Unit Testing for Domain-Specific Languages

Hui Wu[1], Jeff Gray[1], and Marjan Mernik[2]

[1] Department of Computer and Information Sciences,
University of Alabama at Birmingham,
Birmingham, Alabama USA
{wuh,gray}@cis.uab.edu
[2] Faculty of Electrical Engineering and Computer Science,
University of Maribor, Maribor, Slovenia
marjan.mernik@uni-mb.si

Abstract. Domain-specific languages (DSLs) offer several advantages by providing idioms that are similar to the abstractions found in a specific problem domain. However, a challenge is that tool support for DSLs is lacking when compared to the capabilities offered in general-purpose languages (GPLs), such as Java and C++. For example, support for unit testing a DSL program is absent and debuggers for DSLs are rare. This limits the ability of a developer to discover the existence of software errors and to locate them in a DSL program. Currently, software developers using a DSL are generally forced to test and debug their DSL programs using available GPL tools, rather than tools that are informed by the domain abstractions at the DSL level. This reduces the utility of DSL adoption and minimizes the benefits of working with higher abstractions, which can bring into question the suitability of using DSLs in the development process. This paper introduces our initial investigation into a unit testing framework that can be customized for specific DSLs through a reusable mapping of GPL testing tool functionality. We provide examples from two different DSL categories that serve as case studies demonstrating the possibilities of a unit testing engine for DSLs.

Keywords: Domain-specific languages, unit testing, tool generation.

1 Introduction

Tool support is an important factor toward determining the success and adoption of any software development paradigm. Software development using domain-specific languages (DSLs) is no exception. This paper discusses the current lack of basic tool support for common tasks that a DSL programmer may expect to perform. The paper motivates the need for DSL testing tools and introduces an initial prototype framework that supports unit testing of DSL programs.

The need to test DSL programs is motivated by the recent trend in end-user programming. Scaffidi et al estimate that there are several million end-user programmers [29]. End-user programmers are more likely to introduce software errors than professional programmers because they lack software development training and proper tool

W.M. Taha (Ed.): DSL 2009, LNCS 5658, pp. 125–147, 2009.

support [16]. As observed in several industry studies, individual examples of software errors have been very costly [18], [30]. For instance, it has been estimated that software failures collectively contribute to over $60 billion in unreported losses per year [32]. Likewise, without the availability of software development tools, the final products of end-user programming can also be dangerous [16]. The proper programming tools (e.g., editor, compiler, test engine, and debugger) are needed for end-users to improve 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 be provided with tools that allow them to detect and find software errors at an abstraction level that is familiar to them.

1.1 Benefits of DSL Adoption

Despite advances in programming languages and run-time platforms, most software is developed at a low-level of abstraction relative to the concepts and concerns within the problem space of an application domain. DSLs assist end-users in describing solutions in their work domains [41]. A DSL is a programming language targeted toward a particular problem domain rather than providing general solutions for many domains [23], [38]. DSLs have also been shown to assist in software maintenance whereby end-users can directly use the DSLs to make required routine modifications [4], [36]. DSLs can help end-users and professional programmers write a software solution in a more concise, descriptive, and platform-independent way. It is often the case that domain experts are not familiar with general-purpose languages (GPLs) [31] in order to solve their domain problems; however, they may be more comfortable with addressing their domain problems through a DSL that is closer to the abstractions of their own domain knowledge. A language that closely represents key domain abstractions may also permit the construction of a more concise program that is also easier to test for correctness compared to the same intention expressed in a GPL. This paper demonstrates the idea of DSL testing using two small DSLs that could be adopted by end-users with little programming experience.

In addition to the benefits offered to end-user programmers, there are also advantages of DSL usage by professionally trained developers. The benefits of using DSLs within a software development project are increased flexibility, productivity, reliability, and usability, which have been shown through empirical evaluation on numerous case studies [9], [21], [23], [40]. Popular examples of DSLs include the languages used to specify grammars in parser generators like CUP [8] or ANTLR (ANother Tool for Language Recognition) [1]. Other examples include the Swing User-interface Language (SWUL), which is a DSL to construct a Java Swing user interface [6]; Structured Query Language (SQL) is a DSL to access and manipulate databases; HTML is a DSL that serves as a markup language for creating web pages.

1.2 Challenges of DSL Tool Implementation

A common practice for DSL implementation is to translate a single DSL construct into several GPL constructs [23], and then reuse the associated GPL tools to provide the infrastructure for interpreting or executing a DSL program. In the research described in this paper, we adopt this popular practice of source-to-source transformation to translate a

DSL to an existing GPL, such as Java or C++, which have well-developed programming tools. Translating a DSL to an existing GPL is a popular implementation approach because the underlying tools of the converted GPL can be reused (e.g., compiler, profiler, testing engine and debugger). Although direct reuse of the existing GPL tools offers several benefits, a GPL tool does not provide the proper abstractions that are needed by DSL programmers. An approach that hides the underlying use of the GPL tools and removes the accidental complexities that cause the abstraction mismatch between the DSL and GPL is needed. The goal of DSL tool implementation is to allow users of a language to work only at the abstraction level of the DSL semantics, while not being concerned about the complexities of the solution as realized in the tools of a supporting GPL. The situation is similar to the desire to have tools for a GPL at the level of the programming language, rather than underlying machine code (e.g., a debugger that is at the machine code level is useless to most GPL programmers). A goal is complete tool support at the proper abstraction level, regardless of how the tool's execution semantics are eventually realized.

The construction of a DSL compiler or interpreter is only the first piece of the needed toolchain. A DSL programmer also needs tools to discover the existence of software errors and locate them in a DSL program. The paucity of such tools can be one of the major factors that may prevent wider acceptance of DSLs in the software industry. Our previous work [43] focused on the generation of debuggers for DSLs through adaptations to the DSL grammar. This paper follows a similar path that considers the generation of unit testing engines for DSLs.

