DSL programs, much in the sense that JUnit is used in automated unit testing of Java programs. After identifying the existence of an error using DUTF, the DDF can then be used to identify the fault location within the DSL program. The DUTF framework invokes the underlying GPL unit test engine (e.g., JUnit [JUnit, 2007] or NUnit [NUnit, 2007]) to obtain unit test results that are remapped onto the abstractions of the domain represented by the DSL. The key mapping activities in DUTF are the translation of the DSL unit test script into GPL unit test cases and interpretation of the GPL unit test results into DSL unit test notations. In the DUTF, the reports of passed and failed test cases appear at the DSL level instead of the underlying GPL level. A failed test case reported within the DUTF reveals the presence of potential errors in the DSL program.

An illustrative overview of the DUTF is shown in Figure 4-1, which has a similar architecture to the DDF. The ANTLR translator generates GPL code from DSL source code, generates GPL unit test cases from DSL test cases, and generates the source code mapping information for both GPL and unit test generation components. The results from the mapping components are re-interpreted into the GPL unit test engine as unit test cases that are executed against the translated GPL code. The *test cases mapping component* uses the generated mapping information to determine which DSL test case is mapped to the corresponding GPL unit test case. The mapping indicates the location of the GPL test case corresponding to a single test case defined in a test script at the DSL level. The test cases mapping component considers the user's test cases at the DSL level to determine what test cases need to be created and executed by the underlying GPL unit test engine.

The GPL unit test engine executes the test cases generated from DSL test scripts. Because the messages from the GPL unit test engine are expressed in a GPL, the test

result at the GPL level is sent back to the DSL test result view by the *test results mapping component*, which is a wrapper interface to remap the test results back into the DSL perspective. The domain experts only see the DSL unit test result view at the DSL level.

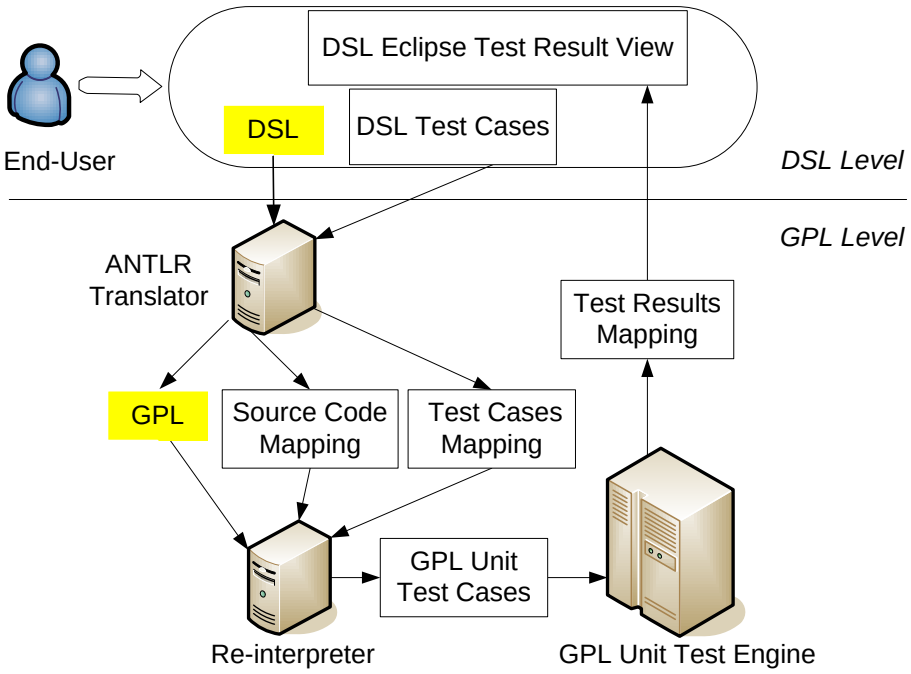


Figure 4-1. DSL Unit Testing Framework (DUTF)

An example of the Robot DSL unit test engine generation process is shown in Figure 4-2. The semantic actions associated with the source code generation use syntax-directed translation and additional semantic functions in the grammar specification to generate the mapping information. In Figure 4-2, with the mapping generator embedded inside the grammar, the lexer and parser generated by ANTLR (step 1) takes the Robot DSL as input. ANTLR not only translates the Robot DSL into the corresponding Robot.java, but also generates the Mapping.java file (step 2). At the same time, another translator generates the JUnit test case (e.g., TestRobot.java) from the Robot DSL unit

test script and another mapping file. The mapping file represents a data structure that records all of the mapping information about which line of the Robot DSL unit test cases is mapped to the corresponding JUnit test cases in the generated TestRobot.java code. A DSL unit test case is interpreted into a JUnit test case against the generated Robot.java code. At the GPL level, the generated JUnit test cases represent the unit testing intension of Robot unit test cases.

The mapping component interacts and bridges the differences between the Eclipse DSL unit test perspective and the JUnit test engine (step 3). There are two round-trip mapping processes involved (step 4 and step 5) between the Robot DSL unit test perspective in Eclipse and JUnit.

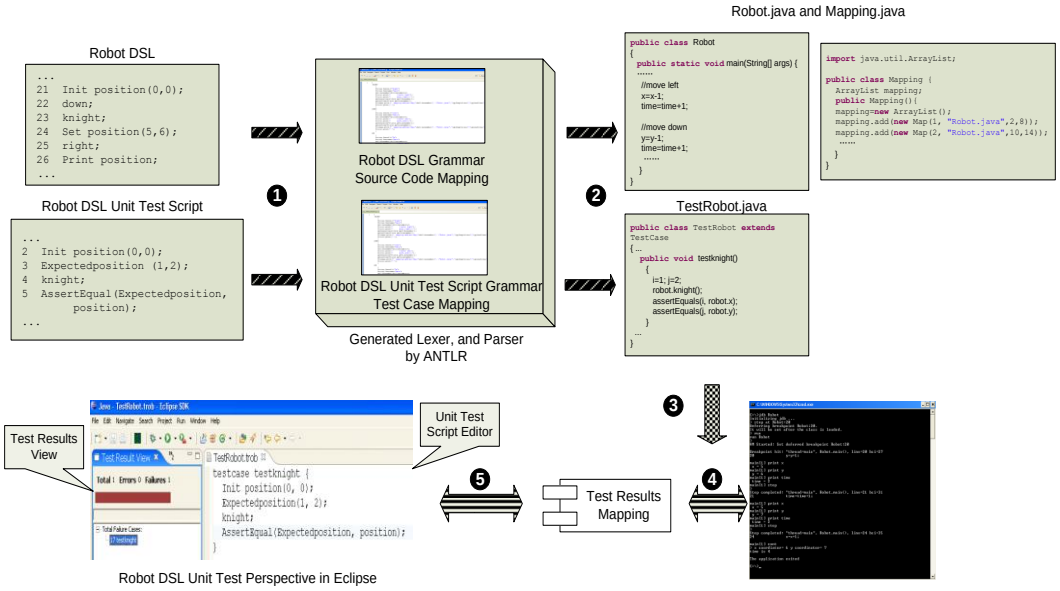


Figure 4-2. DSL Unit Test Engine Generation Process

4.2 Source Code Mapping

Along with the basic functionalities translated from a DSL to its equivalent GPL representation, the syntax-directed translation process also can produce the mapping information augmented with additional semantic actions embedded in the DSL base grammar. ANTLR is used to translate the Robot language to a GPL (e.g., Java) using a process that is similar to the DDF described in Section 3.2. The DUTF also generates GPL unit test cases and the test case mapping hooks that interface with the DUTF infrastructure. This mapping information is provided as additional semantic actions in the Robot language grammar specification and is used for generating unit test cases. The mapping usually contains the DSL unit test case line number, the translated GPL unit test file name, and the line number of the corresponding unit test case in the GPL.

4.3 Test Cases Mapping

The abstraction mismatch between DSLs and GPLs also contributes to the mismatch in construction of unit test cases. When translating DSL unit test cases, a variable in a unit test case at the DSL level may not have an equivalent mapping to a single variable in a GPL (i.e., a DSL variable may be translated into several variables, even objects, in the generated GPL). The presentation format of the DSL variable may also differ from the GPL representation. In the case of DUTF, the DSL unit test script is mapped to the corresponding GPL unit test cases by a test case translator that was also written in ANTLR. In the DUTF, the generated GPL test cases are exercised by the underlying GPL unit test engine (e.g., JUnit or NUnit). The main task of a unit test is to assert an expected variable value against the actual variable value when the program units

are executed. The test case mapping from a DSL program to the corresponding GPL is used to construct the translation of test cases at the GPL source code level. The base grammar of the unit test script is augmented with additional semantic actions that generate the test case line number mapping.

In the following discussion, the Robot imperative DSL (described in Section 2.3.1) is used as the target DSL. Figure 4-3 shows the mapping from a Robot DSL unit test case (called **`testknight`**) to a corresponding JUnit test case of the same name. In the Robot DSL unit test script, line 2 on the left side is mapped to lines 2 and 3 on the right side (initialize the position as <0, 0>); line 3 on the left side is mapped to lines 4 and 5 on the right side (sets the expected position as <1, 2>). One assertion statement in the Robot DSL unit test script may be translated into two or more separate assertion statements in JUnit due to the mismatch of variables between the DSL and GPL. For example, the variable called **`position`** in the Robot DSL is translated into two variables (**`x`** and **`y`**) in Robot.java; line 5 (left side of Figure 4-3) is mapped to lines 7 and 8 (right side of Figure 4-3). One assertion of Robot variable **`position`** is mapped into two assertions of Robot.java integer variables **`x`** and **`y`**. The one-to-many test case results must be remapped by a corresponding many-to-one mapping back into the DSL view, which is described in Section 4.4.

Robot DSL Unit Test Case

```
1  testcase testknight {
2    Init position(0,0);
3    Expectedposition(1,2);
4    knight;
5    AssertEqual (Expectedposition, position);
6  }
…
```

GPL Unit Test Case (JUnit)

```
1  public void testkinght() {
2       robot.x = 0;
3       robot.y = 0;
4       int x = 1;
5       int y = 2;
6       robot.move_knight();
7       assertEquals(x, robot.x);
8       assertEquals(y, robot.y);
9  }
…
```

Figure 4-3. Robot Language Test Cases Mapping

The Car declarative FDL (described in Section 2.3.2) is used as another target DSL case study. Figure 4-4 shows the mapping from a Car FDL unit test case (called **`testFeatures`**) to a corresponding JUnit test case of the same name. In the Car FDL unit test script, line 2 on the top of Figure 4-4 is mapped to the JUnit test case at the bottom. In this figure, the expected car features are from lines 12 to 14, where three specific features (e.g., **`carbody`**, **`manual`**, **`highPower`**) are desired features. Line 3 invokes a unit of the original Car FDL program that executes all four features defined in the Car FDL program; line 4 invokes a constraint that requires every car feature combination list to include a **`pullsTrailer`**. The parse function used in line 27 of the JUnit test case is a helper function that stores the output of the car Java program into an organized data structure, and then converts it to the same class type as the current tested car's feature called **`testFeatures`**. The **`compareFeatures`** function used in line 27 of the JUnit test case is another helper function that compares the two parameters. The traditional JUnit built-in assertion functions (e.g., **`assertEquals`**) are not applicable and not capable of handling the particular scenarios in FDL that compare car features.

This limitation is due to the fact that the order of the car's features written in the FDL test case script is irrelevant. However, the **`assertEquals`** assertion in JUnit will report an error if two objects are not exactly equal. In other words, the order of the features of the current car and expected car are not equal. Even if the contents of these two objects are equal the result is still false. However, at the Car FDL abstraction level, they are equal. To address this issue, a method called **`compareFeatures`** has been built to handle this situation where only the contents matter and the ordering issue can be ignored.

In Figure 4-4, line 6 of the Car FDL unit test case is mapped to line 28 of the JUnit test case. This line represents an assertion to assess the number of possible valid feature combinations. The **`getFeatureListNumber`** function retrieves the number of feature combinations from the parsed data structure. It is not possible to obtain the size of a feature list because the existing FDL compiler does not provide such a method, so a helper method was needed. The **`assertEquals`** statement is used to compare the actual feature list size with the expected feature combination number.

In this case, one assertion statement in the Car FDL unit test script is translated into one assertion statement in JUnit. This one-to-one test case assertion mapping is simpler than the one described in the Robot unit test engine case but the comparison function is more complicated than the previous case. JUnit does not support the sophisticated assertion functionality that is needed for FDL unit testing. Such helper and comparison functions were needed to realize the unit test intention for FDL programs.

Car FDL Unit Test Case

```
1  TestCase testFeatures {
2  Expectedfeature:(carbody, manual, highPower);
3  use Car.FDL(All);
4  Constraint C1: include pullsTrailer;
5  AssertTrue(contain(Expectedfeature, feature));
6  AssertEqual(6, numberof feature);
}
```

GPL Unit Test Case (JUnit)

```
11  public void testFeatures () {
12        testFeatures.add("carbody");
13        testFeatures.add("manual");
14        testFeatures.add("highPower");
…
27  assertTrue(compareFeatures(testFeatures,parse(fc,root,cons)));
28  assertEquals(6,getFeatureListNumber(parse(fc,root,cons)));
…
```

Figure 4-4. FDL Test Cases Mapping

4.4 Testing Results Mapping

JUnit reports the total number of test cases, total number of failed test cases (i.e., those representing the expected values and the current tested values are not equal during test execution), and total number of error test cases (i.e., those representing run-time exception errors during test execution). If one test case in a DSL is translated into multiple test cases at the GPL level, mapping the many-to-one GPL unit test results back to the DSL intention can be challenging. According to the unit test concept, one test case's failure result should not affect other test cases. In order to get the final test result of one test case in the DSL, all corresponding GPL test cases have to be tested and analyzed before sending the results back to the DSL level. An algorithm is needed to process the test result back to the DSL level. However, a rather simple approach can solve this issue by imposing one-to-one mapping at the test case level (i.e., each test case specified at the

DSL level is translated into one test case at the GPL level) to take advantage of the relationship between test cases and assertions in the unit test concept and realize the many-to-one mapping at the assertion level. Due to the abstraction gap between the DSLs and GPLs, within each test case, instead of translating one test case in the DSL unit test script into many test cases in the GPL unit test cases, one assertion in a DSL test case may be translated into one or many assertions in a GPL test case. The one-to-many mapping that is encapsulated inside the individual test case makes the test result easier to interpret across the abstraction layers.

One failed assertion at the GPL level should result in an entire test case failure. Only those GPL test cases that have passed all assertions should result in a successful test case at the DSL level. Such assertion encapsulation and test case mapping also helps to determine the location of a failing DSL test case without considering the many-to-one consequence from the line number mapping. The test result from JUnit indicates the location of the failed test case in the JUnit code space, which is not helpful for end-users to locate the position of the specific failed test cases in their DSL unit test script. For simplicity, the test case names are kept the same during the translation process. By matching the test case name through the test case mapping information, when a generated GPL test case fails the corresponding line number of the DSL unit test script can be obtained from the JUnit test case line number mapping in the test result report to locate the failed DSL unit test case.

4.5 Illustrative Examples

This section illustrates the application of the DUTF on the Robot language and the FDL.

4.5.1 Generation of an Imperative DSL Unit Test Engine

This sub-section describes the generation of a unit test engine for the imperative Robot DSL introduced in Section 2.3.1. The DUTF adapts the JUnit Eclipse plug-in graphical user interface by adding a new view called the DSL Test Result View, which is similar to the Eclipse JUnit plug-in unit test result view, but representing the DSL abstraction. Figure 4-5 shows a screenshot of a Robot DSL unit test session. Adapting the JUnit test case construction concept, a DSL test case is composed of a test case name (e.g., **`testknight`**) and test body. The test body defines the expected value of a certain variable, the module to be tested, as well as the criteria for asserting a successful pass (e.g., **`AssertEqual`**). The Robot DSL unit test cases are specified in a unit test script, which itself is a DSL designed for the purposes of this dissertation reseach. The Robot DSL unit test script translator has been implemented in ANTLR to generate JUnit test cases from the DSL test script. The source code mapping for the Robot DSL unit test script is also generated by adding additional semantics to the base DSL grammar.

