Annotation Processing as Local Variable Crosscutting

Ulises Juárez-Martínez
CINVESTAV - IPN
Department of Electrical Engineering
México, D. F. 07360
Email: ujuarez@computacion.cs.cinvestav.mx

José Oscar Olmedo-Aguirre
CINVESTAV - IPN
Department of Electrical Engineering
México, D. F. 07360
Email: oolmedo@es.cinvestav.mx

Abstract—Annotations describe program metainformation that can be used by pre-processing tools. However, annotation information cannot be passed to the compiler, precluding in this way the design of language extensions to the Java programming language. Besides, annotations do not provide the required level of granularity to get access to important language features like local variables. In this paper, we describe Enfasis, a tool designed to support crosscutting concerns on local variables. The crosscutting model is based on bytecode instrumentation and on a pattern matching mechanism based on instruction-paths to identify and access any local variable in the method body. The contribution of this paper consists on showing how annotations can be used to define crosscutting on local variables. This important kind of crosscutting is not supported by AOP languages including AspectJ.

Keywords: Annotation processing, local variable crosscutting, fine grained aspects.

I. INTRODUCTION

Annotations provide information that describe specific program features in a self-contained format. The annotation processor retrieves this information to generate descriptor files, class definitions or any additional information about the application. Annotations are also known as metadata that allow to introduce statements at different points of the source code. For example, consider the class Testable shown below where the field s is annotated with @Test.

```java
public class Testable {
    @Test String s = "An annotated field";
}
```

An annotation does not change the meaning of any member defined in the class. At the source code level, annotations look like an extra modifier for class members that can be recognized and processed by third party annotation processing tools. For the given example, the annotation definition is shown next.

```java
import java.lang.annotation.*;
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {}
```

An annotation resembles an interface with an at-sign @ as prefix and additional meta-annotations. @Target specifies the kind of element, in this case, on a field. Other values are PARAMETER, ANNOTATION_TYPE, CLASS, CONSTRUCTOR, METHOD, LOCAL_VARIABLE, PACKAGE and TYPE. @Retention specifies where annotations can be available: at source code (SOURCE), at class files (CLASS) or at run-time (RUNTIME). Runtime annotations are read reflectively by the virtual machine. Class annotations are available in the class but can be discarded by the virtual machine and source code annotations are discarded by the compiler.

Despite of the retention policies, each annotation needs its own processor. Runtime annotations need a processor built with a reflective API such as java.lang.reflect. Source code annotations can be built with the annotation processing tool (apt) or directly with the Java compiler. In latter case, the programmer uses the Mirror API to model the semantic structure of a program. Finally, class annotations are basically the same as runtime annotations but labelled as invisible by the virtual machine and can be read by instrumentation tools.

The following excerpt from an annotation processor source code shows how to check the @Test annotation on class Testable.

```java
import java.lang.reflect.Field;
public class CheckAnnotation {
    public static void main(String[] args) throws Exception {
        Class klass = Testable.class;
        Field f = klass.getDeclaredField("s");
        Test t = f.getAnnotation(Test.class);
        System.out.println(t);
    }
}
```

In this example, the declared and annotated field s on class Testable is retrieved via reflection applying method getAnnotation(Test.class), shown as @Test() in the source file.

A. Annotation limitations

Retention policies are not the same for different elements in a program, resulting in inconsistencies that can be categorized as follows:

- **Local variables.** Annotations for local variables are only available at source code level and cannot be used...
during program execution. Without runtime support, local variable annotations become only markers. Local variables play an important role in runtime enforcement of assertion verification: detecting references to null pointer values [6], checking method pre- and post-conditions, monitoring class invariants and estimating run-time program complexity. Although the Java community has targeted annotations for local variables using package com.sun.mirror.type, its design only considers the local variable type as a derived type of formal parameters in methods and constructors [5].

- **Reflection.** Due to the fact that Java supports reflection only on class members, annotations bear the same limitations. For instance, local variable annotations are not supported neither at runtime nor at class level.

- **Complexity for building annotation processors.** When annotations have runtime targets it is not difficult to build its corresponding processor as shown in the last section. However, if the programmer needs to process many annotations, a number of problems arise when different annotations use the same name because they become indistinguishable for their processors. For source code annotations, the apt tool is useful but it does not represent an important reduction in the complexity of dealing with multiple annotations.