1.3 Tool Support for Testing DSL Programs

The research described in this paper focuses on a framework that assists in customizing a tool that tests a DSL program's behavior. In order to assess the benefits of our approach, we define a unit test script that is customized to consider the domain-specific characteristics for different types of DSLs. To guide developers in testing their DSL programs, we adopt the traditional unit testing concepts such as defining the expected value of a test case, test assertions, and test result reports.

Building DSL testing tools from scratch for each new DSL is time consuming, error prone, and costly. In this paper, we show that such tools can be implemented by a generalization of a testing tools framework implemented within Eclipse (Figure 1) through reuse of existing GPL tool functionality. Figure 1 highlights the architecture of a DSL unit testing framework; we have previously developed a similar framework for DSL debuggers [43]. A key technique of the framework is a mapping process that records the correspondence between the DSL code and the generated GPL code. The translator that defines the mapping between the DSL and GPL levels is written in ANTLR, which is a lexer and parser generator [1]. This translator generates GPL code and source code mapping information that can be used to determine which line of the DSL code is mapped to the corresponding segment of generated GPL code.

The correspondence between DSL testing actions and GPL testing actions are given by pre-defined mapping algorithms that also can be specialized to deal with specific domain features. Through the explicit mapping of source code and test case mappings, the GPL testing tool responds to the testing commands sent from the re-interpreter component (bottom-left of Figure 1). The test result at the GPL level is

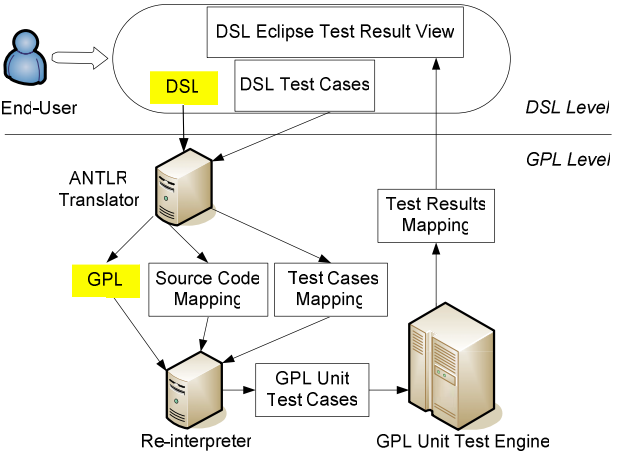

Fig. 1. Separation of DSL Perspective and GPL Tools

later sent back to the DSL testing perspective by the testing results mapping component, which is a wrapper interface to convert the GPL testing result messages back into a form to be displayed at the DSL level (i.e., rephrasing the GPL test results into the testing or debugging perspective within Eclipse that is tailored for a specific DSL). As a result, the framework enables a DSL programmer to interact directly with the testing perspectives at the DSL level. More details about our framework, including video demonstrations and complete examples, can be found at the project website [11].

The remainder of this paper is organized as follows: Section 2 introduces the necessary background information to provide the reader with a better understanding of other sections of the paper; Section 3 describes an overview of the issues concerning DSL unit testing; Section 4 presents two case studies that illustrate our ideas; Section 5 shares a few lessons learned and describes several existing limitations that point toward future work; Section 6 discusses related work; Section 7 offers a summary of the paper.

2 Background

Our approach exploits a technique to build DSL tools from existing GPL tools available for debugging, unit testing, and profiling (e.g., jdb, JUnit, NetBeans Profiler) with interoperability among the tools provided by plug-in mechanisms offered by most IDEs. The Eclipse plug-in development environment [14] was selected due to its ability to serve as a tool integration platform that offers numerous extension points for customization through an extensible architecture. As an example, Eclipse provides a reusable debugging interface (e.g., buttons for common debugging commands and variable watch lists) and integration with JUnit through extension mechanisms.

To provide tool support consistent with accepted software engineering practice, a DSL unit test engine should offer developers the ability to discover the existence of

software errors within the DSL program. After identifying the presence of an error through testing, DSL debuggers can further help end-users to locate the errors in the DSL code. This section introduces the necessary background of the basic tools and techniques mentioned throughout the paper, and provides a brief description of the JUnit test platform and DSL Debugging Framework (DDF). This section also introduces two sample DSLs that serve as case studies in later sections of the paper.

2.1 Eclipse JUnit Test Platform

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 the errors in their program.

JUnit is a popular unit testing tool for constructing automated Java test cases that are composable, easy to write, and independent [20]. A JUnit plug-in for Eclipse [12] provides a framework for automating functional unit testing [1] on Java programs. 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) as: the total number of passed or failed test cases; the true expected value and current 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

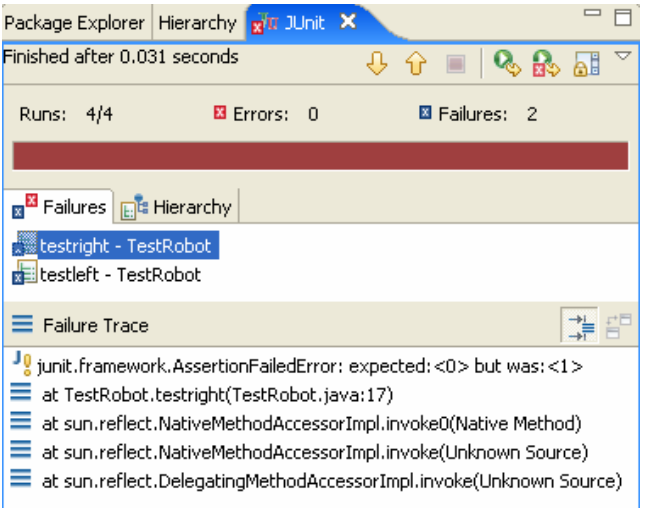

Fig. 2. Screenshot of the JUnit Eclipse Plug-in

code locations of the tested program. The test cases are displayed in a hierarchical tree structure that defines the relationship among test cases. In its current form, JUnit is focused solely on Java and is not applicable to general testing of DSL programs.