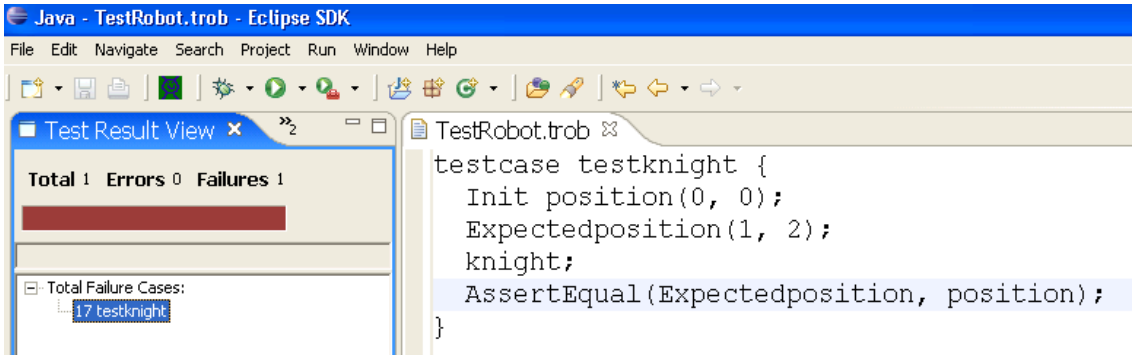


Figure 4-5. Screenshot of Unit Testing Session on Robot Language

The right side of Figure 4-5 is the DSL unit test script editor, which shows an actual Robot DSL unit test script called TestRobot.trob. The Robot language variable **`position`** is initialized to <0, 0>. The highlighted test case called **`testknight`** has an expected value that is set as position <1, 2>. The function unit to be tested is the **`knight`** move and the assertion criteria determines whether there is a distinction between the expected position and the actual position after **`knight`** is executed. An incorrect knight method is intentionally implemented as shown in the right side of Figure 4-6 (i.e., line 3 incorrectly updates the robot to position <+1, +1>). When the **`testknight`** test case is executed on the incorrect **`knight`** implementation, the expected position value (i.e., <1, 2>) is not equal to the actual position (i.e., <2, 2>).

Because the position variable in the Robot DSL is represented by two variables (i.e., **`x`** and **`y`**) at the GPL level, the **`testknight`** test case is translated into two JUnit test assertions. One assertion tests the value of the **`x`** variable, and another tests the **`y`** variable. If either of these two assertions at the GPL level fails, then the single test case at the DSL level is reported as a failure. In this **`testknight`** example, the assertion on the **`x`** coordinate will fail on the incorrect **`knight`** implementation, but the assertion to test **`y`**

will succeed. Consequently, the **testknight** test case is reported as a failure in the Test Result View on the left side of Figure 4-5.

The **AssertEqual** assertion in this DSL unit test script determines whether its two parameters are equal. The Test Result View also indicates the total number of test cases (in this case, it was 1), the total number of failures (in this case, there was 1 failure), and the number of error test cases (in this case, there were 0 errors causing run-time exceptions). The progress bar that appears in the Test Result View indicates there is at least one test case that failed (the bar actually appears red in failure or error cases, and green when all test cases are successful). The list of test cases underneath the progress bar indicates all the names of the test cases that failed to pass the test case (e.g., **testknight**).

Correct knight method	Incorrect knight method
1 knight: 2 position (+0,+1); 3 position (+0,+1); 4 position (+1,+0); 5 knight:	1 knight: 2 position (+0,+1); 3 position (+1,+1); 4 position (+1,+0); 5 knight:

Figure 4-6. Correct and Incorrect Knight Methods

The DUTF provides a capability that allows the DSL end-user to double-click on the test cases listed in the Test Result View, which will then highlight the specific test case in the editor view. Figure 4-7 is an example of the plug-in code that was written to interact with JUnit to handle the end-users double-clicking on the failed test case. The method searches through the source code mapping to find the selected test case name

(line 8) and then gets the line number (line 9) of the test case. This information is used to display the test script editor and highlight the clicked test case in the test script (line 13).

```
1  protected void handleDoubleClick(DoubleClickEvent dce) {
2    IStructuredSelection selection = (IStructuredSelection) dce.getSelection();
3    Object domain = (TestResultElement) selection.getFirstElement();
4    String casename = ((TestResultElement) domain).getFunctionName();
5    int linenumber = 0;
6    for (int i = 0; i < mapping.size(); i++) {
7      Map map = (Map) mapping.get(i);
8      if (map.getTestcasename().equals(casename)) {
9              linenumber = map.getDslnumber();
10     }
11   }
12   OpenDSLTestEditorAction action = null;
13   action = new OpenDSLTestEditorAction(this, "TestRobot.trob", linenumber);
14   action.run();
15 }
```

Figure 4-7. **`handleDoubleClick`** Method in **`TestResultView`** Class

4.5.2 Generation of a Declarative DSL Unit Test Engine

In addition to generating a unit test engine for an imperative DSL like the Robot language, the DUTF was also used to generate a declarative DSL unit test engine for the FDL (described in Section 2.3.2). The modification to DUTF was isolated in the component that translates the unit test script into unit test cases in the GPL, which are specified in the script grammar. Because of the domain-specific syntax of the FDL, the Robot language unit test script translator needed modification so that it could generate the correct unit test cases for FDL in Java. Also, another modification in the declarative DSL case is the test result mapping from the DSL to GPL, which is different from the imperative DSL case. In the Robot language the variable to be tested is **`position`** and in the Car FDL the variables are various features.

The right side of Figure 4-8 is the FDL unit test script editor, which shows an actual Car FDL unit test script called **`TestCar`**. The test case, called **`testfeatures`**,

has an expected value, called **Expectedfeature** that is set as {**carbody**, **manual**, **highPower**, **electric**, **pullsTrailer**}. The target unit to be tested is all the features (from feature 1 to feature 4 in Figure 2-4) plus one constraint (constraint 1 in Figure 2-4). Another assertion, called **AssertTrue**, can assess whether the tested unit will return true or false. If the tested unit returns true, the **AssertTrue** assertion will succeed, otherwise it will fail. An assertion is set to test whether the **Expectedfeature** is contained in the set of possible features after executing all the features and one constraint. The **contain** method is a helper function that compares the expected features and current features. In the left side of Figure 4-8, the Test Result View indicates that this assertion was a success, so **Expectedfeature** is one of the possible features after the execution of all test case features. Also, the order of the features in the **Expectedfeature** is not important in this case because the features can be either atomic or composite.

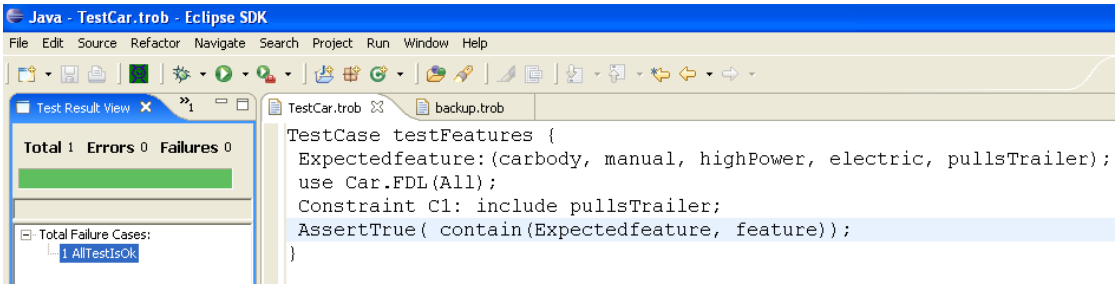


Figure 4-8. Screenshot of Unit Testing Success Session on Car FDL

Figure 4-9 is another test case called **testNumberofFeatures**, which is highlighted in the Test Result View. The targeted testing unit consists of three features

(from feature 1 to feature 3 in Figure 2-4). The **numberof** operator returns the size of a set of features. The meaning of the **AssertEqual** assertion is the same as described in previous sections. An **AssertEqual** assertion tests whether the number of possible features after expanding all these three composite features (e.g., **Car**, **Transmission**, and **Engine**) to atomic features, but not expanding others (e.g., Horsepower) is 6. In the left side of Figure 4-9, the Test Result View indicates this assertion fails. The set of features executed are **Car**, **Transmission**, and **Engine**. According to the individual feature definitions, there are two options for **Transmission** (**automatic** and **manual**), three options for **Engine** (**electric**, **gasoline**, and **electric/gasoline**), and two options for **pullsTrailer** (with or without). The total number of features is 12 (2*3*2) rather than 6, which causes the test case to fail as indicated in the DSL unit Test Result View of Figure 4-9.

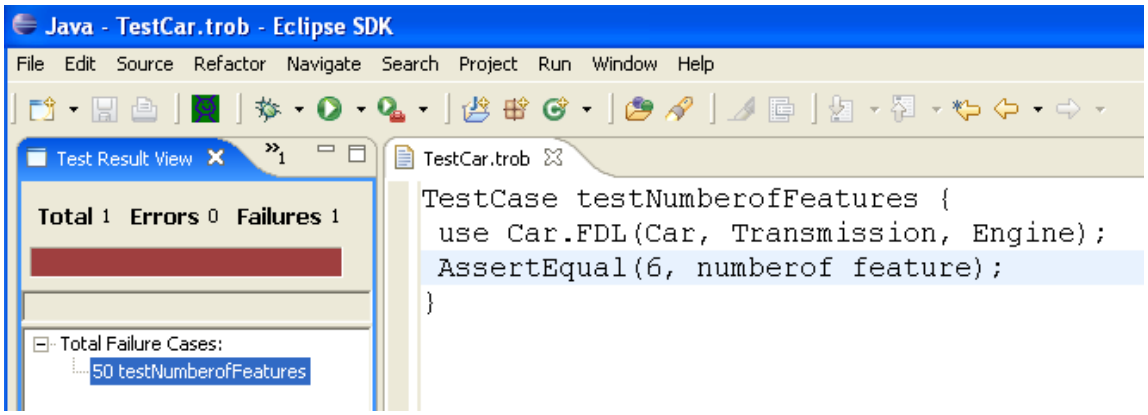


Figure 4-9. Screenshot of Unit Testing Failure Session on Car FDL

4.5.3 Generation of a Hybrid DSL Unit Test Engine

Hybrid DSLs have similar language characteristics as imperative DSLs. From the unit testing point view, it can be considered an extension of imperative DSLs. For example, the Hybrid Robot language is an extension of the Robot imperative DSL. For debugging support, DSL development tools should be able to debug hybrid DSLs by switching language modes between two different language domains. However, unit testing a hybrid DSL is no different from unit testing an imperative DSL; therefore, this section has focused on the generation of unit test engines for the Robot imperative DSL and FDL declarative DSL.

4.6 Case Study Evaluation

While designing DUTF, it has been observed that there are generic and specific parts of the unit test generation process. The architecture of the framework, the generation process, and the test result analysis algorithm are generic parts of the DUTF that can be reused across different unit test engines for the different types of DSLs. The source code and test cases mapping are the specific parts of this framework that may require customization for each DSL.

4.6.1 Generalization of DUTF Usage

To generate testing engines for different types of DSLs, or even the same type of DSLs but different instances of DSLs, one of the specific parts of the DUTF is the test cases mapping component, which is represented as a test script interpreter. The test script interpreter also utilizes the mapping information generated by the source code mapping that is identical to DDF. The description of the test cases mapping component among the different DSL unit test engines is presented in Section 4.3. Unlike the case studies in the DDF, the backward mapping from the JUnit test results to the DSL the test results is handled by the encapsulation of multiple assertions in a single test case, as described in Section 4.4

Various styles of DSLs served as evaluation artifacts to determine the generality of this grammar-driven approach by comparing the generation of different DSL unit test engines. A total of two different unit test engines for two types of DSLs were generated using DUTF. Several quantitative measurements were observed to analyze the amount of effort required to generate new unit test engines. In particular, this section addresses the

question, “How many of the generic software components from DUTF can be reused without modification or small changes, as compared between different types of DSLs?” The amount of code that was written for each new unit test engine is used to quantify the level of effort required to adapt a unit test engine. Among 22 software components in DUTF, there are 3,001 lines of code that are generalized and reused to generate the different DSL unit test engines. On average, through our use of AspectG (please see Chapter 5) in modifying grammars to hook into the DUTF, less than 360 lines of code are needed for each new unit test engine. The majority of the customized code is focused on the test script language used by each new DSL. Among the 239 lines of code in the Robot unit test engine case, there are 231 lines of code for building the Robot DSL test script interpreter; among the 482 lines of code in the FDL unit test engine case, there are 429 lines of code for building the FDL test script interpreter.

In the FDL unit test engine case, there were two extra helper functions needed that deal with storing the current objects in a managed data structure and comparing the expected results regardless of the ordering of attributes stored in the data structure object. The comparison presented in Table 4-1 indicates that the amount of code and overall effort needed to adapt the specifics of a particular DSL unit test engine is relatively small when using DUTF.

DSL Category	***DSL Name***	***Number of Specific Functions or Classes***	***Customized Lines of Code***
Imperative DSL	Robot Language	2	239
Declarative DSL	FDL	4	482

Table 4-1. Generality Analysis of DUTF

Different GPL unit test engines support different GPLs and have different interfaces with users, especially the syntax and the test result output format. The re-interpreter is the one component that plays a specific role to adjust the variability in this framework. Once the underlying GPL unit test engine changes, the wrapper interface of the re-interpreter must be modified to adapt to the differences among the specific GPL unit test engines. This specificity depends on the kind of programming language that the DSL program is translated into. In this dissertation research, all the DSL examples were translated to Java, so the choice of the underlying GPL unit test engine was JUnit.

If the DSL programs are translated into C#, managed C++, or Visual Basic .Net, an alternative for a different GPL unit test engine can be NUnit [NUnit, 2007], which is a unit testing engine that provides general unit testing functionality for all languages support by .Net. NUnit is implemented using C# and has many .NET language features (e.g., custom attributes and other reflection related capabilities).

The DUTF provides developers the freedom of choosing the underlying GPL unit test engine according to the generated GPL program. Table 4-2 is a comparison of the two types of GPL unit test engines (e.g., JUnit and NUnit). It lists four commonly used types of the basic unit test assertion actions (e.g., equality assert, condition test, utility method, identity assert) that are implemented and used in the DUTF. According to the comparison table in Table 4-2, there is much similarity among the unit testing assertion semantics between JUnit and NUnit, which are almost identical with exception of small syntactical differences. For example, the equality assertion is called **`assertEquals`** (expected, actual) in JUnit and **`Assert.AreEqual`** (expected, actual) in NUnit. The actual meaning of these two equality assertions is also the same (i.e., to assess whether

the expected value is equal to the actual value, with slight differences in the assertion naming and invocation). The assertion return type and the parameters are also the same between these two assertion functions. A similar pattern can be observed in other assertions listed in Table 4-2.

Unit Test Actions	***JUnit (Java)***	***NUnit (.Net languages)***
Equality Assertion	`assertEquals (expected, actual)`	`Assert.AreEqual (expected, actual);`
Condition Test	`assertTrue(actual)`	`Assert.IsTrue(actual)`
	`assertFalse(actual)`	`Assert.IsFalse(actual)`
	`assertNull(actual)`	`Assert.IsNull(actual)`
Utility Method	`fail()`	`Assert.Fail()`
Identity Assert	`assertSame (expected, actual)`	`Assert.AreSame (expected, actual)`

Table 4-2. JUnit and NUnit Basic Unit Test Actions Comparison

The manner in which DUTF uses the underlying GPL unit test engines is to translate the DSL unit test script into GPL unit test cases that are invoked by the GPL unit test engines. The DUTF then translates the unit test results back to the DSL unit test level. In order to make the DUTF work with these two different GPL unit test engines, the developers can change the test script interpretation (e.g., the different unit testing assertion syntax) in the DSL unit test script grammar shown in Figure 4-10. Also, the output format of JUnit and NUnit is different, which requires modification of the sanitize function to handle different raw testing results. The sanitized results are then displayed in a uniform format that end-users can understand.

