A Join Point Model for Fine-Grained Aspects

ULISES JUAREZ-MARTINEZ
Cinvestav - IPN
Department of Electrical Engineering
México, D. F. 07360
MEXICO

JOSE OSCAR OLMEDO-AGUIRRE
Cinvestav - IPN
Department of Electrical Engineering
México, D. F. 07360
MEXICO

Abstract: Crosscutting concerns, defined at the finest level of granularity for statements and expressions, considerably improves expressiveness of pointcut descriptions, making possible to address applications that cannot be approached otherwise. In this paper is introduced a join point model for fine-grained aspects based on path expressions defined upon the Abstract Syntax Tree of the method body. The model includes local variable crosscutting for getting and setting values within an expression. We show the capabilities of this model for an annotation processing concern.

Key–Words: Join point model, Fine-grained aspects, Local variable crosscutting, Annotation processing

1 Introduction

A Join Point Model (JPM) provides the common frame of reference to properly coordinate the execution of both aspectual and non-aspectual code [11]. Most aspect-oriented languages are tied to the AspectJ JPM and they only expose join points around the interfaces of a class. However this kind of expressiveness is not useful for many other important domains. The main problem with the expressiveness of current JPMs and aspect languages is that they do not allow aspect definitions at finer levels of granularity. For example, several type of concerns like program visualization [2], verification, tracing, debugging, testing and monitoring require precision and adaptation in a JPM that addresses crosscutting concerns at the statement and expression levels of the base language [15]. Real-time systems [14] have lengthy methods that typically contain many features intertwined in highly complex patterns that could not be conveniently described with the AspectJ JPM.

The proposals that address these problems include identifying new join points [7], new expressions [3] and annotations [5]. Other approaches are the State-Join Point Model (SJPM) [1] and the Point-in-time Join Point Model (PITJPM) [12]. SJPM uses the transitions of the aspect state to identify the join points during system execution. In PITJPM a join point represents a point-in-time (or an instant of program execution) rather than a region-in-time (or an interval).

Only a few approaches address (in a limited way) a JPM with support for finer granularity. For instance, Usui and Chiba [18] present Bugdel, a specialized tool for debugging based on aspect-orientation. They extend the thisJoinPoint variable with specific information about line numbers and local variables that are visible at the join point. In [16] the authors explore a minimal language design based on fine-grained pointcuts to express structures of the base language through logic meta-variables. Finally, Ubayashi [17] proposes a framework to introduce new kinds of JPMs and extend existing JPMs based on the structure of an AST (Abstract Syntax Tree) where new join points are registered with a hook interface.

In this paper we propose a JPM for fine-grained aspects (FGA) as an implementation of our previous work in providing a formal model for crosscutting on local variables [10]. In this model, it is possible to access local variables, one of the finest granularities available in the source code. The approach brings the possibility to define aspects that crosscut method bodies, statements and expressions in order to solve the inconveniences mentioned before and to address concerns whose functionality resides on white-box testing and fine-grained genericity.

The rest of the paper is organized as follows. Section 2 describes the join point model for fine grained aspects. The implementation of our JPM is described in section 3. Section 4 shows the capabilities of this model for an annotation processing concern. In section 5 we discuss fine-grained genericity and FGA in regard to other related works. Some concluding remarks are given in section 6.
2 JPM for Fine-Grained Aspects

The JPM for the proposed FGA Model uses a minimal extension to Java by introducing labels that mark the local variable positions within expressions and statements. Such an extension can be avoided by merging the labels within comments, but in that follows we keep them uncommented to prevent turning unreadable the program text. The main reason to introduce this extension is by the need of retrieving and manipulating information about the local variables within the code in which they appear.

2.1 Local Variable Crosscutting

In order to present some of the features of our FGA Model, we analyze a case of study for local variable crosscutting and the corresponding part of the formal model [10]. In our FGA Model, high level description of local variable pointcuts is achieved by introducing a source program notation for join points and by binding each join point to its corresponding accessing instruction address. The binding can be produced by visiting in post-order the abstract syntax tree (AST) of the source program and by visiting the bytecode program in a synchronized manner. The synchronization consists in pairing a program fragment (as a subtree in the AST) with the corresponding segment of the bytecode generated by the compiler. As an example, consider the assignment:

\[ m = \text{hi} + \text{lo}/2; \]

that uses only local variables. From this assignment, the statement:

\[ S_3 : m = (L_5 : \text{hi} + L_6 : \text{lo})/2; \]

contains decorations using labels \(S_1\) (for a store instruction), and \(L_5\) and \(L_6\) (for load instructions) over local variables \(m, \text{hi}\) and \(\text{lo}\), respectively, to indicate the join points that are of interest. The AST for this fragment can be written as follows:

\[ L_5 : \text{hi}, L_6 : \text{lo}, +, /, S_3 : m \]

whenever the nodes of the AST are visited in post-order. After comparing this fragment with the bytecode generated by the compiler:

12: load\_hi 14: iadd 16: idiv
13: load\_lo 15: iconst\_2 17: istore 4

it can be shown, for each local variable, that a join point \(L_i (S_i)\) can be bound to the address of the corresponding load(store) instruction. This is possible because the order in which variables occur in the program text is preserved by the compiler in the generated code, except for the assignment statement. In this case, the assigned variable is placed after the expression code:

\[ [L_5 \rightarrow 12, L_6 \rightarrow 13, S_3 \rightarrow 17,...] \]

Binding join point labels to instruction addresses helps to maintain a high level perspective for the pointcut descriptor. The FGA Model introduces a language for pointcut descriptors to recognize patterns of join points detected at run-time. The language incorporates some novel features like path and filter expressions to better the resolution in the join point selection mechanisms.