Testing frameworks similar to JUnit, such as NUnit [26], also focus at the GPL level and do not provide opportunities for unit testing DSL programs. In Section 3, we describe how our mapping framework enables unit testing of DSL programs using JUnit as the underlying unit test engine.

2.2 DSL Debugging Framework (DDF)

DDF represents our prior work [43] that was developed as a set of Eclipse plug-ins providing core support for DSL debugging. In the DDF, a DSL is specified using ANTLR, which can be used to construct recognizers, compilers, and translators from grammatical descriptions containing Java, C++, or C# actions. A DSL is usually translated into a GPL that can be compiled and executed [23]. From a DSL grammar, the DDF generates GPL code representing the intention of the DSL program (i.e., the DSL is translated to a GPL and the GPL tools are used to generate an executable program) and the mapping information that integrates with the host GPL debugger (e.g., the stand-alone command line Java debugger – jdb [19]). The generated mapping code and pre-defined debugging method mapping knowledge re-interpret the DSL program and DSL debugging states into a sequence of commands that query the GPL debugger. The debugging result responses from the GPL debugger are mapped back into the DSL debugger perspective. Thus, the end-user performs debugging actions at the level of abstraction specified by the DSL, not at the lower level abstraction provided by the GPL. The DDF was developed using architecture similar to Figure 1, but with mappings to different GPL tools (in the DDF case, a debugger rather than a unit test engine).

2.3 Sample DSLs: The Robot and Feature Description Languages

DSLs can be categorized as imperative or declarative [23]. An imperative DSL follows a similar definition used for imperative programming languages, which assumes a control flow that is centered on the importance of state changes of variables. A declarative DSL is a language that declares the relationships among input and output values, with little concern over specific control flow. This paper uses an example from both the imperative and declarative categories to illustrate the concept of unit testing of DSLs. A third category, called embedded DSLs, is discussed in Section 5 as a topic of future work for DSL unit testing.

The top of Figure 5 illustrates a very simple imperative DSL that we have used in various aspects of our work on DSL testing engines. This simple language moves a toy robot in various directions (e.g., up, down, left, right) and provides a mechanism for users to write their own navigation methods (e.g., a **`knight`** method that moves according to the rules of a knight in chess). An implicit **`position`** variable keeps track of the position of the robot as the control flow of the program modifies the robot location.

The Feature Description Language (FDL) [37] is a declarative DSL for describing feature models in software product lines. The upper part of Figure 3 (adapted from [37] and previously presented in [43] within the context of DSL debugging) is an example written in FDL to describe car features. The lower part of Figure 3 enumerates all of the possible legal configurations that result from the features defined on the

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)
```

Fig. 3. Car Features Specified in FDL and List of Possible Car Configurations (adapted from [37])

upper part of the figure. In feature 1 of Figure 3, a **Car** is made of four mandatory parts: **carbody**, **Transmission**, **Engine**, and **Horsepower**. As shown at the end of feature 1, a **Car** has an optional feature called **pullsTrailer**. Features starting with a lowercase letter are primitive features that are atomic and cannot be expanded further (e.g., the **carbody** feature). Features that start with an uppercase character are composite features, which may consist of other composite or primitive features (e.g., the **Transmission** feature consists of two primitive features, **automatic** and **manual**). There are several composition logic operators to help describe more complex situations. In feature 2 of Figure 3, 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. FDL also provides constraint keywords to describe a condition that a legal composition must satisfy. 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**.

3 DSL Unit Testing Framework

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 [27] to reduce the cost of software failures. To complement the DDF, the DSL Unit Testing Framework (DUTF) assists in the construction of 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 (Section 2.2) 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) to obtain unit test results that are remapped onto the abstractions representing the DSL. The key mapping activity in DUTF translates the DSL unit test script into GPL unit test cases that are executed at the GPL tool level. 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 a potential error in the DSL program.

An illustrative overview of the DUTF is shown in Figure 4. With the mapping generator embedded inside the grammar of the Robot language, the lexer and parser generated by ANTLR (step 1) takes the Robot DSL program as input. ANTLR not only translates the Robot DSL into the corresponding Robot.java representation, 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 information about which line of a Robot DSL unit test case is mapped to the corresponding JUnit test case in the generated 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 intention 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).

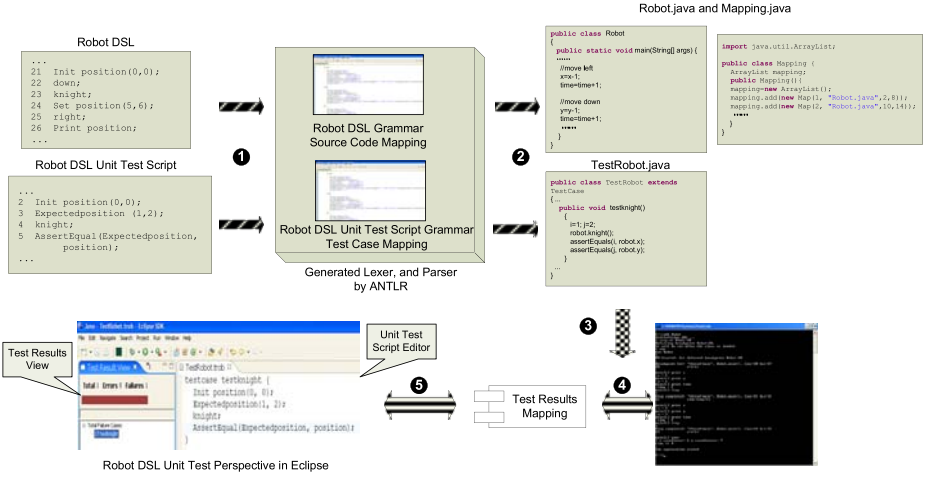

Fig. 4. DSL Unit Testing Framework (DUTF)

There are two round-trip mapping processes involved (step 4 and step 5) between the Robot DSL unit test perspective in Eclipse and JUnit. The results from the mapping components are reinterpreted into the GPL unit test engine as unit test cases that are executed against the translated GPL code. The *source code mapping component* (Section 3.1) 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* (Section 3.2) 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 (in this example, JUnit) executes the test cases generated from DSL test scripts. Because the messages from the GPL unit test engine are still in the GPL format, the test result at the GPL level is sent back to the Eclipse DSL test result view by the *test results mapping component* (Section 3.3), which is a wrapper interface to remap the test results back into the DSL perspective. The domain experts only see the DSL test result view at the DSL level. The following sub-sections describe the various mappings that are needed to reuse a GPL testing tool at the DSL abstraction level.

3.1 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 DSL to a GPL (e.g., Java) and also to generate the mapping hooks that interface with the DUTF infrastructure. The base grammar of the DSL is modified with additional semantic actions that generate the source code mapping needed to create the DSL unit test engine. The mapping consists of the line number of the DSL statement, the translated GPL file name, the line number of the first line of the mapped code segment in the GPL, the line number of the last line of the corresponding code segment in the GPL, the name of the function at the current DSL line number, and the statement type (e.g., **functiondefinition**, **functioncall**, or **none**) at the current DSL line number.

A **functiondefinition** contains **functionhead**, **functionbody**, and **functionend**, where: **functionhead** marks the start of a function (line 3 on the top of Figure 5 is the **functionhead** of **knight**); **functionbody** is the actual definition of a function (lines 4 to 6 on the top of Figure 5 represent the **functionbody** of **knight**); **functionend** indicates the end of a function (line 7 on the top of Figure 5 is the **functionend** of **knight**). A **functioncall** is the marker for a function that is being called from another program location. The statement type for a built-in method or statement of a GPL program is set to **none**. For example, the mapping information at Robot DSL line 13 in the top of Figure 5 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 defined 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 unit

Program Written in Robot DSL