JUnit (Java)	Assertequal
	: (
	  AE LPAREN EXPECT VARIABLES COMMA VARIABLES RPAREN
	  {
	  fileio.print(" assertEquals(i, robot.x);");
	  fileio.print(" assertEquals(j, robot.y);");
	  fileio.print(" }");
	  }
	)
	;

NUnit (C#)	Assertequal
	: (
	  AE LPAREN EXPECT VARIABLES COMMA VARIABLES RPAREN
	  {
	  fileio.print(" Assert.AreEqual(i, robot.x);");
	  fileio.print(" Assert.AreEqual(j, robot.y);");
	  fileio.print(" }");
	  }
	)
	;

Figure 4-10. Comparison of JUnit and NUnit Assertion Usage

For different types of IDEs (e.g., Eclipse or .Net), the specific part of DUTF has the same issue as DDF, which is the way to plug the different components into the IDE. In this dissertation research, the targeted IDE is Eclipse along with its Java plug-ins, which influenced the choice of JUnit as the underlying unit test engine. Implementing DUTF in Visual Studio .Net would require changing the plug-in implementations to C# to hook into the new APIs provided by VS .Net. The plug-in implementation is different from one IDE to another IDE and depends on what features an IDE provides and how they can be extended to assist in DSL unit testing.

4.7 Related Work of Domain-Specific Language Unit Test Engines

There is no current evidence in the research literature of previous work that has investigated unit testing concerns at the DSL level. However, there are a large number of approaches that provide general language tools (e.g., lexer, parser, editor, and compiler).

There also are many language definition framework tools available (e.g., ASF+SDF, JTS, LISA, and SmartTools). These tools are surveyed in this section with a forward-looking view on how the ideas of DDF and DUTF might be integrated with each tool.

4.7.1 ASF+SDF

As related work in program transformation, ASF+SDF is the meta-language of the ASF+SDF Meta-Environment [van den Brand *et al.*, 2002], which is an interactive language environment to define and implement DSLs, generate program analysis and transformation tools, and produce software renovation tools. ASF+SDF is a modular specification formalism based on the Algebraic Specification Formalism (ASF) and the Syntax Definition Formalism (SDF). ASF uses simplified algebraic expressions that are based on conditional rewrite rules to define semantics. SDF is a BNF-like formalism for defining the lexical, context-free, and abstract syntax of languages using scannerless generalized LR parsing. Each ASF+SDF module defines the syntax and semantics of a language or language fragment. ASF+SDF supports language specification importing, renaming, and parameterization. The ASF+SDF compiler generates C code as its target, but there is a provision that the target code can be other GPLs (e.g., Java or C++), or even native machine code. ASF+SDF claims to have produced many language tools including a debugger, but unit testing tool support has not been reported.

4.7.2 JTS

The Jakarta Tool Suite (JTS) [Batory *et al.*, 1998] is a set of tools for extending a programming language with domain-specific constructs. The focus of JTS is DSL

construction using language extensions that realize a product line of DSLs. JTS consists of Jak (a meta-programming language that extends a Java superset) and Bali (a tool to compose grammars). Internally, JTS represents programs as an AST and a surface syntax tree (SST), where the AST semantically checks an SST that is annotated with type declaration and references to the symbol table. JTS does not provide rich transformation functionalities like those provided in DMS. JTS requires a separate client to support program generation. An important comparative difference between JTS and the research described in this dissertation is that the DSL testing framework does not need a complicated mechanism (e.g., language extension) to implement a DSL and the associated tools.

4.7.3 LISA

The Language Implementation System based on Attribute Grammars (LISA) [Mernik *et al.*, 2002] tool is a grammar-based system to generate a compiler, interpreter, and other language-based tools (e.g., finite state automata, visualization editor). To specify the semantic definition of a language, LISA uses an attribute grammar, which is a generalization of context-free grammars where each symbol has an associated set of attributes that carry semantic information. With each grammar production, a set of semantic rules is associated with an attribute computation [Mernik and Žumer, 2005]. LISA is platform-independent and offers the possibility to work in a textual or visual environment. It provides an opportunity to perform incremental language development of an IDE such that users can specify, generate, compile-on-the-fly, and execute programs in a newly specified language. LISA's lexical, syntax, and semantic analyzers can be

applied to different types and can operate in a stand-alone manner. Using templates, LISA is able to describe the semantic rules that are independent of grammar production rules. LISA achieves better modularization than ANTLR through templates and inheritance formalism.

The debugger, test engine, and profiler are not in the list of LISA generated language tools. From initial experience with LISA [Henriques *et al.*, 2005], it offers all of the necessary features to interact with the framework to also generate the test engines. The reason for choosing ANTLR over LISA is primarily a choice of convenience - ANTLR has pre-existing support as an Eclipse plug-in.

4.7.4 SmartTools

SmartTools [Attali *et al.*, 2001] is a language environment generator based on Java and XML. Internally, SmartTools uses the AST definition of a language to perform transformation. It uses a well-known visitor design pattern technique to specify semantic analysis on XML Document Object Model (DOM) tree structures. The principal goal of SmartTools is to produce open and adaptable applications more quickly than existing classical development methods. The implementation is based on the concept of a software factory [Parigot, 2004; Greenfield *et al.*, 2005] and is adapted to the design and implementation of applications that rely on a specific data model. SmartTools can generate a structured editor, UML model, pretty-printer, and parser specification, but a debugger, test engine, and profiler are not generated by SmartTools.

4.7.5 Other Related Testing Tools

Language definition tools help domain experts to develop their own programming languages and also generate useful language tools for the new languages (e.g., editor, compiler, and debugger). Most testing research in this area has been focused on testing methods and the efficient way to generate the unit test cases such as parameterized unit testing [Tillmann and Schulte, 2005], testing grammar-driven functionality [Lämmel and Schulte, 2006], generating unit tests using symbolic execution [Xie *et al.*, 2005], and generating test inputs of AspectJ programs [Xie and Zhao, 2006]. However, there does not appear to be any literature or relevant discussion related to unit testing of DSL programs.

4.8 Summary

As the cost of software failures rise substantially each year and the number of end-user programmers involved in the software development process increases, there is an urgent need for a full suite of development tools appropriate for the end-user's domain. Software failures pose an increasing economic risk [Gelperin and Hetzel, 1988] as end-user programmers become more deeply involved in software development without the proper unit testing capabilities for their DSL applications. Unit test cases are constructed in such a way that expresses the software module's design intent. A test script serves as a "living document" and can be easily understood by clients and other developers [Unit Testing, 2007]. A recent software development method called test-driven development (TDD) has become popularized through adoption as a practice of Extreme Programming (XP) [XP, 2007]. TDD is a software development practice that requires software

developers to write a test case first, implementation second, and constantly giving the feedback on the correctness of the application behavior as the code evolves and is refactored [TDD, 2007].

Although unit testing has many advantages that influence software engineering practices, unit testing at the GPL level has been the topic of research for several decades. Due to the lack of unit test engine support on DSL programs, end-users have not been able to take advantage of these software development methods. The DUTF provides end users a framework that can integrate traditional unit testing methods into DSLs. The DUTF allows end users to take full advantage of unit testing to experience the latest software development and maintenance methods (e.g., TDD and XP). DUTF is a novel framework for automatically generating unit test engines for DSLs by augmentation of DSL grammars.

CHAPTER 5

ASPECTG: WEAVING ASPECTS INTO DSL GRAMMARS

To address the crosscutting concerns observed in DSL grammars during the generation of DSL debuggers and unit test engines, an aspect-oriented language, AspectG, has been developed to weave aspects into DSL grammars. This chapter concentrates on the role of AspectG in assisting in the modularization of tool generation concerns through grammar augmentation. A discussion is provided on the issues and challenges involved in the design and implementation of AspectG that are focused within the domain of language grammars. Implementation details are also described from lessons learned in developing AspectG using a program transformation engine. The chapter concludes with an overview of related work and a summary.

5.1 AspectG Design Challenges

There is an urgent need for Domain-Specific Aspect Languages (DSALs) that can address particular crosscutting concerns appearing in language grammars [Rebernak *et al.*, 2006-b]. This chapter contributes to the design, implementation, and application of a DSAL, called AspectG, which is focused within the domain of language specification rather than traditional programming languages.

When designing a new DSAL, such as AspectG, a join point model (JPM) can be adopted as an alternative to the JPM used by a General-Purpose Aspect Language (GPAL) like AspectJ. The main issues in designing a JPM for AspectG were:

- What are the join points that will be captured in AspectG?
- Are AspectG join points static or dynamic?
- What is an appropriate pointcut language syntax to describe these join points?
- What are advice in this domain?
- Does the ordering of the advice matter? If so, how is the weaving order prioritized?
- How to deal with weaving conflicts and avoid infinite weaving?

The syntax specification formalism BNF is an example of a DSL with a purely declarative character. BNF is not primarily meant to be executable but nevertheless useful for application generation. AspectG uses static join points because a language specification is static. Advice in AspectG represents additional semantic rules that have to be attached to particular productions in the grammar. Hence, join points are syntactic production rules and the designed pointcut language must match arbitrary syntax productions. AspectG weaves into an ANTLR grammar, which is a syntax-directed translator where the order of semantic rules is important. This has the consequence that new semantic rules specified in advice have to be weaved at join points that are between semantic rules of a particular syntax production. Hence, the pointcut language in AspectG consists of predicates that match the location of an appropriate point in the language specification. Moreover, in AspectG it must be specified whether new semantic rules are weaved before or after the matched location.

5.2 AspectG Overview

It has been observed in practice that crosscutting concerns emerge in grammars used in language specification [Wu *et al.*, 2005]. In particular, from our own experience, the implementation hooks for various language tools (e.g., debugger and unit testing engine) require modifications to be made to many of the productions throughout a grammar. Manually changing the grammar through invasive modifications proved to be a very time consuming and error prone task. Because of challenges in manually adapting a language specification, it is difficult to build new testing tools for each new DSL of interest and for each supported platform.

The simpler version of the Robot language described in Chapter 2 has been written in ANTLR and partially provided in Figure 5-1. This example illustrates the ANTLR specification language with semantic rules defined in Java. From this language specification, ANTLR generates Java source code representing the scanner and parser for the Robot language.

Using the DDF and DUTF, a DSL debugger and unit test engine can be generated automatically from the DSL grammar provided that an explicit mapping is specified between the DSL and the translated GPL. To specify this mapping, additional semantic actions inside each grammar production are defined. A crosscutting concern emerges from the addition of the explicit mapping in each of the grammar productions. The manual addition of the same mapping code in each grammar production results in much redundancy that can be better modularized using an aspect-oriented approach applied to grammars. In the case of generating a debugger for the Robot language, the debug mapping for the Robot DSL debugger was originally specified manually at the Robot

DSL grammar level (see Figure 5-2). For example, lines 12 to 18 represent the semantic rule of the **RIGHT** command. Line 12 keeps track of the Robot DSL line number; line 14 records the first line of the translated GPL code segment; line 16 marks the last line of the translated GPL code segment; lines 17 and 18 generate the mapping code statement used by the DDF.

```
class P extends Parser;  // The Robot parser class in ANTLR

root:(
BEGIN
       {
          fileio.print("public class Robot {");
          fileio.print("public static void main(String[] args) {");
          fileio.print("int x = 0; int y = 0;");
       }
commands END EOF!
       {
          fileio.print("System.out.println(\"x coord= \" + x +
          \" \" + \"y coordinator= \" + y); } }");
          fileio.end();
       }
  );

commands:( command commands  |  );

command :(
LEFT   {
          fileio.print("x=x-1;");
          fileio.print("time=time+1;");
       }
|RIGHT {
          fileio.print("x=x+1;");
          fileio.print("time=time+1;");
       }
|UP    {
          fileio.print("y=y+1;");
          fileio.print("time=time+1;");
       }
|DOWN  {
          fileio.print("y=y-1;");
          fileio.print("time=time+1;");
       }
  );

class L extends Lexer; // The Robot lexer class in ANTLR

BEGIN  : "begin";
END    : "end";
LEFT   : "left";
RIGHT  : "right";
UP     : "up";
DOWN   : "down";

WS : ( ' ' // whitespace
  | '\t'
  | '\r' '\n' f
  | '\n' { newline(); }
 ) {$setType(Token.SKIP);} ;
```

Figure 5-1. Robot DSL Specification in ANTLR

These semantic actions are repeated in many terminal productions. The same mapping statements for the **LEFT** command appear in lines 20, 22, and 24 to 26. 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. An aspect-oriented approach can offer much benefit in such a case.

This chapter considers the DDF as a case study to outline two different approaches for weaving a debugging concern into a DSL grammar. A similar approach also applies to DUTF. Each approach assumes that an ANTLR grammar is used to specify the syntax and semantics of a DSL. ANTLR permits semantic action code written in a GPL to be attached to each grammar production.

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

Figure 5-2. Part of the Robot DSL Specification with Additional Debug Information

5.3 Weaving at the Generated GPL Code Level

There are two approaches that were explored in this research to determine the best mechanism for augmenting the existing DSL grammars with aspects for testing tools. The first approach to modularizing a debugging concern in a DSL assumes the existence of an aspect weaver for the generated GPL. For example, AspectJ is an aspect-oriented extension to Java that assists in modular implementation of numerous crosscutting concerns [Kiczales *et al.*, 1997; AspectJ, 2007]. In Figure 5-3, ANTLR automatically generates the *Lexer* and *Parser* from the DSL grammar. Assuming the generated parser is written in Java, AspectJ can be used to define a debugging aspect that weaves the debug mapping code to generate a new lexer and parser (*Lexer'* and *Parser'*). After the debug concern is weaved into the lexer and parser, DDF uses the transformed GPL and mapping code to generate the DSL debugger.

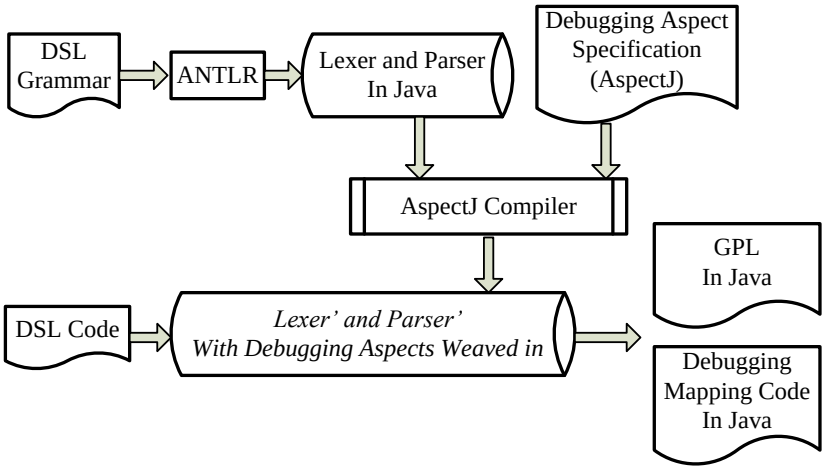


Figure 5-3. Post-ANTLR Processing (AspectJ Approach)

A fragment of an aspect for capturing the debug mapping using AspectJ is specified in Figure 5-4, with complete details in Appendix B. The **`after`** advice

specified in line 6 is associated with a **call** pointcut that captures all **command** method calls made by class **P**, which is the name of the parser class that is automatically generated by ANTLR. This aspect states that the DSL line number should be incremented (i.e., **dsllinenumber=dsllinenumber+1;**) after all calls to **P.command**, regardless of the specific **command** method (i.e., this aspect keeps track of the DSL line number across all **command** grammar productions). Within the parser generated by ANTLR, there is a long switch statement that is used to match the productions to the current token during parsing. In line 8 of Figure 5-4, the pointcut **after(int statementname)** is passed an integer parameter that matches a specific production in the grammar from this switch statement. In this case, **match(int)** represents the method call for each production that is modified to address the debugging concern within the generated parser. In line 9 of Figure 5-4, the advice **handle(statementname)** increments the DSL line number (i.e., **dsllinenumber=dsllinenumber+1;**).

```
…
6   after(): call (void P.command())
7      { dsllinenumber=dsllinenumber+1; }
8   after(int statementname): call(void antlr.Parser.match(int)) && args(statementname)
9     { handle(statementname); }
…
```

Figure 5-4. Fragment of DSL Line Mapping Aspect in AspectJ

The advice of Figure 5-4 handles the increment of the DSL line number that is weaved at the beginning of each production. Several other aspects are needed to specify the complete debug mapping. Another aspect was needed to keep track of the first and last line number of the translated segment of GPL code. After the weaving process is accomplished by AspectJ, the Parser of Figure 5-3 becomes Parser', which not only

translates the DSL to the GPL, but also generates the necessary mapping needed by DDF to generate the DSL debugger. The debugger generation process is described in Chapter 3.