- **Annotation mismatch to Java code.** Annotations do not provide the means to get access to the current values of local variables during program execution as required in runtime assertion enforcement. Another crosscutting concern is to decide on program termination. Program termination requires to find a monotonically decreasing sequence of the non-negative values that takes an integer local variable. The implementation requires to introduce a new variable that store the previous value of the variable of interest. Annotations can be used to generate new source code, but this approach hardly solves the problem of getting access to the runtime environment. Class members (fields and methods) can be accessed according to introspective capabilities of the reflection package. However, local variables are not supported in the reflection model, turning impossible to write applications that used them.

To approach these problems, we have developed Ënfasis [7], a tool with aspect-oriented facilities for local variables and fine-grain crosscutting. Ënfasis allows storing local variable information in the class file and provides a special notation (signature patterns) for selecting and recovering information of specific local variables. These capabilities do not require annotation processing, avoiding rising most of the problems and limitations discussed before.

The rest of the paper is organized as follows. Section 2 explains the relationship between annotations and aspects. Ënfasis patterns and its language are introduced in section 3. Section 4 presents how Ënfasis does bytecode instrumentation and its API. Section 5 presents the related work and Section 6 points out our conclusions and some directions of future work.

II. ANNOTATIONS AND ASPECTS

Aspect-Oriented Programming (AOP) allows better separation of concerns at development level [4]. AOP techniques rely on the notion of a join point model where aspects can identify principled places in the program execution to alter the program behavior [8]. Such principled places are known as join points and a set of join points with their values are quantified by a pointcut or crosscut. To identify join points, AspectJ and AspectJ-like languages use a pattern matching mechanism based on signatures of classes, fields, methods and constructors [8], [4], [3]. To introduce the pattern matching concepts, consider again the class Testable with some modifications.

```java
public class Testable {
  String s = "An annotated field";
  void m(double d) {
    double old_d = d;
    d += old_d;
  }
}
```

The pattern `String Testable.s` identifies the field `s` of type `String` inside the class `Testable`. The pattern `void Testable.m(double)` select the method `m` with a `double` as argument and which no returns value. The pattern for local variable `d` is `void Testable.m(double).double d`. If we define an annotation for each case, our class could become something like this:

```java
public class Testable {
  @Field String s = "An annotated field";
  @Method void m(@Args double d) {
    @Local double old_d = d;
    d += old_d;
  }
}
```

Considering that a programmer needs to modify any class in the system, it is necessary to do a lot of work. Here is where an aspect takes advantages. Pattern matching mechanism allows using wildcards to specify any possible combination of signatures. If we want to identify any field of type `String` inside the class `Testable`, the pattern is `String Testable.*`. In the same way, any over-headed method `m` with any argument and any return value, the pattern is `* Testable.m(..)`. In this case, the first wildcard matches any return value of `m`, and ".." matches any arguments (also if some version of method `m` takes no arguments). The power of signature patterns has better-quality respect to annotations to identify an element in the class definition, but unfortunately, AspectJ and AspectJ-like languages do not support local variable signatures. Here is where Ënfasis plays an important role to solve these limitations.
III. Énfasis

Énfasis is a model and a tool for local variable crosscutting and it was conceived as a response behind the weakness of several AspectJ-like languages [13], [11], [14], [12] to support local variables. Énfasis join point model retrieves an ordered set of join points from each program fragment along with a minimal set of composition operations defined on them. The model introduces a notation to describe local variable pointcuts with name, type, and occurrence number within a program fragment.

A. Example: tracing local variables

Tracing is a concern intended to retrieve valuable information during program execution. Usually, tracing involves access to local variables in some specific points of interest in the class methods of the program. The search method, shown next, will be used to illustrate the limitations of Java annotations and the advantages of Éfasis with its pattern matching.

```java
@Local(id="lo")
int lo = 0;

@Local(id="hi")
int hi = a.length - 1;

while(lo <= hi) {
  @Local(id="m")
  int m = (hi + lo)/2;
  if(x < a[m])
    hi = m - 1;
  else if(x > a[m])
    lo = m + 1;
  else
    return m;
}
return -1;
```

In order to keep simple the pattern matching is possible to use line numbers or a set of occurrences of local variables. To select some occurrences of local variable m using line numbers: int m#{11,13,16} where # means "at line number" and double dots represents a range. For specific occurrences we have: int m@{1,2} where @ specify the first and second occurrences of m within the given program fragment. Énfasis allows combinations of paths and occurrences.