2.2 Pointcut Descriptor Syntax

The abstract syntax for this Java subset includes only the aforementioned minimal extension that can be hidden during lexicographic analysis. The extension consists on decorating each local variable with a unique label that fix the position of a join point for the variable.

\[
J ::= [L : x = E; | J_1 J_2 | if(B) J_1 \text{else} J_2 | \ldots \\
E ::= c | [L : x = E_1 + E_2 | \ldots \\
B ::= \text{true} | \text{false} | E_1 < E_2 | \ldots | B_1 \text{and} B_2 | \ldots \\
V ::= M[P]::F(\text{int} \; x)|[@N]|#N \\
P ::= P/K | K | + \\
K ::= C[\text{int}] | C[S] \\
C ::= \text{assign} | \text{if} | \text{while} \ldots \\
S ::= \text{then} | \text{else} | \text{do} \ldots \\
F ::= \text{hasLocal} | \text{getLocal} | \text{setLocal} \\
D ::= V | D_1 \text{and} D_2 | D_1 | D_2 | ! D
\]

In the above grammar rules, \(x\) ranks over variable names, \(c\) and \(n\) ranks over the integer and natural numbers respectively, and \(N\) ranks over sets of ordered natural numbers. Syntactic category \(M\) describes a restricted form of method declaration comprising only the method name, type and parameter list. By simplicity types are restricted to either \text{int} or \text{int[]} in any part of a program in which (arithmetic or boolean) expressions may occur. In that follows, syntactical parenthesis \([P]\) denote the ordered set of joint points in program fragment \(P\) (represented by its abstract syntax tree).

2.3 Path Expressions

To explain paths expression we are going to introduce a tracing local variables example. Tracing is a well-
known crosscutting concern intended to retrieve valuable information during program execution. Tracing usually involves access to local variable in some specific points of interest in the class methods of the program. For example, local variables occurring in the statements of a loop statement, and more precisely, the assignments where a local variable is defined. A binary search program is shown next.

```java
01 public class BinarySearch {
02   public static int search(
03       int[] a, int x) {
04       int lo = 0;
05       int hi = a.length - 1;
06       while(lo <= hi) {
07           int m = (hi + lo) / 2;
08           if(x < a[m])
09               hi = m - 1;
10           else if(x > a[m])
11               lo = m + 1;
12           else
13               return m;
14         }
15         return -1; }
```

In order to trace a variable, say m, it is necessary to define: 1) a pattern for matching a specific occurrence (or set of occurrences) within the method, and 2) a path to select the program fragment where the set of local variables occur.

Using AspectJ-like syntax to locate all occurrences of variable m, the pattern:

```
* BinarySearch.search(..)/int m
```

specifies all occurrences of the variable within the body of method search of class BinarySearch.

To locate the variables inside the body of while loop (lines 6 to 13) the pattern is extended with a path:

```
* BinarySearch.search(..)
  /while/do::getLocal(int m)
```

where `getLocal` is the specific designator to apply on such variable. If the `do` path is not included in the pattern, it is also possible to select variables inside the boolean expression in the condition of `while` loop.

Selecting variables of interest by the number of line in which they occur can be described by making explicit the line number.

For example, to select some occurrences of local variable m using line numbers, we can write:

```
int m@{7..9, 12}
```

where @ means “at line number” and double points represent a range. A pattern for filtering variables when they occur in the right-hand side of an assignment (by using `setLocal`) can be:

```
* BinarySearch.search(..)
  ::setLocal(int * @3,4))
```

that allows crosscutting on variables lo and hi at lines 3 and 4, respectively. Finally, the occurrence of variable m at line 12 can be match with the full path as follows:

```
* BinarySearch.search(int[], int)
  /while/do/if/else/if/else/int m
```

### 2.4 FGA Aspects

The interleaving of base and aspectual code, called aspect weaving, is obtained by inserting advice code at the points of interest described by path expressions. Class Trace weaves code of method search with the advice code for the local variable tracing.

```java
01 class Trace extends Aspect {
02   public static void main(){
03     Pointcut p = new Pointcut(
04       "public static int" +
05       "BinarySearch.search(int[]," +
06       "+" + "int x)="/while/ +
07       "do::setLocal(int m);" +
08       "Advice b = new Before(" +
09       "System.out.println(" +
10       ""before:"+m+":"+a[m]);" +
11       "Advice a = new After(" +
12       "System.out.println(" +
13       ""after:"+m+":"+a[m]);" +
14       "Weaver w = new Weaver();
15       w.add(new Crosscut(p,b));
16       w.add(new Crosscut(p,a));
17       w.weave();
18     } 19   }
```