The aspect of Figure 5-5 tracks the last line number of the translated segment of GPL code that is weaved after the statement (e.g., **`time=time+1;`**) within each production. Line 16 indicates the location of the weaving after the statement (i.e., the **`String`** represented by **`st`**). The **`file_io`** object is the handle to which the generated code is written. Line 20 is the line number tracking statement, which assigns the current GPL line number to the variable **`gplendline`** by calling the **`getLinenumber`** method.

In order to write the correct AspectJ code on the generated parser, a deep understanding of the generated parser code is required to identify the type and the correct function call for a matched production. The complete AspectJ source code is in Appendix B, which serves as an example of the Robot example in DDF.

```
…
16  after(FileIO file_io,String st):
17        target(file_io)&&call (void FileIO.print(String))&&args(st)
18        {
19        if((st=="time=time+1;")){
20                  gplendline=file_io.getLinenumber();
21                  }
22        }
…
```

Figure 5-5. Fragment of DSL Last Line Tracking Aspect in AspectJ

The lack of mature aspect weavers for many languages (e.g., Object Pascal, C, or Ada) is a serious disadvantage of this first approach, which requires an aspect weaver for the generated GPL as the mechanism for modularizing the debug concern. Another disadvantage of this first approach is that it requires the developer of the DSL to have detailed knowledge of the code generator within ANTLR in order to construct the

appropriate pointcuts. In some cases, the translation is done by a legacy parser, which creates a challenge because the generated parser code can be messy and generally unreadable by a human.

5.4 Weaving at the DSL Grammar Level

Although the previous section's post-ANTLR processing approach using AspectJ can solve the crosscutting problems in augmenting a DSL grammar, this method is infeasible when an aspect weaver does not exist for the generated GPL. The results of the previous section were favorable because the generated code was Java, which allowed AspectJ to be used for post-ANTLR weaving. A different technique is needed when the parser generation targets a GPL that does not have an aspect weaver. A program transformation system (e.g., DMS, as presented in Section 2.2) can be used to weave crosscutting concerns into the actual grammar definition. After weaving the aspects into the grammar using DMS, the change in terms of the aspects that were added will propagate automatically into the generated parser through the grammar productions. Unlike the first approach described in Section 5.3, it is not necessary to weave into the generated parser because the debugging concern is weaved at an earlier stage in the grammar itself.

In order to weave the aspects into DSL grammars, the first step was to construct an ANTLR parser using DMS. In Figure 5-2, the Robot DSL grammar contains an ANTLR specification of BNF syntax (e.g., lines 10, 11, and 19). The semantic action is specified using Java by separating the action code with a pair of curly braces. Note that the Java domain is embedded within ANTLR, which makes it difficult to parse two

different syntactic constructs (i.e., ANTLR and Java) using any one particular parser. A naïve solution would be to include all the tokens and productions from both domains to form a combined grammar and then generate the parser using the DMS parser generator. However, this approach does not make use of the existing DMS Java parser. A better approach reuses the existing DMS Java tools and separates the ANTLR grammar productions from the Java grammar productions, but still parses the input source containing tokens from both languages. This requires a minor extension of the DMS ANTLR grammar. To parse the embedded semantic action (i.e., essentially Java code) within the ANTLR domain, a special DMS string token called **ANTLR_ACTION** is used. The regular expression associated with this token is as follows:

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

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

The next step involves retrieving the associated string expressions from the specific **ANTLR_ACTION** nodes and parsing them with the pre-existing DMS Java parser. However, an inherent difficulty in using a regular Java parser is that the string

expressions linked to an **ANTLR_ACTION** node are not complete Java programs, only fragments (i.e., statement blocks). Therefore, to avoid exceptions thrown by the DMS Java parser, minor modifications are made to the AST root node (i.e., starting production in the Java grammar specification file) and the parser is regenerated to allow partial parsing. Because the approach specifically targets the translation from a DSL to a GPL, the semantic actions in an ANTLR grammar specification are primarily method call statements (with one string parameter, see Figure 5-2, lines 13, 15, 21, and 23).

After the parse tree for the **ANTLR_ACTION** nodes are retrieved using the modified Java parser, new debugging aspects are weaved using the **ASTInterface** API provided by DMS, which provides methods for modifying a given syntax tree to regenerate a new tree structure. The steps describing the process for building an ANTLR weaver are shown in Figure 5-6.

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

Figure 5-6. Steps to Weave Debugging Aspects into an ANTLR Grammar

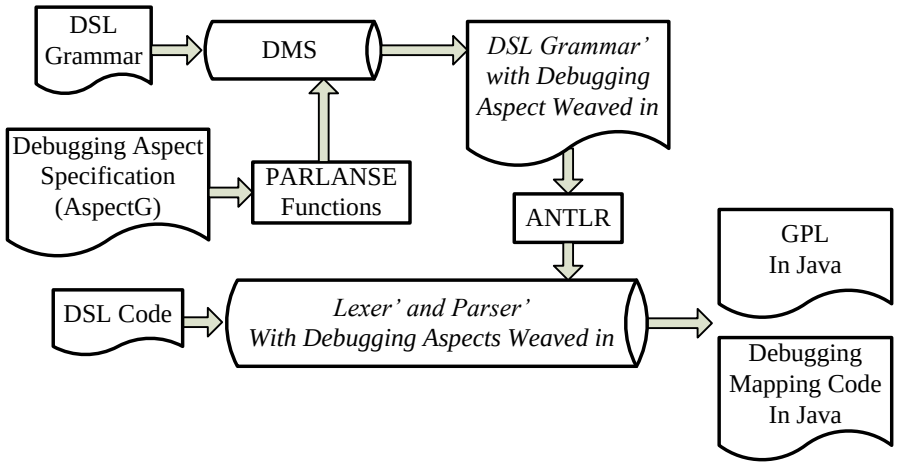


Figure 5-7. Pre-ANTLR Processing (DMS Approach)

In this approach toward modularizing concerns in a grammar, DMS is used to weave the debugging concern directly into the grammar itself, rather than the ANTLR generated GPL source code. Although the initial approach of pre-ANTLR processing using DMS can solve the crosscutting problems emerging in the DSL grammar, this method can be challenging to use because of the need to write low-level transformation rules. In DMS, low-level transformation functions are written in a language called PARLANSE (Parallel Language for Symbolic Expression) [Baxter et al., 2004]. From practical experience, transformation rules and PARLANSE functions are hard to specify and understand. Thus, program transformation tools are beyond the grasp of many developers. To resolve the problems of direct transformation of a grammar, an aspect language for grammars has been implemented, called AspectG (see example in Figure 5-8). This approach to weaving directly into grammars has the side benefit of language independence. It does not matter which GPL serves as the generated target. The DMS ANTLR domain is capable of parsing the grammar and adding the needed debug

transformations for a large set of programming languages that are pre-defined in DMS (e.g., Ada, C, C++, C#, COBOL, FORTRAN, HTML, Java, PHP, SQL, and XML).

In Figure 5-7, a debugging aspect is specified in AspectG, which generates DMS program transformation rules written in the Rule Specification Language (RSL) [Baxter *et al.*, 2004] that provides transformation functionality using pattern matching and rewrite specifications on the AST of a source program (in this case, the source is actually a grammar file). In Figure 5-7, before the grammar is even processed by ANTLR, it is first pre-processed by DMS in order to weave the debugging aspect into the original grammar productions. The transformed grammar is then submitted to ANTLR in order to generate the parser and lexer for a specific GPL. The underlying infrastructure of this grammar-focused aspect language is based on program transformation, which is a key to automating software maintenance and reengineering [Burson *et al.*, 1990]. The contribution of this second approach is the transformation of the grammar itself, rather than the generated parser code. The specification of the debug mapping is modularized in a single place – the AspectG specification.

5.4.1 AspectG Specification

An aspect language for a GPL, such as AspectJ, typically has three critical elements: a join point model, a pointcut specification language for specifying join points (which provide a type of quantification across a base artifact [Filman and Friedman, 2004]), and advice to be applied to each join point [AspectJ, 2007]. AspectG adopts the same concepts from AspectJ and targets a different software artifact (i.e., a language grammar). A contribution of this work is an investigation into aspects as they apply to

grammars (e.g., determining the meaning of a join point model within the context of a grammar). By hiding the complicated transformation rules and functions as a back-end, the DSL tool developer only needs to interact with the AspectG language rather than direct interaction with DMS. The use of AspectG provides a clean and modular implementation of crosscutting concerns in grammars (e.g., debugging and testing concerns added to a base grammar). The complete AspectG language specification written in ANTLR is provided in Appendix C. Because the crosscutting concerns are addressed at the grammar level (i.e., meta-language level), language-independent aspect-oriented programming [Lafferty and Cahill, 2003] can be realized.

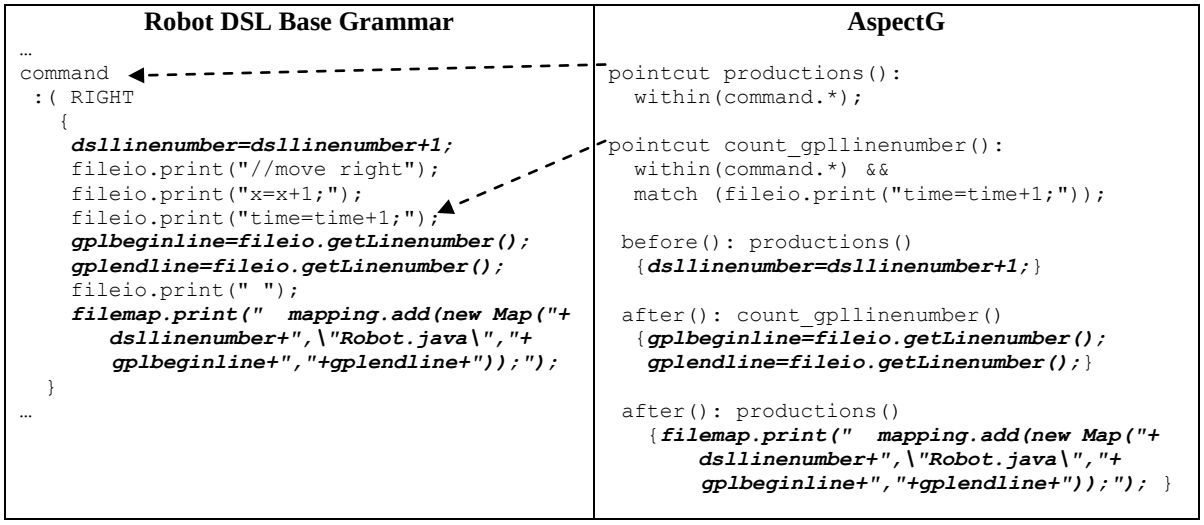


Figure 5-8. AspectG Pointcut Model

The AspectG join point model can match on both the syntax of the grammar and the syntax of the semantic action code within each production that is written in Java. Join points in ANTLR are static points in the language specifications where additional semantic rules can be attached. A set of join points in AspectG is described with pointcuts that define the location where the advice is to apply. A wildcard can be used

within the signature of a pointcut. The wildcard '*' matches zero or more terminal or non-terminal symbols to represent a set of qualified join points. Some examples of pointcut specifications are shown below:

***.*;**
matches any production in the entire Robot language

command.*;
matches any production in a ***command*** *production in the Robot language*

Pointcuts in AspectG are defined using the reserved word **pointcut** and two keywords representing pointcut predicates (i.e., **within** and **match**). The **within** predicate is used to locate grammar productions at the syntax level and **match** is used to define the location of a GPL statement within a semantic rule. Each pointcut has a unique name, a list of actual patterns (i.e., composed by terminals, non-terminals, and wild cards), and semantic rules. The patterns are used to identify the location of join points. They are passed into weaving functions to weave the semantic rules into the language grammar. Consider the following pointcut:

```
pointcut productions(): within (command.*);
```

The pointcut called **productions** is defined with the wildcard **command.*** and matches **command** productions in the Robot grammar (e.g., **RIGHT** and **LEFT**). As an example of a pointcut that combines both predicate types (i.e., **within** and **match**), consider the following:

```
pointcut count_gpllinenumber(): within (command.*)
            && match (fileio.print("time=time+1;"));
```

The pointcut **count_gpllinenumber** is a pattern specification corresponding to **command** productions having a semantic action with a statement matching the signature **fileio.print("time=time+1;")**. The advice in AspectG is defined in a similar manner to AspectJ, which brings together a pointcut that selects join points and a body of code representing the effect of the advice [AspectJ, 2007]. The advice are semantic rules written as native Java statements that can be applied at join points specified by pointcuts. In ANTLR, the order of GPL statements in semantic rules is very important. Therefore, in AspectG the ability to apply advice **before** or **after** a join point is necessary, as shown in the following example:

```
before(): productions() {dsllinenumber=dsllinenumber+1;}
after(): count_gpllinenumber()
       {gplbeginline=fileio.getLinenumber();}
```

The **before** advice defined on the **productions** pointcut means that before the parser proceeds with execution of each command production, the DSL line number is incremented (i.e., **dsllinumber=dsllinenumber+1;**). The **after** advice associated with the **count_gpllinenumber** means that line numbers for the GPL are updated (i.e., **gplbeginline=fileio.getLinenumber();**) after the parser matches a timer increment (i.e., **fileio.print ("time=time+1;");**).

After weaving a grammar aspect and parsing the Robot DSL code, the new ANTLR grammar can generate the mapping information that contains the information needed by DDF and DUTF (i.e., each Robot DSL code statement line number along with its corresponding generated Java statement line numbers are recorded in the mapping).

5.4.2 AspectG Implementation

AspectG uses the DMS program transformation system to perform the underlying weaving on the language specification. The AspectG abstraction hides the details of the accidental complexities of using the transformation system from the users; i.e., a user of AspectG focuses on describing the crosscutting grammar concerns at a higher level of abstraction using an aspect language, rather than writing lower level program transformation rules [Wu *et al.*, 2005]. In AspectG, each of the crosscutting concerns is modularized as an aspect that is weaved into an ANTLR grammar using parameterized low-level transformation function calls. Four weaving functions have been developed to handle four different types of join points that may occur within a grammar.

The four possible join points provided by AspectG are: before a semantic action (i.e., before the first statement of a semantic action code segment); after a semantic action (i.e., after the last statement of a semantic action code segment); before a specific statement that is inside a semantic action; and, after a specific statement that is inside a semantic action. These join points are represented in AspectG by **`before`** and **`after`** keywords within the context of a semantic action or specific statement. Weaving takes place after the initial phase of AspectG's compiler, which is responsible for parsing the AspectG specification and generating the program transformation rules. The generated

program transformation rules provide bindings to the appropriate weaving function parameters corresponding to the pointcut and advice defined in the aspect language.