```
…
3   begin knight:
4     position(+0,+1);
5     position(+0,+1);
6     position(+1,+0);
7   end 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;
…
```

Generated Java Code

```
…
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 coord="+x+""+
27                      "y coord= " + y);}
…
```

Fig. 5. Robot DSL Source Code Mapping

test engine and unit test platform can be separated from different DSLs and GPLs. Variable mapping implicitly exists within the DSL compiler specified during the syntax-directed translation in the semantics specification. Figure 6 describes a 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 the top of Figure 5 into lines 7 and 8 of the generated Robot.java in the bottom of Figure 5. 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 DUTF for generating the DSL unit test engines.

```
Functionbody
 :(VARS LPAREN op1:OP func_n1 :NUMBER COMMA op2:OP func_n2: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+"\""+"));");
    }
 );
```

Fig. 6. Part of Robot DSL Grammar Specification

The source code mapping shown in Figure 5 is the same mapping that is also used in the DDF for DSL debuggers [43]. In previous work, we showed how the adaptations to the grammar that support DSL tools represent a crosscutting concern. The grammar concerns can be specified using an aspect language that is specific to language grammars [42]; however, this topic is out of scope for the discussion in this paper.

3.2 Test Cases Mapping

The abstraction mismatch between DSLs and GPLs also contributes to the mismatch in test cases. When writing a unit test case at the DSL level, one variable in a DSL program may not be equivalent to one variable in the corresponding GPL representation (i.e., a DSL variable may be translated into several variables or 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 written in ANTLR. In the DUTF, the generated GPL test cases are exercised by the underlying GPL unit test engine (e.g., JUnit). The main task of a unit test is to assert an expected variable value against the actual variable value when the program is executed. The variable mapping from a DSL program to the corresponding GPL program is used to construct the mapping 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 variable mapping and test case line number mapping.

Figure 7 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; line 3 on the left side is mapped to lines 4 and 5 on the right side. One assertion statement in the Robot DSL unit test script may be translated into multiple 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 the Java translation; line 5 (left side of Figure 7) is mapped to lines 7 and 8 (right side of Figure 7). This one-to-many test case assertion mapping must be re-mapped back into the DSL view.The Car example, specified in the declarative FDL of Section 2.3, is used as another target DSL case study. Figure 8 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 8

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 (JUnit) Test Case

```
1  public void testknight() {
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);}
…
```

Fig. 7. Test Case Mapping Between Robot Test Case and JUnit Test Case

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 result of lines 3 and 4 is the creation of an explicit variable called **feature**, which keeps track of the list of features that are included in the actual features generated by the given FDL specification.

The parse function used in line 27 of the JUnit test case is a helper function that stores the output of the generated Java program into a unified 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 feature test assertions. 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 may not be equal. Even if the contents of these two objects are equal the result is still false when compared with **assertEquals** at the JUnit level. However, at the Car FDL abstraction level, they are equal. To address this issue, an external function called **compareFeatures** has been written to handle this situation where only the contents matter and the ordering issue can be ignored.

Line 6 of the Car FDL unit test case is mapped to line 28 of the JUnit test case. It is an assertion to assess the number of possible valid feature combinations. The external **getFeatureListNumber** function retrieves the number of the feature combinations from the parsed data structure. It is not possible to get 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 example, 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

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);
7  }
```