B. Annotations as pointcuts

The @Local annotation allows marking each local variable definition with an id and a description. The equivalent pattern to do that is:

```
* BinarySearch.search(..).int *{#1}
```

where int * {#1} means that we are interested in the first occurrence of each local variable (its declaration). To identify each pattern we use a Pointcut type.

```java
Pointcut mark =
  new Pointcut("setLocal( * BinarySearch.search(..).int * )");
```

C. Annotation processing as advice

An annotation processor looks for annotations in the program and does some actions based on their values. This processor can be expressed in the context of an aspect as an advice. An advice is the code to be executed at a join point that has been selected by a pointcut [8]. Inside an advice we can use any local variable and any regular Java sentence. Advice can execute before, after, or around the join point. Around advice can modify the execution of the code that is at the join point, it can replace it, or it can even bypass it. The processor looks like next code:

```java
Advice b = new Before(
  "System.out.println("before:");" +
  "System.out.println("after:");" +
  "System.out.println("around");" +
  "System.out.println("bypass");" +
  "System.out.println("pass");" +
  "System.out.println("fail");" +
  "System.out.println("skip");" +
  "System.out.println("skip2");"
);```
m + ":" + a[m]));

To recover information related to each element matched by the pattern we can use the name of the local variable. In this way, the annotation mismatch problem is avoided because the name and the value are available at any instant of execution.

D. The aspect

Énfasis implementation is provided as a Java library. The main advantage for the programmer is the ability to include aspects in the same base language. This approach has successfully been used in other implementations [9]. In Énfasis an aspect is introduced either by extending the Aspect class or by implementing the IAspect interface. The Aspect class implements the IAspect interface. The complete aspect for the @Local annotation is shown below.

```java
01 public class Local extends Aspect {
02 public static void main(){
03 Pointcut p = new Pointcut(
04 "setLocal(*
05 BinarySearch.search(..).int *)");
06 Advice b = new Before(
07 "System.out.println("before:" +
08 m + ":" + a[m]);");
09 Weaver w = new Weaver();
10 w.add(new Crosscut(p,b));
11 w.weave();
12 }
13 }
```

Aspects are introduced in class Trace by inheritance. In lines 3 to 5, a new pointcut is created by defining the pattern that all points of interest must meet, written as a string and ended by a semicolon. Advice Before is created in lines 6 to 8 by defining the actions to be done upon the join points found, in this case, printing the value of variable m. Crosscut at line 10 calls its advice just before the join points of the pointcut are observed. Finally, the weaver begins to weave all crosscuts at line 11. The `weaver` is a compiler-like entity that composes the final system.

E. Pointcut composition

Pointcut composition makes possible to define complex patterns of join points from simpler ones by means of a minimal set of logical connectives. For example, the following pointcut describes a set of all join points where variables lo and hi appear in any expression except those occurring in the condition part of a while statement.

```java
Pointcut pc =
      new Pointcut("public static int " +
      "BinarySearch.search(int[] a, int x) " +
      "/*:getLocal(int lo, hi) &&" +
      "/while/cond::getLocal(int lo, hi)");
```

This description comprises only the join points at line 5 of the `search` method shown in section III-A. Note that logical connectives are interpreted as basic operations on sets as described in the programming model.

IV. ÉNFASIS IMPLEMENTATION AND ITS API

Our implementation requires information from both the source code and the Java bytecode. As AspectJ, we consider each bytecode instruction as a possible join point which is known as a static shadow [19]. Énfasis weaver is implemented using an extension of Javassist, a toolkit for bytecode transformation based on reflection [17], [18]. All information required by the weaver is recovered from the local variable table (LTV) and the line number table (LNT). The LTV table contains information about each local variable like its start, length, slot, type, name and signature. The start and length fields of the table describe the variable scope in the method code. The former marks the initial address of the scope and the latter indicates the extension of the scope. With the slot (or index) of a local variable it is possible to select all points of execution where such variable is accessed for reading or writing.

A. Bytecode instrumentation

As an example of bytecode instrumentation, consider the statement `m = 0`. The compiler generates for this statement the following snippet of code:

```java
17: iconst_0
18: istore_m
```

where the variable name replaces its slot in instruction `istore` for the sake of readability.

A join point can be associated to address 18 that points to the store instruction for `m`. An implementation that supports an advice of type `before` may transform this code into the following by inserting the advising code at the address its pointcut:

```java
17: iconst_0
18: getstatic
    [java/lang/System.out]
21: ldc
    [String before a local variable!]
23: invokevirtual
    [java/io/PrintStream.println]
26: istore_m
```

The inserted code allows tracing the variable using an advising mechanism similar to the one used in AspectJ. The advice `after` implementation follows an analogous strategy. In the case of advice `around`, the instruction is wrapped by the advising code. As the example shows, the advising mechanism of Énfasis is based on knowing the address of the load or the store instruction for any join point.

B. Application programming interface

Énfasis has an architecture based on tiers. The first tier gives information about the structure of class file: packages, classes, fields, methods, local variables, line numbers and data associated to other attributes. The second tier allows bytecode
instrumentation on any instruction. This tier considers that code insertions are free of side effects (the stack is not altered with the new instructions). The third tier lids with aspects. Each tier passes information to the next one in order to create a complete framework to support finer grained aspects and pattern matching for local variables.

With this architecture is possible to use each tier of ´Enfasis as an individual application. Each tier has an application programming interface (API) and its usage. For example, the code below shows how to know about the existence of a specific local variable.

```java
ClassExplorer ce = new ClassExplorer("BinarySearch");
MethodExplorer me = new MethodExplorer(ce);
me.getMethod("int", "search", "int[]", int[]);
if(me.hasVariable("int", "hi"))
  System.out.println(me.getVariable("int", "hi"));
```

Here, ClassExplorer is a class to look into a class file and recover a method of interest. MethodExplorer class allows asking about the existence of a variable. If the variable is found, all data about such variable is stored as an object of type LocalVariable. All classes shown in the example belongs to enfasis.bytecode package. If we want to instrument a class, we use the previous tier aside bytecode instrumentation tier:

```java
ClassExplorer ce = new ClassExplorer("BinarySearch");
MethodExplorer method = new MethodExplorer(ce);
MethodModifier modif = new MethodModifier(ce.getMethod("int", "search", "int[]", int[]));
modif.addLocalVariable("int", "c");
modif.insertBefore("{ c = 0; }");
modif.insertAt(5, "{c++; }" );
modif.insertAt(12, "{System.out.println(c); }" );
ClassModifier klassMod = new ClassModifier(ce);
klassMod.write("c:\\enfasistesting");
```

MethodModifier class enables a class file to instrumentation. This class has many useful methods to support instrumentation in a flexible way. The method addLocalVariable inserts a new local variable, insertBefore allows inserting code at the beginning of a method and insertAt inserts code at a specific line number. ClassModifier is a class for saving any change.

V. RELATED WORK

The JSR-308 [2] proposal describes the extensions to the Java language and classfile format to support annotations on all kind of Java types. It includes annotations on interface implementations, class extensions, typecast, generics, object creation, type test, arrays, exception throws, among other interesting constructions. From the instrumentation point of view, each bytecode instruction and the structure in the class format have a correspondence with the elements previously mentioned. In this sense, Enfasis programming model supports all extensions in JSR 308 except those which are not based in a bytecode instruction. With a minimum of effort it is possible to enable Enfasis to support annotation processing (as advice) for all proposed extensions.

Javassist [17], [18] is probably the most flexible instrumentation tool behind Enfasis. Javassist has many classes to support bytecode instrumentation but Enfasis extends it with a more flexible API to support both, a better control of insertions and a higher and easy level of abstraction to instrument class files.

VI. FUTURE WORK AND CONCLUSIONS

The main contribution of this work is on defining annotation processors for local variables as advice. We showed how pattern matching in Enfasis can reduce the effort to annotate each location in source code with the additional advantage of annotate any occurrence of a variable in a compact way. Also, this approach avoid the mismatch problem found in Java with its target and retention policies for local variables due to in Java this kind of annotations are only markers.

Currently, Enfasis allows store additional attributes in classfile. As future work, this capability will be extended with a complete support for extract source code annotations and store their information in order to take advantages of traditional annotations.

REFERENCES