The weaving algorithm (see Figure 5-9) describes the weaving procedure for AspectG. Note that the algorithm requires two parameters (i.e., advice and join point) and weaves the advice parameter into the join points designated by the pointcut predicate. The two nested loops ensure that any legitimate pair of pointcut and advice will make a weaving call. One of the DMS transformation tools, called **`RuleApplier`**, is the main driver application to do the program transformation on the input DSL grammar specification according to the transformation rules generated from the AspectG specification. **`RuleApplier`** is a generated DMS transformation tool that can manipulate the AST tree of the input source by the domain parser and then reconstruct the valid AST tree within the same language domain based on the customized transformation rules. The actual weaving of the language specification invoked by the weave function in the AspectG weaving algorithm (see Figure 5-9) is done by the DMS program transformation engine according to the different transformation rules generated by the AspectG compiler.

```
for all jp in pointcutslist do
  for all a in advicelist do
    if jp's name equals a's pointcut name then
      weave(jp, a)
    end if
  end for
end for
```

Figure 5-9. Generalized Algorithm for AspectG Weaving

The weave function first looks for all potential pointcut positions in the semantic sections of a grammar. The weaver then backtracks to the pointcut's ancestor node type and value at the syntax level to filter out the unqualified pointcut positions. Finally, the advice is inserted in the correct position of the grammar specification using the **`ASTInterface`** API provided by DMS, which specifies methods for modifying a given syntax tree to regenerate a new tree structure. Part of the "weaving after semantic action" function is shown in Figure 5-10. Lines 11 to 26 represent the PARLANSE function that finds all nodes that end with a left curly bracket in the input AST and stores them in a variable called **`exec_node`**. Line 11 uses an **`ASTInterface`** API function, called **`FindChildWithProperty`**, to retrieve all the child nodes from the node that is passed the third argument from the transformation rule. Line 16 is the conditional statement to filter the qualified nodes. Lines 28 to 43 represent the function that gets the nodes that are of type **`_definition_1`**, which is used to specify a particular non-terminal in the ANTLR grammar's specification. Line 28 uses an **`ASTInterface`** API function, called **`FindParentWithProperty`**, to retrieve all the parent nodes of **`exec_node`** whose node type is **`_definition_1`**. After these two search and pattern match procedures, the exact locations of the nodes defined as the pointcut are stored in **`mid_node`**, which is later used as an input parameter for AST manipulation processes that generate the new AST with woven aspects. Lines 45 to 68 represent the function to manipulate the AST tree. **`ConnectNthChild`** is an **`ASTInterface`** API to connect a sub-tree to a node's Nth child. **`DisconnectNthChild`** is an **`ASTInterface`** API to disconnect a node's Nth child. Bounded by the structure of ANTLR, a new AST tree is reconstructed by trimming off the pointcut branches and transplanting the new sub-tree

with the aspect constructed from the original input AST. The complete PARLANSE code of the **`after`** function is available in Appendix D.1 and the weaving algorithm of the **`middle`** function is available in Appendix D.2.

```
…
11 ((= exec_node (AST:FindChildWithProperty  arguments:3
12   (lambda (function boolean AST:Node
13            )function
14     (value (local (;; );;
15       (;;
16         (ifthen (== (AST:GetNodeType ?) GrammarConstants:NodeTypes:_~'~)~')
17          (return ~t)
18         )ifthen
19        (return ~f)
20       );;
21      )local
22    ~f
23    )value
24   )lambda
25  )
26 )
27
28 (= mid_node (AST:FindParentWithProperty exec_node
29   (lambda (function boolean AST:Node
30    )function
31    (value (local (;; );;
32      (;;
33        (ifthen (== (AST:GetNodeType ?) GrammarConstants:NodeTypes:_definition_1)
34         (return ~t)
35        )ifthen
36       (return ~f)
37      );;
38     )local
39     ~f
40    )value
41   )lambda
42  )
43 )
44
45 (ifthen (~= mid_node (void AST:Node))
46   (;;
47   (= mid_node (AST:GetNthChild mid_node 1))
48    (ifthen (==(@ (AST:GetString mid_node)) (@ (AST:GetString arguments:2)))
49     (;;
50      (ifthen (~= exec_node (void AST:Node))
51       (;;
52       (= parent (AST:GetParent exec_node))
53       (= search_node (AST:GetNthChild parent 2))
54       (= representation_instance (AST:GetForestRepresentationInstance
55       (AST:GetForest arguments:1 )  (AST:GetRepresentation arguments:1)))
56                 (=new_node(AST:CreateNode representation_instance
                                  GrammarCostants:NodeTypes:_semantic_strings_2))
57       (= semi_node (AST:GetNthChild search_node 3))
58       (AST:ConnectNthChild new_node 2 arguments:1)
59       (AST:ConnectNthChild new_node 3 semi_node)
60       (AST:DisconnectNthChild parent 2 search_node)
61       (AST:ConnectNthChild new_node 1  search_node)
62       (AST:ConnectNthChild parent 2 new_node)
63       );;
64      )ifthen
65     );;
66    )ifthen
67   );;
68 )ifthen
69  (return arguments:3)
70 );;
…
```

Figure 5-10. Part of **`after`** function in PARLANSE

5.5 Illustrative Examples

This sub-section illustrates the process of using AspectG to weave a DDF debugging aspect into the Robot language grammar. The mapping between the DSL (in this case, ANTLR) and the generated GPL (in this case, Java) is represented by a line number counter that keeps track of which DSL line number corresponds to the current line of GPL code being debugged. Figure 5-11 shows an aspect that counts the DSL line numbers. The DSL line number counter update statement (i.e., **`dsllinenumber=dsllinenumber+1;`**) must be inserted after each Robot language statement. The pointcut called **`productions`** (shown earlier) matches the production rules of the Robot language grammar within any instance production whose name begins with **`command`**. The aspect specifies code (i.e., **`dsllinenumber = dsllinenumber+1;`**) to run at a join point matched by the pointcut **`productions`** to update the DSL line number counter every time there is a DSL statement defined in the **`command`** production rule set.

```
aspect dsllinenumber (


   pointcut productions(): within (command.*);

   after(): productions()
      {dsllinenumber=dsllinenumber+1;}

}
```

Figure 5-11. DSL Line Number Counter Aspect in AspectG

From the high-level aspects specified in AspectG, a series of low-level transformation rules are generated that are executed on the DMS transformation engine in order to weave changes into an ANTLR grammar. Specifically, rules are generated in the

Rule Specification Language (RSL) of DMS. A series of template RSL rules have been designed as part of this research that correspond to the four types of weaving that may occur within AspectG (i.e., weaving **`before`** or **`after`** a production, and weaving **`before`** or **`after`** a semantic action). Figure 5-12 shows the low-level RSL specification generated from the **`dsllinenumber`** aspect shown in Figure 5-11, which is used to weave a debugging DSL line number into the Robot grammar specified in ANTLR. The weaving process will insert the statements that map the DSL line number statement into the appropriate places of the Robot grammar. The first line of the rule establishes the default base language domain to which the transformations are applied (in this case, ANTLR). In this example, the pattern **`after_advice`** (line 3) is an external library function that was written to perform the actual process of sub-typing, naming, and weaving. The rule **`print_after_tree`** on line 8 triggers the transformation on the Robot grammar by invoking the specified external pattern. Notice that there is a condition associated with this rule (line 10), which describes a constraint stating that the rule should be applied only to those join points where a transformation has not occurred already. This is because the DMS re-write engine will continue to apply all sets of rules until no rules can be fired. It is possible to have an infinite set of rewrites if the transformation results are not conditionally halted (i.e., when one stage of transformation continuously introduces new trees that can also be the source of further pattern matches). After applying this rule to the Robot language grammar, a new semantic segment will be generated with the line number update statement inserted at the end of every production in the **`command`** set (i.e., **`RIGHT`**, **`LEFT`**, **`DOWN`**, and **`UP`**). The essence of the

transformation can be seen in line 9 of the transformation, which states that a **java_seq** is rewritten ("->") with a parameterized call to the **after_advice** external function.

After the DSL line number counter aspect is weaved into the Robot language grammar, during the parsing phase of this modified grammar the inserted line number counter of the advice executes throughout the **command** production propagation. This line number information helps to track each Robot statement and is also passed to the source code mapping component for Robot language debugger generation.

```
1  default base domain Antlr.
2
3  external pattern after_advice(af_adv:statement_string,
4                   lefthandside: IDENTIFIER,
5                   orig_stmt:semantic):
6                   semantic = 'after_advice' in domain Antlr.
7
8  rule print_after_tree(java_seq: semantic): semantic -> semantic
9    =" \java_seq " -> "\after_advice\(\aft_advice\(\) \, \lhs\(\)\, \java_seq\)"
10 if java_seq~="\:semantic \after_advice\(\aft_advice\(\)\,\lhs\(\)\,\java_seq\)".
11  pattern aft_advice(): statement_string = "dsllinenumber=dsllinenumber+1".
12
13  pattern lhs(): IDENTIFIER ="command".
14
15  public ruleset a = { print_after_tree }.
```

Figure 5-12. Low-level Rule Transformation Generated from AspectG

Figure 5-13 is a screenshot of the generated RSL transformation rule applied on the original Robot language grammar. This screenshot shows the newly constructed AST with the DSL line number counter inserted as the last statement in each production rule's semantic section within the **command** production rule set (please note that the **DOWN** rule has been removed for space consideration, but is transformed similarly).

```
C:\DMS\Domains\Antlr\Tools\RuleApplier>run domainruleapplier
RTS Version 19.1.7
Copyright 1998 Semantic Designs, Inc. All Rights Reserved
Link_Encoding_UNICODE: 0
Code size: 13060276
Enter source: C:\Documents and Settings\Me\Desktop\p_3.g
Enter transform: rules/Robot/a
Registry: Loading RSL definitions from "c:/DMS/Domains/Antlr/rules/Robot.rsl" ...
Registry: Successfully loaded RSL definitions from "c:/DMS/Domains/Antlr/rules/Robot.rsl".
 f__after command f__after command f__after command mid_node3  mid_node4  f__after command
Print syntax tree? (yes/no):
Enter target:
class P extends Parser ;
  {
    String funcall = "Right" ;
  }
  command
  :
  (
  Right
  {
    String funcall = "Right" ;
    String funcname = "main" ;
    fileio . print ( "          //move right" ) ;
    fileio . print ( "          move_right();" ) ;
    fileio . print ( "time=time+1;" ) ;
    fileio . print ( " " ) ;
    dsllinenumber = dsllinenumber + 1 ;
  } | Left
      {
        String funcall = "Left" ;
        String funcname = "main" ;
        fileio . print ( "              //move left" ) ;
        fileio . print ( "              move_left();" ) ;
        fileio . print ( "time=time+1;" ) ;
        fileio . print ( "" ) ;
        dsllinenumber = dsllinenumber + 1 ;
      } | Up
          {
            String funcall = "Left" ;
            String funcname = "main" ;
            fileio . print ( "          //move left" ) ;
            fileio . print ( "          move_left();" ) ;
            fileio . print ( "" ) ;
            dsllinenumber = dsllinenumber + 1 ;
```

Figure 5-13. Applied Weaving of "After" Transformation Rule on the Robot Grammar

AspectG is also able to weave debugging aspects into various locations in the middle of any Robot grammar segment. As an example, the mapping between the DSL and the generated GPL needs another line number counter that keeps track of which GPL line number corresponds to the current line of DSL code being debugged. Figure 5-14 shows the GPL line number counter aspect in AspectG. The pointcut called

count_gpllinenumber matches the production rules of the Robot language grammar within any instance production whose name begins with **command**. It specifies code (i.e., **gplbeginline=fileio.getLinenumber();**) to run at a join point matched by the pointcut **count_gpllinenumber**. The intent of this aspect is to update the GPL line number counter every time there is a DSL statement defined with a semantic statement (i.e., **fileio.print("time=time+1;")**) in the **command** production rule set.

```
aspect gplbeginlinenumber (

pointcut count_gpllinenumber():
  within (command.*) && match (fileio.print("time=time+1;"));

  after(): count_gpllinenumber()
    {gplbeginline=fileio.getLinenumber();}

)
```

Figure 5-14. GPL Line Number Counter Aspect in AspectG

Figure 5-15 shows the low-level RSL specification generated by AspectG based on the **gplbeginlinenumber** aspect specification, which defines weaving a GPL line number counter into the Robot grammar. The weaving process will insert the statements that get the GPL line number from the Robot grammar propagation process. In this example, an external library pattern, a pre-defined PARLANSE transformation function called **mid_advice** (line 5), performs the actual weaving. The transformation rule **print_mid_tree** on line 10 performs the transformation process on the Robot grammar by invoking this external pattern **mid_advice**. After applying this rule to the Robot language grammar, a new semantic segment will be generated with the GPL line

number counter statement inserted at the end of every production in the **command** set (i.e., **RIGHT**, **LEFT**, **DOWN**, and **UP**) and after the statement (**fileio.print("time=time+1;")**).

After the GPL line number counter aspect is weaved into the Robot language grammar, during the parsing phase of this modified grammar the inserted line number counter of the advice executes throughout the **command** production propagation. This line number information helps track the beginning GPL line number corresponding to each Robot statement. The line number is also passed to the source code mapping component for Robot language debugger generation.

```
1 default base domain Antlr.
2
3 pattern semi(): QUOTED_STRING ="\"time=time+1;\"".
4
5 external pattern mid_advice(bef_adv:semantic_strings,
6                  semico: QUOTED_STRING,
7                  orig_stmt:semantic):
8                  semantic = 'mid_advice' in domain Antlr.
9
10 rule print_mid_tree(java_seq: semantic): semantic -> semantic
11     = " \java_seq " -> "\mid_advice\(\mi_advice\(\) \, \semi\(\)\, \java_seq\)"
12 if java_seq~="\:semantic\mid_advice\(\mi_advice\(\) \, \semi\(\)\, \java_seq\)".
13
14 pattern m_advice():semantic_strings="command;gplbeginline=fileio.getLinenumber();".
15
16 public ruleset a = { print_mid_tree }.
```

Figure 5-15. Low-level Rule Transformation Generated from AspectG

Figure 5-16 is a screenshot of the weaving process that shows the DMS **RuleApplier** applying the program transformation rule from Figure 5-15 to transform the original Robot language grammar into the new Robot language grammar. In this example, the GPL line number counter statement is inserted after the statement

(**`fileio.print("time=time+1;");`**) in the middle of semantic sections within the **`command`** production rule set.

```
C:\DMS\Domains\Antlr\Tools\RuleApplier>run domainruleapplier
RTS Version 19.1.7
Copyright 1998 Semantic Designs, Inc. All Rights Reserved
Link_Encoding_UNICODE: 0
Code size: 13060276
Enter source: C:\Documents and Settings\Me\Desktop\p_3.g
Enter transform: rules/Robot/a
Registry: Loading RSL definitions from "c:/DMS/Domains/Antlr/rules/Robot.rsl" ...
Registry: Successfully loaded RSL definitions from "c:/DMS/Domains/Antlr/rules/Robot.rsl".
 time  time  time 1 time 1Enter transform:
Print syntax tree? (yes/no):
Enter target:
class P extends Parser ;
  {
    String funcall = "Right" ;
  }
  command
  :
  (
  Right
  {
    String funcall = "Right" ;
    String funcname = "main" ;
    fileio . print ( "          //move right" ) ;
    fileio . print ( "          move_right();" ) ;
    fileio . print ( "time=time+1;" ) ;
    gplbeginline = fileio . getLinenumber ( ) ;
    fileio . print ( " " ) ;
  } | Left
      {
        String funcall = "Left" ;
        String funcname = "main" ;
        fileio . print ( "              //move left" ) ;
        fileio . print ( "              move_left();" ) ;
        fileio . print ( "time=time+1;" ) ;
        gplbeginline = fileio . getLinenumber ( ) ;
        fileio . print ( "" ) ;
      } | Up
          {
            String funcall = "Left" ;
            String funcname = "main" ;
            fileio . print ( "          //move left" ) ;
            fileio . print ( "          move_left();" ) ;
            fileio . print ( "" ) ;
          } | Down
  )
```

Figure 5-16. Applied Weaving of “Middle” Transformation Rule on the Robot Grammar

5.6 Related Work in the Area of Aspect-Oriented Grammars

This section provides an overview of related work in the area of aspect-oriented languages for weaving crosscutting concerns into grammars.

5.6.1 AspectLISA

AspectLISA [Rebernak *et al.*, 2006-a] uses an explicit join point model, such that join points are static points in a language specification where additional semantic rules can be attached. These join points can be syntactic production rules or generalized LISA rules. One pointcut can match productions in different languages over the entire hierarchy of languages. For each pointcut, several advice can be defined, which represent parameterized semantic rules written as native Java assignment statements. In AspectLISA, there is only one way to apply advice on a specific pointcut, because attribute grammars are declarative and the order of equations in semantic rules is not important (i.e., applying advice before or after a join point is not applicable). Distinguishing features of AspectLISA are inheritance of advice and pointcuts. Moreover, advice in AspectLISA is reusable because it can be parameterized on grammar symbols.

5.6.2 AspectASF

AspectASF [Klint *et al.*, 2004] is an aspect language that can weave crosscutting properties into an ASF+SDF specification. Because of its pointcut pattern language, which is based on matching patterns over the structure of equations, there are two types of available pointcuts: entering an equation (i.e., after a successful match of the left side), and exiting an equation (i.e., just before returning the right side). Two types of advice are also available: after entering (i.e., prepending the advice conditions to the list of conditions of the equation that is matched by the pointcut), and before exiting (i.e., appending the advice conditions to the list of conditions of the subject equation) [Klint *et al.*, 2004]. To apply AspectASF toward the creation of a debugger would require that the

debugger developer carefully identify all of the potential actions (e.g., step into, step out, and step over) across all possible program locations, which would make the final grammar complicated, cumbersome, and difficult to reason about. In comparison, the DDF dynamically adapts the debugging actions from the debugging events as an end-user debugs a DSL program through the Eclipse debug perspective. Because the DDF is attached to the host IDE, the DSL grammar only needs to have source code mapping and test result mapping information weaved into the base language definition.