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)));
…
```

Fig. 8. FDL Test Cases Mapping

function is more complicated than the Robot example. JUnit does not support the sophisticated assertion functionality that is needed for FDL unit testing. Thus, helper and comparison functions were needed to realize the unit test intention for FDL programs. The provision of such helper functions for the specific use of FDL unit testing represents additions that were needed to customize the DUTF for FDL.

3.3 Unit Test Result Mapping

JUnit reports the total number of test cases, total number of failed test cases, and total number of error test cases (i.e., those representing run-time exception errors during test executions). If one test case in a DSL is translated into multiple test cases at the GPL level, the result mapping can become challenging (i.e., rephrasing the results at the JUnit level back into the DSL perspective). 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. The approach adopted in DUTF is to use 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). Within each test case, 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 a simple mapping also helps to determine the location

```
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, testFileName, linenumber);
14   action.run();
15 }
```

Fig. 9. handleDoubleClick function in TestResultView Class

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, we keep the test case name the same during the translation process. By matching the test case name through the test case mapping information, we can obtain the corresponding line number of the DSL unit test script from the JUnit test case line number in the test result report. This is illustrated in more detail in the concrete examples presented in Section 4 (e.g., Figures 12 and 13).

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

4 Example DSL Test Engines

This section illustrates the application of the DUTF on two different types of DSLs (e.g., imperative DSLs and Declarative DSLs). We have implemented unit test engines for various DSLs (e.g., the toy Robot language and the FDL, both described in Section 2.3) using our approach.

4.1 Generation of Declarative DSL Test Engine

This sub-section describes the generation of a unit test engine for the imperative Robot DSL. The DUTF adapts the JUnit graphical user interface in Eclipse by adding a new view called the DSL **`Test Result View`**, which is similar to the underlying

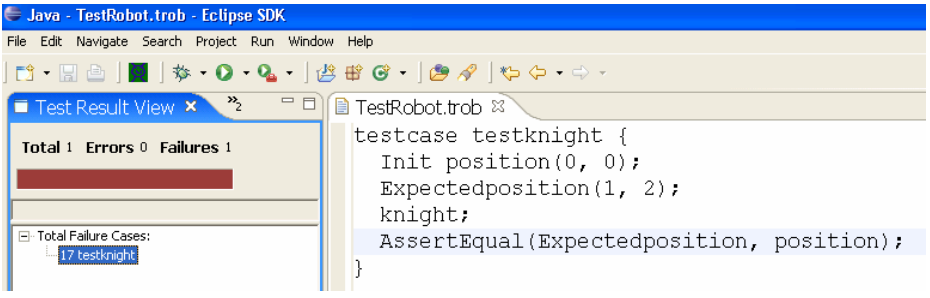

Fig. 10. Screenshot of Unit Testing Session on Robot Language

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

Fig. 11. Correct and Incorrect Knight Methods

JUnit **`Test Result View`**, but mapped to the DSL abstraction. Figure 10 shows a screenshot of a Robot DSL unit test session. 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. We have implemented the Robot DSL unit test script translator in ANTLR to generate JUnit test cases from the DSL test script. The source code mapping for the Robot DSL unit test script can also be generated by adding additional semantics to the base DSL grammar (in this case, the grammar of the Robot language). The base grammar generates the equivalent Java code for the Robot DSL (see Section 3.1) and the additional semantics generate the Java unit test cases for the Robot DSL unit test script (see Section 3.2).

The right side of Figure 10 is the DSL unit test script editor, which shows an actual Robot DSL unit test script called **`TestRobot`**. 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 determine whether there is a distinction between the expected **`position`** and the actual **`position`** after **`knight`** is executed. An incorrect implementation of the knight method is shown in the right side of Figure 11 (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 (e.g., <1, 2>) is not equal to the actual **`position`** (e.g., <2, 2>). 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 10. The **AssertEqual** assertion in this DSL unit test script tests whether its two parameters are equal. The **Test Result View** also indicates the total number of test cases (in this case 1), the total number of failures (in this case, there was 1 failure), and the number of run-time 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**).

4.2 Generation of Declarative DSL Test Engine

In addition to generating a unit test engine for an imperative DSL like the Robot language, we also used the DUTF to generate a declarative DSL unit test engine for the FDL. The declarative DSL test engine translates the unit test script into unit test cases in JUnit, which are specified in the script grammar. Because of the domain-specific syntax of the FDL, the DUTF unit test script translator required modification so that it can generate the correct unit test cases for FDL in Java. Also in the declarative DSL case, the variable mapping from the GPL to DSL is different from the imperative DSL case. In the Robot language, a commonly referenced abstraction is the **position** variable and in the FDL an important abstraction is the **feature** variable, which contains the list of possible features up to the current line of execution in the DSL.

A developer or end-user who uses the FDL often tries to narrow down a design space amid a large range of possible configurations. Manually analyzing a large design space can be challenging and even infeasible; an automated testing tool that assists in determining the correct configuration specification can be very helpful to a developer. A screenshot of the unit testing session on a Car FDL program is shown in Figure 12. The right side of Figure 12 is the FDL unit test script editor, which shows an 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 expected value has the legitimate car feature configuration according to the Car FDL program. The target unit to

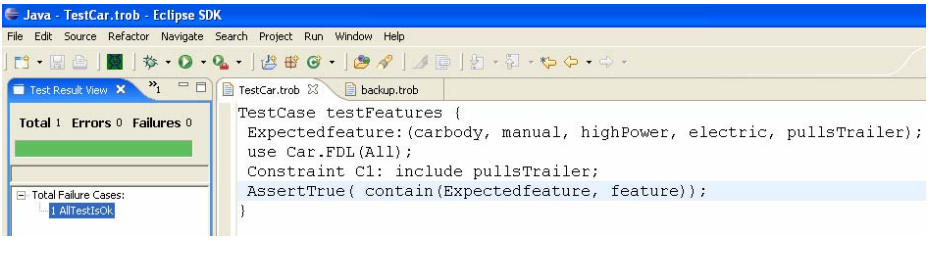

Fig. 12. Screenshot of Unit Testing Session on Car FDL

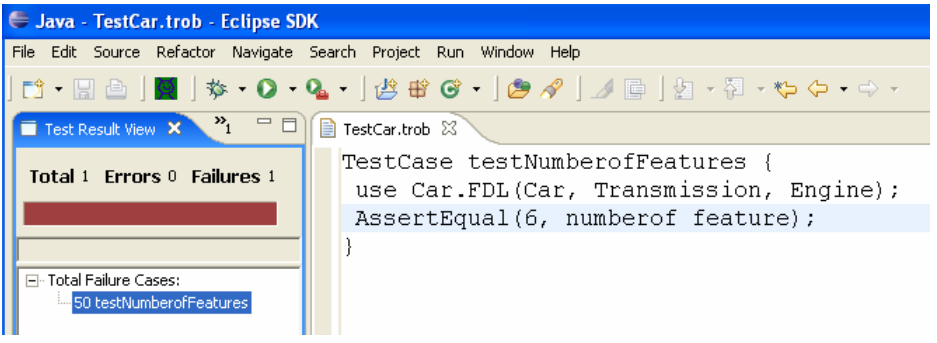

Fig. 13. Screenshot of Unit Testing Session on Car FDL

be tested is all the features (from feature 1 to feature 4 in Figure 3) plus one constraint (constraint 1 in Figure 3). We introduce another assertion called **`AssertTrue`** to assess whether the tested unit will return true or false. If it returns true, the **`AssertTrue`** assertion will pass, 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. In the left side of Figure 12, the **`Test Result View`** indicates this assertion succeeds, so **`Expectedfeature`** is one of the possible features.

Figure 13 is another test case called **`testNumberofFeatures`**, which is highlighted in the **`Test Result View`**. The expected number of possible features is 6. The targeted testing unit consists of three features (from feature 1 to feature 3 in Figure 3). The **`numberof`** operator returns the size of a set. An **`AssertEqual`** assertion tests whether the number of possible features after executing all these three features is 6. In the left side of Figure 13, the test result view indicates this assertion fails. The only features executed are **`Car`**, **`Transmission`**, and **`Engine`**. 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 actually 12 (2*3*2) rather than the expected 6, which causes the test case to fail as indicated in the DSL unit **`Test Result View`** of Figure 13.

5 Lessons Learned, Limitations, and Future Work

One of the most challenging tasks in creating the DUTF was identifying the differences among unit testing between the GPL and DSL levels for each particular mapping. The main task of a unit test is to assert an expected variable value against the actual variable value when the program is executed. The abstraction mismatch between DSLs and GPLs also contributes to the mismatch in test cases. When writing a unit test case at the DSL level, one variable in a DSL program may not be equivalent to one variable in a GPL (i.e., a DSL variable may be translated into several variables or objects in the generated GPL). Moreover, the concept of comparing raw values

may not be relevant at the DSL level. For example, in the case of the FDL, a software developer may be interested in testing the number of all possible configurations, or testing if a particular configuration is present in the specified program. In the latter case, the order of particular features in a configuration is irrelevant. Hence, a unit test engine should not report a failure if only the order of features is different (i.e., the equality assertion at the FDL level is about set equality of feature lists).

By using the DUTF, the development effort to build DSL unit testing tools was significantly reduced compared to developing a testing tool for each DSL from scratch. DUTF consists of 22 reusable software components implemented in over 3k lines of code. The amount of code that was written for each new unit test engine can be used to quantify the level of effort required to adapt a unit test engine. Based on the two example DSL unit test engines described in this paper for the Robot and FDL languages, on average, less than 360 additional lines of code were needed for each new DSL unit test engine (e.g., the Robot DSL required two additional classes representing 239 lines of code, and the FDL DSL required four additional classes representing 482 lines of code). Most of the customized code handles the different scripting languages that were needed for each type of DSL (e.g., imperative or declarative). Of course, more complex and feature-rich DSLs will likely require additional customization, but the infrastructure of DUTF provides a basis for general unit testing concepts that can be reused by different DSL tooling efforts.

There are opportunities for applying the ideas of this paper to other contexts and testing tools:

- Software developers 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. The same approach as depicted in Figure 1 can be applied to generate a DSL profiler (i.e., the framework's GPL unit test engine can be replaced by a GPL profiler, which monitors the run-time characteristics of the execution environment). The framework can use the NetBeans Profiler as the underlying GPL profile server in the case when a DSL is generated to Java. The NetBeans Profiler provides basic profiling functionalities including CPU, memory and threads profiling, as well as basic JVM monitoring. The GPL profile engine could execute the profiling commands generated from the reinterpreter inside the framework. The domain experts will only see the DSL profiling result view and interact at the DSL level. We are currently in the process of completing the DSL profiler architecture. An alternative to the NetBeans Profiler is JFluid [10], which could also be integrated into the general testing tool framework.
- Even though our framework implementation is integrated within Eclipse using Java-based tools (e.g., jdb and JUnit), we believe the concepts of the framework can be generalized and applied to other IDEs (e.g., Microsoft Visual Studio [33]) such that other testing tools (e.g., Cordbg or NUnit) can be leveraged for reuse when the underlying generated GPL (e.g., C++ or C#) changes.

There remain several open issues that need to be investigated further, such as:

- Embedded DSLs are becoming more popular and represent the case when DSL statements are embedded inline within the code of a GPL. From our previous experience with generating embedded DSL debuggers [43], we believe the same approach described in this paper can apply to unit test engines for embedded DSLs. DSL segments mapping to a GPL represent the same situation discussed in this paper. GPL segments surrounding embedded DSL statements are already in the GPL format and do not require further mapping. The interesting question for unit testing embedded DSLs concerns the interaction between the DSL and GPL boundaries (e.g., when data defined by a DSL is processed by GPL code).