5.7 Summary

A DSL offers end-users a notation for specifying the intent of a software system using idioms appropriate to the domain of interest. This chapter presented an approach that generates the tools needed (e.g., editor, compiler, and debugger) to use a DSL from a language specification captured in a grammar by weaving the tool generation concerns into the grammar. The difficulty of manual implementation of a DSL debugger as part of an IDE led to the idea of generating the debugger from a language specification. Yet, the decomposition of a language specification along the dimension of grammar productions forces some concerns to be scattered and tangled within the grammar. Table 5-1 indicates that on average 67 lines of additional statements need to be weaved into the grammars for each of the five different case studies. The specific contribution of the research described in this chapter is the ability to modularize the debugging concern within the DSL grammar using AOP principles. The chapter presented two approaches for weaving the debugger concern in conjunction with the DDF plug-in, with arguments as to why the grammar-level weaving is preferred.

DSL Category	*DSL Name*	*Additional Statements in Modified Grammars*
Imperative DSL	Robot Language	97
Declarative DSL	FDL	22
	BNF	31
Hybrid DSL	Hybrid Robot Language	117
	SWUL	66

Table 5-1. Usage Analysis for AspectG

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

AspectG represents a focused approach toward providing a language that allows a tool builder (i.e., a programmer who is developing a testing tool for a DSL) to define a

specific type of concern in a programming language specification. AspectG can be contrasted with GPALs (e.g., AspectC++ and AspectJ) that provide a more general language for capturing a broader range of crosscutting concerns. Within the research on DSALs, much of the application is centered on specific concerns for a language like Java or C++. This chapter differs from the scope of general AOP research by describing the investigation into DSALs for DSLs such as language specification.

The chapter also summarized the challenges of DSAL development and presented separate case studies of AspectG applied to different scenarios. An objective of the research described in this chapter concerns the topic of weaving aspects into grammars during the tool generation process. The work makes a contribution in the area of Grammarware [Klint *et al.*, 2005] by impacting the status of grammars, grammar transformations, and their relationship to tool plug-ins. A key enabler of this research is the application of aspect-oriented concepts to support a new generative approach for language tool construction.

CHAPTER 6

FUTURE WORK

This chapter describes several ideas for extending the contributions that were presented in the previous three chapters. Specifically, the future extensions include the following: 1) an extension of the current framework that enables DSL profiling; 2) investigation into the scalable, reliable, and extensible DSL testing framework that enables DSL end-users to develop their DSL application with full software engineering testing support; 3) addressing additional opportunities to generalize the framework in the areas of tool-independence and language-independence; and 4) exploiting further areas of aspect-oriented features within the language grammar. These topics are outlined in the following sub-sections.

6.1 DSL Profiler Platform

The framework described in Chapters 3 and 4 is used to automatically generate debuggers and unit test engines for DSLs. End-users may also be interested in the performance of their DSL applications during the execution of their program (e.g., CPU performance profiling, memory profiling, thread profiling). A DSL profiler would be helpful to determine performance bottlenecks and hotspots during execution.

As noted in Figure 1-2, the future research will investigate a DSL profiler framework (DPF) that generates DSL profilers by modifying the DSL grammar. The same approach and architecture depicted in Figure 3-1 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 the NetBeans Profiler as the underlying GPL profile server. As the active successor of JFluid [Dmitriev, 2004], the NetBeans Profiler, "provides a full-featured profiling functionality for the NetBeans IDE. The profiling functions include CPU, memory and threads profiling as well as basic JVM monitoring, allowing developers to be more productive in solving memory or performance-related issues" [NetBeans Profiler, 2007].

Figure 6-1 provides a tentative architecture for the tool integration of a profiler. The ANTLR translator will generate GPL code from DSL source code, which includes the source code mapping information. The profiling methods mapping component receives the end-user's profiling commands from the profiling perspective at the DSL level to determine what type of profiling commands need to be issued to a command-line profiler at the GPL level. The profiling actions will be mapped by the profiling methods mapping component as one of the input parameters of the re-interpreter. The results from the two mapping components will be re-interpreted into the GPL profile server as a series of GPL profiling commands against the generated GPL code.

The GPL profile engine will execute the profiling commands generated from the re-interpreter. Because the profiling results from the GPL profile engine will be tied to a particular GPL, the result message at the GPL level is sent back to the DSL profile result view by the profiling results mapping component, which is a wrapper interface to remap

the profiling results back into the DSL perspective. The domain experts will only see the DSL profiling result view and interact at the DSL level.

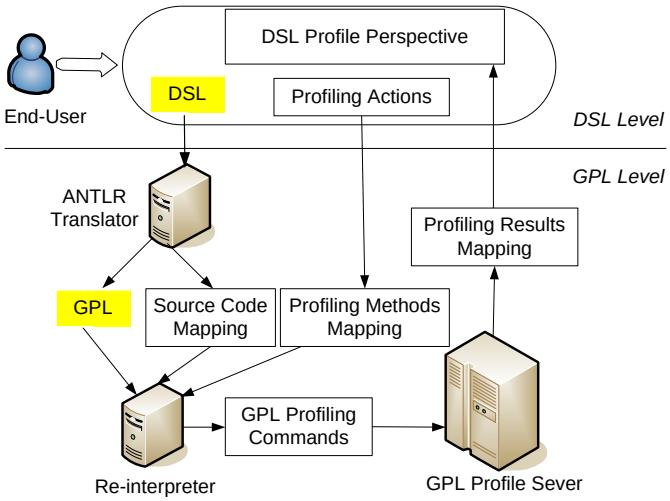


Figure 6-1. DSL Profiler Framework (DPF)

6.2 Application of Different IDE Platforms and GPLs

The generalization of this research summarized at the end of Chapters 3 and 4 can be further applied by factoring out the specific parts of the IDE to which the DSL tools are hosted. For example, a future work will explore the requirements for providing the same set of DSL tools in Microsoft Visual Studio [Parsons and Randolph, 2006], rather than Eclipse. The exploration of the Microsoft .Net plug-in architecture and its own debugging framework will help to adapt the current framework's interfaces. The Microsoft Shared Source Common Language Infrastructure (SSCLI) [Stutz *et al.*, 2003] provides many low-level code manipulation functions and programming language technologies that target the internal structure of the Microsoft .Net Framework, which can also provide an alternative solution.

The framework presented in this dissertation considers only the situation where the generated GPL is Java. However, the methodology described in this dissertation is language-independent. Another future work will be an investigation into a framework that supports different GPLs other than Java. Chapters 3 and 4 summarized how the different GPL debuggers (e.g., JDB, GDB, and Cordbg) and unit test engines (e.g., JUnit and NUnit) can be used to extend the framework to handle the situation that DSLs are generated into different GPLs.

Adaptation of the DDF and DUTF to different IDEs other than Eclipse and different generated GPLs other than Java will abstract both the tool-dependent and language-dependent nature of the framework. This new exploration would broaden the applicability of the current approach and framework. Factoring out the commonality among these extensions may identify additional limitations to help improve and refine the algorithms and architecture design.

6.3 Adaptation of DDF and DUTF to Address more Complex DSLs

Although the BNF debugger offered an initial investigation into a more complex source-to-source mapping, other mapping cases still need to been taken into consideration, such as the non-consecutive mapping case. The focus of this dissertation concerned the situation when one line of DSL code is generated into a set of consecutive lines of code in a GPL. Chapter 3 also presented an initial investigation on a DSL (e.g., BNF) that is translated into a non-trivial mapping between the DSL and the GPL. Such a non-trivial mapping occurs when the non-consecutive code translation and complex data structures are involved. 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 [Cleenewerck and D' Hondt, 2005]. An extension to the approach can also address the generation of non-consecutive GPL code by keeping track of all GPL non-consecutive line numbers at various locations and mapping the last line of the generated non-consecutive GPL code as the last line of the generated GPL code in a consecutive situation. The soundness of this potential solution needs further investigation.

As presented in Chapter 3, an initial investigation was conducted on the feasibility of the generation of a BNF debugger, where the language syntax is implemented as a table-driven parser. The framework managed to generate the basic BNF debugger by modifying the debugging algorithms. However, the ongoing research is still working towards discovering more generic ways to generate such debuggers. There are still several unanswered issues involving the BNF debugger (e.g., what does it mean to step over or step into one symbol in the left-hand side of a production?). Such questions will be addressed in the future research.

Other additional debugging features (e.g., set a watch point, roll back the program pointer, and trace a variable value history) can give the framework more options to help end-users to debug DSLs in more complicated situations.

6.4 Debugging Behavior through Event Grammars

Event grammars are defined to describe the structure of an event trace of the precise program behavior [Auguston, 1998]. With an event grammar, the semantics of a

programming language can be monitored during run-time or through post-mortem analysis. Event grammars represent a formal approach to the development of a wide range of software testing tools (e.g., debuggers, test engines, and profilers) based on a precise model of static and dynamic program behavior. A set of events (i.e., event trace) represents the program behavior and the computation over these event trace provides basic information for testing programs (e.g., assertion checking, debugging queries, execution profiles, and performance measurements) in a non-destructive way. This can be very useful in building query-based declarative debuggers and profilers for a new DSL. A traditional inspective debugger can step through the program statements and display the current variable values and unit test engines that can compare the expected values and program variable values. The event grammar approach will enrich the support of testing features of the current research and eliminate the limitation of traditional inspective debuggers (i.e., lack of software error analysis). Another intriguing application of the event grammar approach using this framework is attacking the challenges of debugging parallel programs. DDF uses a socket connection to communicate between JDB and the Eclipse debugging perspective. Different threads can be used to keep track of different branches of computation events.

Future work may also consider opportunities to apply the grammar weaving technique on event grammars.

6.5 Extending the Role of Aspects in Grammars

During the grammar weaving process, there are still several unanswered questions, such as the order of advice weaving. Different language grammar types (e.g., attribute

grammar) have different features. It is unclear at this point how to specify the AOP notation on various grammar types. Using Model Engineering [Bézivin *et al.*, 2007], the three core components (e.g., joinpoint, pointcut, and advice) of AOP can be abstracted to describe AOP at a higher abstraction level so that it is easy to extend the AOP concept to different target languages.

AspectG is able to weave the advice into only the semantic specification section of the language grammar. It would be interesting to explore the option of weaving into the syntax specification segment of the grammar. Furthermore, the weaving process presented in this dissertation is a static weaving based on the program source-to-source transformation. The task of static vs. dynamic weaving on a program grammar is another question for future research. Also, control flow is not yet provided in AspectG. Future work may include a new pointcut predicate that assists in specifying the control flow within a grammar. Such a predicate would allow aspects associated with various forms of run-time analysis to be specified and captured.

In the aspect weaving process of the current framework, as shown by Figure 5-7, the target language is a text-based language. An investigation into applying aspect weaving to numerous visual programming languages is also desirable. Using the experience gained from AspectG, a future work will investigate the graphical expression of aspects within the weaving process.

CHAPTER 7

CONCLUSIONS

As the number of software applications developed by end-user programmers rises significantly, the lack of testing tools appropriate for end-users increases the cost of software maintenance and heightens the chances of software system failures caused by software errors. Recent software development trends suggest that DSLs are becoming more popular in general usage because of their ability to assist end-users in developing applications expressed in problem domains that leverage their domain expertise. Without proper end-user DSL testing tools to assist in the detection of such errors, it is difficult, costly, and sometimes impossible to test and maintain a DSL program. Usually, while debugging and testing DSL applications, end-user programmers are forced to contend with the generated code from the existing GPL testing tools.

To solve these problems, the research in this dissertation presented a grammar-driven framework with a grammar-weaving technique that generates testing tools for DSLs. This enables DSL end-users to debug and unit test their application at the DSL level instead of being strained to test their applications at the unfamiliar GPL level. To illustrate such an approach, this dissertation presented two types of testing tools (e.g., debugger and unit test engine) that included seven different use cases of testing tools for three different DSL categories (e.g., imperative, declarative, and hybrid). The case

studies in the research include simple DSLs (e.g., Robot language), complicated DSLs used in other research efforts (e.g., FDL and SWUL), and a widely used DSL (e.g., BNF syntax specification language). The DSL testing framework and approaches described in this dissertation provide DSL programmers with the proper testing tools to debug and unit test their DSL programs at the correct abstraction level. The complete details about the research described in this dissertation, including video demonstrations and complete examples, can be found at the project website [DSL Testing Studio, 2007].

During the experimental validation phase of this research, the generic and specific parts of the testing tool generation process were observed. For different types of DSLs, the specific parts of the DDF that may need to be customized for each DSL are the debugging action algorithms and the debugging result mapping. The debugging result mapping is represented as additional semantics in the DSL grammar. For different types of DSLs, the specific parts of the DUTF are the test case mapping, scripting languages used by the end-user, and additional testing assertion functions that the GPL unit test engine does not provide. For different types of targeted GPLs, the specific part of DDF and DUTF is the underlying GPL debugger and unit test engine. The re-interpreter is the one component that plays a specific role to adjust the variability in this framework. Different GPL debuggers and test engines have different user interfaces. When the underlying GPL testing tools change, the wrapper interface of the re-interpreter must be modified to adapt the output format differences among the specific GPL testing tools. The architecture of the framework, the grammar-driven approach, and the mapping knowledge base are generic parts of the automated tool generation procedure that can be reused across different DSL testing tools.

A by-product of this research evolved from the initial manual approach of manipulating the DSL grammar in order to insert the debugging mapping information to construct the DSL debugger. To address the challenges of manual adaptation of grammars, a contribution was made in the area of grammar-driven software by analysis and transformation of the grammars applied to DSL tool generation. Specifically, the dissertation provides a description of the benefits of aspects that are applied to the construction of language testing tools. Such grammar adaptation enables end-user programmers the ability to debug and unit test their DSL application using the abstractions contained in the DSL. This dissertation also demonstrated the potential for reusing existing GPL language tools through grammar-driven automation. Initial evidence suggests that automated software engineering applied to the adaptation of existing IDE interfaces will become a future trend of tool construction.

LIST OF REFERENCES

[Aho *et al.*, 2007] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, 2nd ed, Pearson/Addison-Wesley, 2007.

[ANTLR-Studio, 2007] Antlr Studio for Eclipse, 2007, http://www.antlrstudio.com.

[ANTLR, 2007] ANother Tool for Language Recognition, 2007, http://www.antlr.org/.

[AspectC, 2007] The Home of AspectC++, 2007, http://www.aspectc.org/.

[AspectC#, 2007] AspectSharp, 2007, http://wiki.castleproject.org/index.php/AspectSharp.

[AspectJ, 2007] The AspectJ Project, 2007, http://eclipse.org/aspectj/.

[Aßmann, 2003] U. Aßmann, *Invasive Software Composition*, Springer Verlag, 2003.

[Atkins *et al.*, 1999] 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.

[Attali *et al.*, 2001] I. Attali, C. Courbis, P. Degenne, A. Fau, J. Fillon, D. Parigot, C. Pasquier, and C. S. Coen, "SmartTools: a Development Environment Generator based on XML Technologies," *Workshop on XML Technologies and Software Engineering*, Toronto, Canada, May 2001.

[Auguston, 1995] M. Auguston, "Program Behavior Model Based on Event Grammar and its Application for Debugging Automation," *Workshop on Automated and Algorithmic Debugging*, Saint-Malo, France, pp. 277-291, May 1995.

[Auguston, 1998] M. Auguston, "Building Program Behavior Models," *Workshop on Spatial and Temporal Reasoning*, Brighton, England, pp. 19-26, August 1998.

[Batory *et al.*, 1998] D. Batory, B. Lofaso, and Y. Smaragdakis, "JTS: Tools for Implementing Domain-Specific Languages," *International Conference on Software Reuse*, Victoria, Canada, pp. 143-153, June 1998.

[Batory *et al.*, 2004] 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.