- The two DSLs introduced in this chapter were rather simple to implement in terms of the source-to-source translation from the DSL program to GPL code. These two examples represented a direct translation where text in the DSL had a linear correspondence to the generated GPL. However, some source-to-source translations of a DSL may have a single DSL construct spread across multiple places in the generated GPL code. The challenge is that a more complex mapping is needed to track the line numbers between the two abstractions and how the test results map back into the DSL perspective.

6 Related Work in DSL Tool Support

The End-Users Shaping Effective Software (EUSES) Consortium [13] represents collaboration among several dozen researchers who aim to improve the software development capabilities provided to end-users. 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 [7]. More specific to the focus of our research, this section provides an overview of related work in the areas of language definition framework tools (e.g., ASF+SDF, JTS, LISA, and SmartTools). The following related work represents frameworks that can generate many useful language tools, however, none of them addresses unit testing for DSLs.

ASF+SDF is the meta-language of the ASF+SDF Meta-Environment [35], 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+SDF has produced many language tools including a debugger.

The Jakarta Tool Suite (JTS) [3] is a set of tools for extending a programming language with domain-specific constructs. JTS consists of Jak (a meta-programming language that extends a Java superset) and Bali (a tool to compose grammars). The focus of JTS is DSL construction using language extensions that realize a product line of DSLs.

The Language Implementation System based on Attribute grammars (LISA) [24], [25] 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. LISA 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 [17].

Another language extension environment for Java is the Java Language Extender (JLE) framework [39], which is also based on attribute grammars written in a specification language called Silver. The JLE permits extensions to a host language (e.g., Java) to incorporate domain-specific extensions.

SmartTools is a language environment generator based on Java and XML [2]. Internally, SmartTools uses the AST definition of a language to perform transformation. The principal goal of SmartTools is to produce open and adaptable applications more quickly than existing classical development methods. SmartTools can generate a structured editor, UML model, pretty-printer, and parser specification.

These language definition framework tools help domain experts to develop their own programming languages and also generate useful language tools for the new languages (e.g., language-sensitive editor and compiler). Other researchers are more interested in testing methods and the efficient way to generate the unit test cases such as parameterized unit testing [34], testing grammar-driven functionality [22], generating unit tests using symbolic execution [44], generating test inputs of AspectJ programs [45]. The idea of applying formal methods to determine the correctness of DSL programs was discussed in [5]. However, there does not appear to be any literature or relevant discussion related to unit testing of DSL programs.

7 Conclusion

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 [15] as end-user programmers become more deeply involved in software development without the proper unit test capabilities for their DSL applications. The utility of a new DSL is seriously diminished if supporting tools needed by a software developer are not available.

To initiate discussion of testing DSL programs, this paper introduced DUTF, which is a novel framework for generating unit test engines for DSLs. The technique is centered on a mapping process that associates DSL line numbers with their corresponding representation in the generated GPL line numbers. The Eclipse plug-in architecture and the JUnit test engine provide the infrastructure and unit testing support needed to layer the concept of DSL unit testing on top of pre-existing GPL tools. Although the contribution described in this paper is only a first effort with an associated prototype [11], we believe that the issue of testing DSL programs and the necessary tool support will become increasingly important as a push is made to adopt DSLs in general practice.

Acknowledgements

This work was supported in part by an NSF CAREER grant (CCF-0643725) and an IBM Eclipse Innovation Grant (EIG).

References

[1] ANTLR, ANother Tool for Language Recognition (2008), http://www.antlr.org/
[2] Attali, I., Courbis, C., Degenne, P., Fau, A., Fillon, J., Parigot, D., Pasquier, C., Coen, C.S.: SmartTools: a Development Environment Generator based on XML Technologies. In: ICSE Workshop on XML Technologies and Software Engineering, Toronto, Canada (2001)
[3] Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: Tools for Implementing Domain-Specific Languages. In: Fifth International Conference on Software Reuse, Victoria, Canada, pp. 143–153 (1998)
[4] Bentley, J.: Little Languages. Communications of the ACM 29(8), 711–721 (1986)
[5] Bodeveix, J.P., Filali, M., Lawall, J., Muller, G.: Formal methods meet Domain Specific Languages. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 187–206. Springer, Heidelberg (2005)
[6] Bravenboer, M., Visser, E.: Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions. In: Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, Canada, pp. 365–383 (2004)
[7] Burnett, M., Cook, C., Pendse, O., Rothermel, G., Summet, J., Wallace, C.: End-User Software Engineering with Assertions in the Spreadsheet Paradigm. In: International Conference on Software Engineering, Portland, OR, pp. 93–105 (2003)
[8] CUP User Manual (2007), http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html
[9] Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Techniques, and Applications. Addison-Wesley, Reading (2000)
[10] Dmitriev, M.: Design of JFluid: A Profiling Technology and Tool Based on Dynamic Bytecode Instrumentation. Sun Microsystems Technical Report. Mountain View, CA (2004), http://research.sun.com
[11] Domain-Specific Language Testing Studio (2009), http://www.cis.uab.edu/softcom/DDF
[12] Eclipse (2008), http://www.eclipse.org
[13] End-Users Shaping Effective Software Consortium (2007), http://eusesconsortium.org
[14] Gamma, E., Beck, K.: Contributing to Eclipse: Principles, Patterns, and Plug-Ins. Addison-Wesley, Reading (2003)
[15] Gelperin, D., Hetzel, B.: The Growth of Software Testing. Communications of the ACM 31(6), 687–695 (1988)
[16] Harrison, W.: The Dangers of End-User Programming. IEEE Software 21(4), 5–7 (2005)
[17] Henriques, P., Pereira, M., Mernik, M., Lenič, M., Gray, J., Wu, H.: Automatic Generation of Language-based Tools using LISA. IEE Proceedings – Software 152(2), 54–69 (2005)
[18] Hilzenrath, D.: Finding Errors a Plus, Fannie says: Mortgage Giant Tries to Soften Effect of $1 Billion in Mistakes, The Washington Post (2003)

[19] JDB, The Java Debugger (2008), http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jdb.html
[20] JUnit (2006), http://www.junit.org
[21] Kieburtz, B.R., Mckinney, L., Bell, J.M., Hook, J., Kotov, A., Lewis, J., Oliva, D., Sheard, T., Smith, I., Walton, L.: A Software Engineering Experiment in Software Component Generation. In: International Conference on Software Engineering, Berlin, Germany, pp. 542–552 (1996)
[22] Lämmel, R., Schulte, W.: Controllable Combinatorial Coverage in Grammar-Based Testing. In: IFIP International Conference on Testing Communicating Systems, New York, NY, pp. 19–38 (2006)
[23] Mernik, M., Heering, J., Sloane, A.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys 37(4), 316–344 (2005)
[24] Mernik, M., Lenič, M., Avdičaušević, E., Žumer, V.: LISA: An Interactive Environment for Programming Language Development. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 1–4. Springer, Heidelberg (2002)
[25] Mernik, M., Žumer, V.: Incremental Programming Language Development. Computer Languages, Systems and Structures 31, 1–16 (2005)
[26] NUnit Project Page (2008), http://www.nunit.org/
[27] Olan, M.: Unit Testing: Test Early, Test Often. Journal of Computing Sciences in Colleges 19(2), 319–328 (2003)
[28] Rebernak, D., Mernik, M., Wu, H., Gray, J.: Domain-Specific Aspect Languages for Modularizing Crosscutting Concerns in Grammars. In: IET Software (Special Issue on Domain-Specific Aspect Languages) (2009) (in press)
[29] Scaffidi, C., Shaw, M., Myers, B.: Estimating the Numbers of End Users and End User Programmers. In: Symposium on Visual Languages and Human-Centric Computing, Dallas, TX, pp. 207–214 (2005)
[30] Schmitt, R.B.: New FBI Software May Be Unusable. Los Angeles Times (2005)
[31] Sebesta, R.W.: Concepts of Programming Languages. Addison-Wesley, Reading (2003)
[32] Tassey, G.: The Economic Impacts of Inadequate Infrastructure for Software Testing. NIST Planning Report 02-3 (2002), http://www.nist.gov/director/prog-ofc/report02-3.pdf
[33] Thai, T.L., Lam, H.: Net Framework Essentials. O'Reilly, Sebastopol (2002)
[34] Tillmann, N., Schulte, W.: Parameterized Unit Tests with Unit Meister. In: European Software Engineering Conference (ESEC)/Symposium on the Foundations of Software Engineering, Lisbon, Portugal, pp. 241–244 (2005)
[35] Van Den Brand, M., Heering, J., Klint, P., Oliver, P.: Compiling Language Definitions: The ASF+SDF Compiler. ACM Transactions on Programming Languages and Systems 24(4), 334–368 (2002)
[36] Van Deursen, A., Klint, P.: Little Languages: Little Maintenance? Journal of Software Maintenance 10(2), 75–92 (1998)
[37] Van Deursen, A., Klint, P.: Domain-Specific Language Design Requires Feature Descriptions. Journal of Computing and Information Technology 10(1), 1–17 (2002)
[38] Van Deursen, A., Klint, P., Visser, J.: Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices 35(6), 26–36 (2000)
[39] Van Wyk, E., Krishnan, L., Schwerdfeger, A., Bodin, D.: Attribute Grammar-based Language Extensions for Java. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 575–599. Springer, Heidelberg (2007)
[40] Wile, D.S.: Lessons Learned from Real DSL Experiments. Science of Computer Programming 51(3), 265–290 (2004)

[41] Wile, D.S., Ramming, J.C.: Guest Editorial: Introduction to the Special Section "Domain-Specific Languages (DSLs)". IEEE Transactions on Software Engineering 25(3), 289–290 (1999)
[42] Wu, H., Gray, J., Roychoudlhury, S., Mernik, M.: Weaving a Debugging Aspect into Domain-Specific Language Grammars. In: Symposium for Applied Computing (SAC) – Programming for Separation of Concerns Track, Santa Fe, NM, pp. 1370–1374 (2005)
[43] Wu, H., Gray, J., Mernik, M.: Grammar-Driven Generation of Domain-Specific Language Debuggers. Software: Practice and Experience 38(10), 1475–1497 (2008)
[44] Xie, T., Marinov, D., Schulte, W., Noktin, D.: Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 365–381. Springer, Heidelberg (2005)
[45] Xie, T., Zhao, J.: A Framework and Tool Support for Generating Test Inputs of AspectJ Programs. In: International Conference on Aspect-Oriented Software Development, Bonn, Germany, pp. 190–201 (2006)
[46] Zhu, H., Hall, P.A.V., May, J.H.R.: Software Unit Test Coverage and Adequacy. ACM Computing Surveys 29(4), 366–427 (1997)