[Baxter *et al.*, 2004] I. Baxter, C. Pidgeon, and M. Mehlich, "DMS: Program Transformation for Practical Scalable Software Evolution," *International Conference on Software Engineering*, Edinburgh, Scotland, pp. 625-634, May 2004.

[Bentley, 1986] J. Bentley, "Little Languages," *Communications of the ACM*, vol. 29, no. 8, pp. 711-721, August 1986.

[Bézivin *et al.*, 2007] J. Bézivin, M. Barbero, and F. Jouault, "On the Applicability Scope of Model Driven Engineering," *International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, Braga, Portugal, 2007.

[Bravenboer and Visser, 2004] 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.

[Burnett *et al.*, 2003] 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.

[Burnett *et al.*, 2005] M. Burnett, C. Cook, and G. Rothermel, "End-User Software Engineering," *Communications of the ACM*, vol. 48, no. 9, pp. 53-58, September 2005.

[Burson *et al.*, 1990] S. Burson, G. B. Kotik, and L. Z. Markosian, "A Program Transformation Approach to Automating Software Re-engineering," *International Computer Software and Applications Conference*, Chicago, IL, pp. 314-322, October 1990.

[Cleenewerck and D' Hondt, 2005] T. Cleenewerck and T. D' Hondt, "Disentangling the Implementation of Local-to-Global Transformations in a Rewrite Rule Transformation System," *Symposium for Applied Computing - Programming Language Track*, Santa Fe, NM, pp. 1398-1403, March 2005.

[Clements and Northrop, 2002] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002.

[Cordbg, 2007] Runtime Debugger (Cordbg.exe), 2007, http://msdn2.microsoft.com/en-us/library/a6zb7c8d(vs.80).aspx.

[Cordy, 2006] J. Cordy, "The TXL Source Transformation Language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190-210, August 2006.

[Cornelissen, 2004] B. Cornelissen, *Using TIDE to Debug ASF+SDF on Multiple Levels*, Universiteit van Amsterdam, December, 2004.

[Crissey, 2004] M. Crissey, "Researchers Aim to Make Debugging Simpler," *USA Today from Associated Press*, 2004.

[CSounds, 2007] A sound and music synthesis system, 2007, http://csounds.com.

[CUP, 2007] CUP User Manual, 2007, http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html.

[Czarnecki and Eisenecker, 2000] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Techniques, and Applications*, Addison-Wesley, 2000.

[Dijkstra, 1976] E. W. Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976.

[Dmitriev, 2004] M. Dmitriev, "Design of JFluid: A Profiling Technology and Tool Based on Dynamic Bytecode Instrumentation," 2004, http://research.sun.com.

[Bison, 2007] The YACC-compatible Parser Generator, 2007 http://dinosaur.compilertools.net/#bison.

[DSL Testing Studio, 2007] DTS, 2007, http://www.cis.uab.edu/wuh/ddf.

[Duclos *et al.*, 2002] F. Duclos, J. Estublier, and P. Morat, "Describing and Using Non-functional Aspects in Component Based Applications," *International Conference on Aspect-Oriented Software Development*, Enschede, Netherlands, pp. 65-75, April 2002.

[Eclipse, 2007] Eclipse, 2007, http://www.eclipse.org.

[EUSES, 2007] End-Users Shaping Effective Software Consortium, 2007, http://eusesconsortium.org/.

[Faith, 1998] R. Faith, *Debugging Programs after Structure-Changing Transformation*, Ph.D. Dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, 1998.

[Faith *et al.*, 1997] 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.

[Filman and Friedman, 2004] R. Filman and D. Friedman, *Aspect-Oriented Software Development*, Addison-Wesley, 2004.

[Floyd, 1979] R. Floyd, "The Paradigms of Programming," *Communications of the ACM*, vol. 22, no. 8, pp. 455-460, August 1979.

[Gamma and Beck, 2003] E. Gamma and K. Beck, *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*, Addison-Wesley, 2003.

[Gamma *et al.*, 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[GDB, 2007] The GNU Project Debugger, 2007, http://sourceware.org/gdb/.

[Gelperin and Hetzel, 1988] D. Gelperin, and B. Hetzel, "The Growth of Software Testing," *Communications of the ACM*, vol. 31, no. 6, pp. 687-695, June 1988.

[Gray *et al.*, 2001] J. Gray, T. Bapty, S. Neema, and J. Tuck, "Handling Crosscutting Constraints in Domain-Specific Modeling," *Communications of the ACM (Special Issue on Aspect-Oriented Programming)*, vol. 44, no. 10, pp. 87-93, October 2001.

[Greenfield *et al.*, 2005] J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, John Wiley and Sons, 2005.

[Groff and Weinberg, 2002] J. Groff and P. Weinberg, *SQL: The Complete Reference*, McGraw-Hill, 2002.

[Harrison, 2005] W. Harrison, "The Dangers of End-User Programming," *IEEE Software*, vol. 21, no. 4, pp. 5-7, July/August 2005.

[Henriques *et al.*, 2005] 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, 2005.

[Hilzenrath, 2003] D. Hilzenrath, "Finding Errors a Plus, Fannie says: Mortgage Giant Tries to Soften Effect of $1 Billion in Mistakes," *The Washington Post*, October 31, 2003.

[JDB, 2007] The Java Debugger, 2007, http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/jdb.html.

[Johnson, 1975] S. C. Johnson, "Yacc -- Yet Another Compiler Compiler," Technical report, AT&T Bell Laboratories, Murray Hill, N. J, 1975.

[JPDA, 2007], The Java Platform Debugger Architecture, 2007, http://java.sun.com/products/jpda/index.jsp.

[JSR 45, 2007] Debugging Support for Other Languages, 2007, http://www.jcp.org/en/jsr/detail?id=045.

[JUnit, 2007] JUnit, 2007, http://www.junit.org.

[Kiczales *et al.*, 1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *European Conference on Object-Oriented Programming*, Jyväskylä, Finland, pp. 220-242, June 1997.

[Kiczales *et al.*, 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," *European Conference on Object-Oriented Programming*, Budapest, Hungary, pp. 327-353, June 2001.

[Kieburtz *et al.*, 1996] B. R. Kieburtz, 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.

[Klint *et al.*, 2004] P. Klint, van. Der Storm, and J. J. Vinju, "Term Rewriting Meets Aspect-Oriented Programming," CWI Technical Report, SEN-E0421, December, 2004, http://db.cwi.nl/rapporten/index.php?persnr=331.

[Klint *et al.*, 2005] P. Klint, R. Lämmel, 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.

[Lafferty and Cahill, 2003] D. Lafferty and V. Cahill, "Language-independent Aspect-Oriented Programming," *Object-Oriented Programming, Systems, Languages, and Applications Companion*, Anaheim, CA, pp. 1-12, October 2003.

[Lämmel and Schulte, 2006] R. Lämmel and W. Schulte, "Controllable Combinatorial Coverage in Grammar-Based Testing," *IFIP International Conference on Testing Communicating Systems*, pp. 19-38, 2006.

[Loy *et al.*, 2002] M. Loy, R. Eckstein, D. Wood, J. Elliott, and B. Cole, *Java Swing*, O'Reilly, 2002.

[Mernik *et al.*, 2002] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer, "LISA: An Interactive Environment for Programming Language Development," *International Conference on Compiler Construction*, Grenoble, France, pp. 1-4, April 2002.

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

[Mernik and Žumer, 2005] M. Mernik and V. Žumer, "Incremental Programming Language Development," *Computer Languages, Systems and Structures*, vol. 1, no. 31, pp. 1-16, 2005.

[NetBeans Profiler, 2007] The NetBeans Profiler Project, 2007, http://profiler.netbeans.org/index.html.

[NUnit, 2007] NUnit Project Page, 2007, http://www.nunit.org/.

[Olan, 2003] M. Olan, "Unit Testing: Test Early, Test Often," *Journal of Computing Sciences in Colleges*, vol. 19, no. 2, pp. 319-328, December 2003.

[Olivier, 2000] P. Olivier, *A Framework for Debugging Heterogeneous Applications*, Ph.D. Dissertation, Universiteit van Amsterdam, 2000.

[Oram and Talbott, 1991] A. Oram and S. Talbott, *Managing Projects with make*, O' Reilly, 1991.

[Parigot, 2004] D. Parigot, "Towards Domain-Driven Development: The SmartTools Software Factory," *Object-Oriented Programming, Systems, Languages, and Applications Companion*, Vancouver, Canada, pp. 37-38, October 2004.

[Parnas and Siewiorek, 1975] 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.

[Parr, 2007] T. Parr, *The Definitive ANTLR Reference Building Domain-Specific Languages*, Pragmatic Bookshelf, 2007.

[Parsons and Randolph, 2006] A. Parsons and N. Randolph, *Professional Visual Studio 2005*, Wiley Publishing, Inc, 2006.

[Rebernak *et al.*, 2006-a] D. Rebernak, M. Mernik, P. R. Henriques, and M. Pereira, "AspectLISA: An Aspect-Oriented Compiler Construction System Based on Attribute Grammars," *Workshop on Language Descriptions, Tools and Applications*, Vienna, Austria, pp. 44-61, April 2006.

[Rebernak *et al.*, 2006-b] D. Rebernak, M. Mernik, H. Wu, and J. Gray, "Domain-Specific Aspect Languages for Modularizing Crosscutting Concerns in Grammars," *Workshop on Domain-Specific Aspect Languages*, Portland, OR, October 2006.

[Robbins, 2003] J. Robbins, *Debugging Applications for Microsoft.Net and Microsoft Windows*, Microsoft Press, 2003.

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

[Ruthruff *et al.*, 2006] J. R. Ruthruff, M. Burnett, and G. Rothermel, "Interactive Fault Localization in a Spreadsheet Environment," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 213-239, April 2006.

[Ryu and Ramsey, 2005] 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.

[Scaffidi *et al.*, 2005] 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.

[Schmitt, 2005] R. B. Schmitt, "New FBI Software May Be Unusable," *Los Angeles Times*, January 13, 2005.

[Sebesta, 2007] R. W. Sebesta, *Concepts of Programming Languages*, 8th ed, Addison-Wesley, 2007.

[Simonyi *et al.*, 2006] C. Simonyi, M. Christerson, and S. Clifford, "Intentional Software," *Object-Oriented Programming, Systems, Languages, and Applications (Onward Track)*, Portland, OR, October 2006.

[Spinellis, 2001] D. Spinellis, "Notable Design Patterns for Domain-Specific Languages," *Journal of Systems and Software*, vol. 56, no. 1, pp. 91-99, February 2001.

[Stutz *et al.*, 2003] D. Stutz, T. Neward, and G. Sbilling, *Shared Source CLI Essentials*, O' Reilly and Associates, 2003.

[Sutcliffe and Mehandjiev, 2004] 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.

[Tassey, 2002] G. Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing," NIST Planning Report 02-3, May, 2002, http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[Thibault *et al.*, 1999] 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.

[Tillmann and Schulte, 2005] N. Tillmann and W. Schulte, "Parameterized Unit Tests with Unit Meister," *The European Software Engineering Conference/The SIGSOFT Symposium on the Foundations of Software Engineering*, Lisbon, Portugal, pp. 241-244, 2005.

[TDD, 2007] Test-Driven Development, 2007, http://en.wikipedia.org/wiki/Test-driven_development.

[Unit Testing, 2007] Unit Test, 2007, http://en.wikipedia.org/wiki/Unit_test.

[van den Brand *et al.*, 2002] 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.

[van den Brand *et al.*, 2005] M. van den Brand, B. Cornelissen, P. Olivier, and J. Vinju, "TIDE: A Generic Debugging Framework," *Workshop on Language Descriptions, Tools, and Applications Tool Demonstration*, Edinburgh, Scotland, June 2005.

[van Deursen and Klint, 1998] A. van Deursen and P. Klint, "Little Languages: Little Maintenance?," *Journal of Software Maintenance*, vol. 10, no. 2, pp. 75-92, March 1998.

[van Deursen and Klint, 2002] 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.

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

[Van Wyk and Johnson, 2007] E. Van Wyk and E. Johnson, "Extensible Compilers and Modular Language Extensions for Computational Geometry," *Hawaii International Conference on Systems Science*, Big Island, HI, January 2007.

[VASG, 2007] VHDL Analysis and Standardization Group, 2007, http://www.eda.org/vhdl-200x/.

[Visser, 2001] E. Visser, "Stratego: A Language for Program Transformation Based on Rewriting Strategies. System Description of Stratego 0.5," *International Conference on Rewriting Techniques and Applications*, Utrecht, The Netherlands, pp. 357-361, May 2001.

[Wile, 2004] D. S. Wile, "Lessons Learned from Real DSL Experiments," *Science of Computer Programming*, vol. 51, no. 3, pp. 265-290, June 2004.

[Wile and Ramming, 1999] D. S. Wile and J. C. Ramming, "Guest Editorial: Introduction to the Special Section 'Domain-Specific Languages (DSLs)'," *IEEE Transactions on Software Engineering*, vol. 25, no. 3, pp. 289-290, May/June 1999.

[Wright and Freeman-Benson, 2004] How to Write an Eclipse Debugger, 2007, http://www.eclipse.org/articles/Article-Debugger/how-to.html.

[Wu *et al.*, 2004] H. Wu, J. Gray, and M. Mernik, "Debugging Domain-Specific Languages in Eclipse," *Eclipse Technology Exchange Poster Session*, Vancouver, Canada, October 2004.

[Wu *et al.*, 2005] H. Wu, J. Gray, S. Roychoudhury, and M. Mernik, "Weaving a Debugging Aspect into Domain-Specific Language Grammars," *ACM Symposium on Applied Computing – Programming for Separation of Concerns Track*, Santa Fe, NM, pp. 1307-1374, March 2005.

[Wu, 2006] H. Wu, "Grammar-Driven Generation of Domain-Specific Language Testing Tools," *Object-Oriented Programming, Systems, Languages & Applications Doctoral Symposium*, Portland, OR, October 2006.

[Wu *et al*., 2007] H. Wu, J. Gray, and M. Mernik, "Grammar-Driven Generation of Domain-Specific Language Debuggers," accepted for publication, *Software Practice and Experience*, 2007.

[Xie *et al.*, 2005] T. Xie, D. Marinov, W. Schulte, and D. Noktin, "Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution," *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Edinburgh, Scotland, pp. 365-381, April 2005.

[Xie and Zhao, 2006] T. Xie and J. Zhao, "A Framework and Tool Supports for Generating Test Inputs of AspectJ Programs," *International Conference on Aspect-Oriented Software Development*, Bonn, Germany, pp. 190-201, March 2006.

[XP, 2007] Extreme Programming, 2007, http://en.wikipedia.org/wiki/Extreme_Programming.

[Zellweger, 1984] P. T. Zellweger, *Interactive Source-Level Debugging of Optimized Programs*, Ph.D. Dissertation, Department of Computer Science, University of California at Berkeley, May 1984.

[Zhu *et al.*, 1997] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366-427, December 1997.

APPENDIX A

DOMAIN-SPECIFIC LANGUAGE GRAMMAR SPECIFICATIONS

The individual sections within this Appendix represent the sample grammars that were created for each case study discussed throughout the dissertation.

A.1 Robot DSL Grammar Specification

The following represents the complete Robot DSL grammar as specified in ANTLR.

```
class P extends Parser;

root
	:
		(
		  BEGIN functions ENDM commands END EOF!
		)
	;

function_names
	:
		(
		  function_name function_names
		 |
		)
	;

functionbodys
	:
		(
		  functionbody functionbodys
		 |
		)
	;

functionbody
	:
		(
		  VARIABLES LPAREN OP NUMBER COMMA OP NUMBER RPAREN
		)
	;

functions
	:
		(
		  function functions
		|
		)
	;
```

```
function
	:
		(
		  FUNCTION_HEADS functionbodys FUNCTION_HEADS ENDF
		)
	;

op
	:
		(
		  PLUS|MINUS
		)
	;

commands
	:
		(
		 command commands
		|
		)
	;

command
	:
		(
		  CALL function_name
		  | INIT VARIABLES LPAREN NUMBER COMMA NUMBER RPAREN
		  | SET VARIABLES LPAREN NUMBER COMMA NUMBER RPAREN
		  | PRINT VARIABLES
		)
	;

function_name
	:
		(
		 RIGHT
		 |LEFT
		 |UP
		 |DOWN
		 |KNIGHT
		)
	;
```

A.2 FDL Grammar Specification

The following represents the complete FDL grammar as specified in ANTLR.

```
class P extends Parser;

root
	:	(
		  feature_definitions feature_constraints
		  EOF!
		)
	;
```

```
feature_definitions
	:	(
		  feature_definition feature_definitions
		  |
		)
	;

feature_definition
	:	(
		  FEATURENAME ASSIGN feature_expression
		)
	;

feature_expression
	:	(
		  ATOMICFEATURE
		  |FEATURENAME
		  |"opt" LPAREN feature_expression RPAREN
		  |"all" LPAREN features RPAREN
		  |"moreof" LPAREN features RPAREN
		  |"oneof" LPAREN features RPAREN
		)
	;

features
	:
		(
		  feature_expression temp
		)
	;

temp
	:
		(
		  COMMA features
		  |
		)
	;

feature_constraints
	:
		(
		  feature_constraint feature_constraints
		  |
		)
	;

feature_constraint
	:
		(
		  ATOMICFEATURE feature_contraint_content
		  |"include" ATOMICFEATURE
		  |"exclude" ATOMICFEATURE
		)
	;

feature_contraint_content
	:
		(
		  "requires" ATOMICFEATURE
		  |"excludes" ATOMICFEATURE
		)
	;
```

A.3 SWUL Grammar Specification

The following represents the complete SWUL grammar as specified in ANTLR.

```
class P extends Parser;

root
	:
		(
		  imports PUBLIC CLASS IDENTIFIER LCURLY main RCURLY EOF!
		 )
	;

imports
	:
		(
		  theimport imports
		  |
		)
	;

theimport
	:
		(
		  IMPORT library:IDENTIFIER SEMI
		)
	;

main
	:
		(
		  PUBLIC STATIC VOID MAIN LPAREN THESTRING
		  IDENTIFIER RPAREN LCURLY swul statements RCURLY
		)
	;


statements
	:
		(
		  statement statements
		  |
		)
	;

statement
	:
		(
		  IDENTIFIER LPAREN signature RPAREN SEMI
		)
	;
```

```
signature
	:
		(
		  IDENTIFIER
		  |
		)
	;

swul
	:
		(
		  JFRAME IDENTIFIER ASSIGN IDENTIFIER LCURLY frame RCURLY SEMI
		)
	;

frame
	:
		(
		  title content
		)
	;

title
	:
		(
		  TITLE ASSIGN IDENTIFIER
		)
	;

content
	:
		(
		  IDENTIFIER ASSIGN PANEL IDENTIFIER BORDER
		   LAYOUT LCURLY center position RCURLY
		)
	;

center
	:
		(
		  IDENTIFIER ASSIGN IDENTIFIER LCURLY IDENTIFIER
		   ASSIGN IDENTIFIER IDENTIFIER RCURLY
		)
	;

position
	:
		(
		  IDENTIFIER ASSIGN PANEL IDENTIFIER GRID LAYOUT LCURLY row RCURLY
		)
	;

row
	:
		(
		  IDENTIFIER ASSIGN LCURLY buttons RCURLY
		)
	;
```

```
buttons
	:
		(
		  button buttons
		  |
		)
	;

button
	:
		(
		  BUTTON LCURLY IDENTIFIER ASSIGN IDENTIFIER RCURLY
		)
	;
```

A.4 Hybrid Robot Grammar Specification

The following represents the complete Hybrid Robot language grammar as specified in ANTLR.

```
class P extends Parser;

root
	:
		(
		  BEGIN functions ENDM commands END EOF!
		)
	;

function_names
	:
		(
		  function_name function_names
		  |
		)
	;

functionbodys
	:
		(
		  functionbody functionbodys
		  |
		)
	;

functionbody
	:
		(
		  VARIABLES LPAREN OP NUMBER COMMA OP NUMBER RPAREN
		| STRING
		)
	;
```

```
functions
	:
		(
		  function functions
		  |
		)
	;

function
	:
		(
		  FUNCTION_HEADS functionbodys FUNCTION_HEADS ENDF
		)
	;

op
	:
		(
		  PLUS
		  |MINUS
		)
	;

commands
	:
		(
		  command commands
		  |
		)
	;

command
	:
		(
		  CALL function_name
		  | INIT VARIABLES LPAREN NUMBER COMMA NUMBER RPAREN
		  | SET VARIABLES LPAREN NUMBER COMMA NUMBER RPAREN
		  | PRINT VARIABLES
		)
	;

function_name
	:
		(
		  RIGHT
		  |LEFT
		  |UP
		  |DOWN
		  |KNIGHT
		  |RANDOM
		)
	;
```

APPENDIX B

ASPECTJ CODE FOR POST-ANTLR GRAMMAR WEAVING

This Appendix provides the AspectJ code that weaves language extensions into the ANTLR-generated Java code. The example aspects modularize the extension that is needed to support debugging of the Robot language within the DDF. The approach typified by this code is discussed in Chapter 5, where aspects that support the DDF and DUTF are weaved into generated code.

```
public aspect SpecificationWeaver {

	int dsllinenumber = 0;
	int gplbeginline = 0;
	int gplendline = 0;
	int b1 = 0, b2 = 0, e1 = 0, e2 = 0;
	boolean status1 = true, status2 = true, status3 = true, status4 = true;
	FileIO filemap = new FileIO();

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

	pointcut count_dsllinenumber():
		call (void P.command());

	after(): count_dsllinenumber(){
		dsllinenumber=dsllinenumber+1;
		filemap.print("  mapping.add(new Map("+dsllinenumber+",
		 \"Robot.java\","+gplbeginline+","+gplendline+"));");
		}

	before(FileIO file_io, String st):
		target(file_io)&&call (void FileIO.print(String))&&args(st)
		{
		if ((st == "time=time+1;")) {
			gplbeginline = file_io.getLinenumber();
		}
	}

	after(FileIO file_io, String st):
		target(file_io)&&call (void FileIO.print(String))&&args(st)
		{
		if((st.startsWith(" x = "))||(st.startsWith("int x =
		))||(st=="int time = 0;")||(st=="System.out.println
		(\"time is \"+time);")||(st=="System.out.println
		(\"x coordiator= \" + x + \" \" + \"y coordinator= \" + y);")){
		gplbeginline=file_io.getLinenumber();

			if(dsllinenumber==1&&status1==true){
				b1=gplbeginline;
				status1=false;
				}
```

```
                if(dsllinenumber==9&&status2==true){
                        b2=file_io.getLinenumber();
                        status2=false;
                        }
                }
        }

    after(FileIO file_io, String st):
        target(file_io)&&call (void FileIO.print(String))&&args(st)
        {
        if ((st.startsWith("y = "))
                || (st == "}")
                || (st == "int time =      0;")
                || (st == "time=time+1;")
                || (st.startsWith("int y = "))
                || (st == "System.out.println(\"x coordiator= \" + x + \"
                \" + \"y coordinator= \" + y);")) {
                gplendline = file_io.getLinenumber();
                if (dsllinenumber == 1 && status3 == true) {
                        status3 = false;
                        e1 = gplendline;
                        filemap.print(" mapping.add(new Map(" + dsllinenumber
                                +",\"Robot.java\","+b1+","+e1+"));");
                } else if (dsllinenumber == 9 && status4 == true) {
                        status4 = false;
                        e2 = file_io.getLinenumber();
                        filemap.print(" mapping.add(new Map(" + dsllinenumber
                                +",   \"Robot.java\","+b2+","+e2+"));");
                        filemap.print("mapping.add(new Map("+(dsllinenumber
                                + 1)+ ",\"Robot.java\"," + e2 + "," +e2
                                + "));");
                        filemap.print("dsllinenumber=" + dsllinenumber);
                        filemap.end();
                }
        }
    }

    private void match(int name){
        switch(name){

          case PTokenTypes.BEGIN:
              filemap.begin("Mapping.txt");
              dsllinenumber=dsllinenumber+1;
              break;

          case antlr.Token.EOF_TYPE:
              dsllinenumber=dsllinenumber+1;
              break;

          default:
              break;
        }
    }
}
```

APPENDIX C

ASPECTG GRAMMAR SPECIFICATION

This Appendix provides the AspectG grammar specification. The following represents the complete AspectG grammar as specified in ANTLR.

```
class AspectG extends Parser;

root
	:
		(
		  ASPECT IDENTIFIER LPAREN pointcuts advices RPAREN EOF!
		)
	;

pointcuts
	:
		(
		 pointcut sp pointcuts
		 |
		)
	;

sp
	:
		(
		  SEMICOLON
		  |AND
		)
	;

pointcut
	:
		(
		  WITHIN STRING
		  |MATCH STRING
		  |POINTCUT IDENTIFIER COLON pointcut
		)
	;

advices
	:
		(
		  advice advices
		  |
		)
	;

advice
	:
		(
		  BEGIN IDENTIFIER STRING
		  |AFTER IDENTIFIER STRING
		  |END IDENTIFIER STRING
		  |BEFORE IDENTIFIER STRING
		)
	;
```

```
signature
	:
		(
		  type_pattern type_pattern
		)
	;

type_pattern
	:
		(
		  type_name_pattern type_pattern
		  |
		)
	;

type_name_pattern
	:
		(
		  STAR
		  | PERIOD
		  | IDENTIFIER
		)
	;
```

APPENDIX D

PARLANSE TRANSFORMATION FUNCTIONS FOR ASPECTG

This Appendix provides the PARLANSE functions that assist in weaving language extensions into the ANTLR grammar prior to code generation of the parser and lexer. These rules help to weave the debugging aspects into the Robot language grammar. These transformation functions are discussed in Chapter 5 of the dissertation, where aspects are weaved directly into grammars to support the DDF and DUTF.

D.1 After Weaving Function

The following represents the complete **`after`** weaving function written in PARLANSE.

```
(define after_advice
  (lambda Registry:CreatingPattern
    (value
      (local (;;
       [exec_node AST:Node]
       [search_node AST:Node]
       [new_node AST:Node]
       [empty_node AST:Node]
       [parent AST:Node]
       [mid_node AST:Node]
       [semi_node AST:Node]
       [representation_instance AST:RepresentationInstance]);;

       (;;
       (= exec_node (AST:FindChildWithProperty  arguments:3
       (lambda (function boolean AST:Node )function
         (value (local (;; );;
           (;;
            (ifthen (== (AST:GetNodeType ?)GrammarConstants:NodeTypes:_~'~}~')
              (;;
               (Console:PutString ` f__after ')
               (return ~t)
               );;
            )ifthen
            (return ~f)
      );; )local
      ~f)
         )value
       )lambda
       )
      )
```

```
  (= mid_node (AST:FindParentWithProperty  exec_node
   (lambda (function boolean AST:Node )function
    (value (local (;; );;
     (;;
      (ifthen (== (AST:GetNodeType ?) GrammarConstants:NodeTypes:_definition_1)
        (return ~t)
      )ifthen
     (return ~f)
     );;
    )local
    ~f
    )value
   )lambda
  )
 )


    (ifthen (~= mid_node (void AST:Node))
     (;;
      (= mid_node (AST:GetNthChild mid_node 1))
       (Console:PutString (@(AST:GetString arguments:2)))
        (ifthen (==(@ (AST:GetString mid_node))(@ (AST:GetString arguments:2)))
         (;;
          (Console:PutString ` mid_node3 ')
           (ifthen (~= exec_node (void AST:Node))
            (;;
             (Console:PutString ` mid_node4 ')
              (= parent (AST:GetParent exec_node))
              (= search_node (AST:GetNthChild parent 2))
              (= representation_instance (AST:GetForestRepresentationInstance
              (AST:GetForest arguments:1) (AST:GetRepresentation arguments:1)))
              (= new_node (AST:CreateNode representation_instance
               GrammarConstants:NodeTypes:_semantic_strings_2))
              (= semi_node (AST:GetNthChild search_node 3))
              (AST:ConnectNthChild new_node 2 arguments:1)
              (AST:ConnectNthChild new_node 3 semi_node)
              (AST:DisconnectNthChild parent 2 search_node)
              (AST:ConnectNthChild new_node 1  search_node)
              (AST:ConnectNthChild parent 2 new_node)
            );;
           )ifthen
         );;
        )ifthen
      );;
    )ifthen
   (return arguments:3)
  );;
 )local
  (void AST:Node)
  )value
 )lambda
)define
```

D.2 Middle Weaving Function

The following represents the complete **`middle`** weaving function written in PARLANSE. It weaves aspects into the middle of a semantic action segment of a DSL grammar.

```
(define mid_advice
 (lambda Registry:CreatingPattern
  (value
   (local
    (;;
     [exec_node AST:Node]
     [search_node AST:Node]
     [search_string (reference string)]
     [node_string (reference string)]
     [new_node AST:Node]
     [empty_node AST:Node]
     [semi_node AST:Node]
     [parent AST:Node]
     [mid_node AST:Node]
     [temp_node AST:Node]
     [advice_node AST:Node]
     [where_to_apply_transform AST:Node]
     [representation_instance AST:RepresentationInstance]
    );;

    (;;
     (= exec_node (AST:FindChildWithProperty  arguments:3
      (lambda (function boolean AST:Node )function
       (value (local (;; );;
        (;;
         (ifthen(== ~t (AST:ContainsString ?))
          (;;
           (= search_string (AST:GetString ?))
            (ifthen (== (@ search_string) (@ (AST:GetString arguments:2)))
             (;;
              (Console:PutString ` time ')
               (return ~t)
             );;
            )ifthen
          );;
         )ifthen
        (return ~f)
        );; )local
        ~f
       )value
      )lambda
     )
    )
     (ifthen (~= exec_node (void AST:Node))
      (;;
       (= mid_node (AST:FindParentWithProperty  exec_node
       (lambda (function boolean AST:Node)function
       (value (local (;; );;
```

```
   (;; (ifthen (== (AST:GetNodeType ?)
                 GrammarConstants:NodeTypes:_definition_1)
                 (return ~t)
       )ifthen
       (return ~f)
   );;
  )local
  ~f
  )value
  )lambda
  )
 )

(ifthen (~= mid_node (void AST:Node))
 (;;
  (= temp_node (AST:GetNthChild arguments:1 1))
  (= temp_node (AST:GetNthChild temp_node 2))
  (= temp_node (AST:GetNthChild temp_node 2))
  (= temp_node (AST:GetNthChild temp_node 1))
  (= temp_node (AST:GetNthChild temp_node 1))
  (= advice_node (AST:GetNthChild arguments:1 2))
  (= mid_node (AST:GetNthChild mid_node 1))
  (ifthen (==(@ (AST:GetString mid_node)) (@ (AST:GetString temp_node)))
        (;;
         (ifthen (~= exec_node (void AST:Node))
          (;;
           (Console:PutNatural (AST:NumberOfParents exec_node))
           (=where_to_apply_transform (AST:FindParentWithProperty
            exec_node
           (lambda (function boolean AST:Node )function
            (value (local (;; );;
             (;;
              (ifthen (== (AST:GetNodeType ?)
               GrammarConstants:NodeTypes:_semantic_strings_2)
               (return ~t)
              )ifthen
              (return ~f)
             );;
             )local
             ~f
            )value
           )lambda
          )
         )
  (=where_to_apply_transform (AST:GetParent where_to_apply_transform ))
  (= search_node (AST:GetNthChild where_to_apply_transform 1))
  (= representation_instance (AST:GetForestRepresentationInstance
  (AST:GetForest arguments:1 ) (AST:GetRepresentation arguments:1)))
  (= new_node (AST:CreateNode representation_instance
   GrammarConstants:NodeTypes:_semantic_strings_2))
  (= semi_node (AST:GetNthChild where_to_apply_transform 3))
  (AST:ConnectNthChild new_node 2 advice_node)
  (AST:ConnectNthChild new_node 3 semi_node)
  (AST:DisconnectNthChild where_to_apply_transform 1 search_node)
  (AST:ConnectNthChild where_to_apply_transform 1 new_node)
  (AST:ConnectNthChild new_node 1 search_node)
   );;
   )ifthen
   );;
   )ifthen
   );;
  )ifthen
 );;
```

```
     )ifthen
    (return arguments:3)
    );;
   )local
  (void AST:Node)
  )value
 )lambda
)define
```