Lines 3 to 7 define a new pointcut. It specifies all points of interest, written as a string ended by a semicolon. The pointcut specifies all the join points in the search method that are located in the first `while` loop. `setLocal` selects all assignments for variable m (line 7). Advices of type `before` and `after` are created at lines 8 to 10 and 11 to 13, respectively. The weaver is created at line 14 and the weaving rules are specified at lines 15 and 16. Finally, line 17 allows weaving process starts.
3 FGA Model Implementation

Working with fine grained aspects requires information from both the source code and the Java bytecode. This combination is followed by other languages like AspectJ to implement its join point model. AspectJ considers each bytecode instruction as a possible join point which is known as a static shadow [9] and our FGA Model uses the same approach.

The JPM for FGA is implemented by extending Javassist, a toolkit for bytecode transformation based on reflection [4]. Javassist allows bytecode instrumentation using Java source code and bytecode instructions. Both approaches are used in the FGA Model to provide a flexible framework for aspect programming in Java.

The weaving process inserts the advice code in some principled places in the bytecodes. All information required by the weaver is retrieved from the bytecode and from both the local variable table (LVT) and the line number table (LNT). The LVT table contains information about each local variable like 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 is possible to select all points of execution where such variable is accessed for reading or writing. As an example of bytecode instrumentation, consider the statement:

```java
/* S: */ m = 0;
```

where label $S$ denotes a join point for variable $m$. The compiler generates for this statement the following snippet of code:

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

where variable name $m$ replaces its slot in instruction `istore` for the sake of readability. Join point $S$ 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 given by $S$:

```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 implementation of the after advice follows an analogous strategy. In the case of the around advice, the instruction is wrapped by the advising code. As the example shows, the advising mechanism of our FGA Model is based on identifying a join point with the address of the load or the store instruction for a local variable. From this address it is possible to find the static shadows that occur just before and after of the join point in order to insert code.

4 Annotation Processing Concern

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 introducing statements at different points of the source code. Working with annotations requires different retention policies to be processed properly: at source code, at class files or at run-time. 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 the 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 they are labelled as invisible by the virtual machine and can be read by instrumentation tools only.

4.1 Annotation Limitations

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

- **Local variables.** Annotations for local variables are only available at source code level and cannot
be used during program execution. Without run-time support, local variable annotations become only markers.

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

- **Annotation mismatch to Java code.** Annotations do not provide the means to get access to the current values of local variables during program execution.

4.2 Annotations and aspects

In order to evidence these problems, let us consider the annotation definition of `@Local`:

```
01 import java.lang.annotation.*;
02 @Target(ElementType.LOCAL_VARIABLE)
03 @Retention(RetentionPolicy.SOURCE)
04 public @interface Local {
05 public int id();
06 public String description()
07 default "";
08 }
```

and the code fragment of binary search with annotations:

```
03 @Local(id="lo") int lo = 0;
04 @Local(id="hi") int hi = a.length-1;
05 while(lo <= hi) {
06   @Local(id="m") int m = (hi+lo)/2;
```

The annotation `@Local` takes two parameters, a required id and an optional description. There is no possibility to recover the value of a local variable because of an annotation processor is independent of the context of program execution (annotation mismatch). Also any change of some value, like a field or variable, is not possible to annotate it.

4.2.1 Annotations as pointcuts

The pattern language of our FGA model presented before is more expressive and effective than annotations in regard to identifying essential AOP features. 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 create a pointcut descriptor with this pattern we use a `Pointcut` type as shown from the following excerpt of the code given in section 2.4.

```
Pointcut mark = new Pointcut( 
   "* BinarySearch.search(..)" + 
   "/:setLocal(int *#{1})");
```

4.2.2 Annotation processing as advice

An annotation processor searches for the annotations inserted in the source program and, if it finds any, it does some actions based on the annotated values. The actions of the processor can be expressed in the tracing aspect as a before advice. The scope of the advice extends to the method scope so any local variable of the method can be used in the advice, as shown next:

```
Advice b = new Before( 
   "System.out.println("before:" + 
   m + ":" + a[m]);");
```

Thus, the information related to each local variable can be retrieved from the joinpoints matching the pointcut descriptor that includes the path to the local variable. This is a novel approach in our FGA Model: the context of local variables is automatically exposed and captured, avoiding introducing extra parameter declarations. In this way, the annotation mismatch problem is avoided because the name and value of any local variable are available within the scope of the variable in the method.

4.2.3 The aspect

The current implementation of FGA Model 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 [13]. In the FGA Model 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.

```
01 public class Local extends Aspect {
02   public static void main(){
03       Pointcut mark = new Pointcut( 
04         "* BinarySearch.search(..)" + 
05         "/::setLocal(int *#{1})");
06       Advice b = new Before( 
```

```
```

```
```
Here is important to point out that at line 9 we are using the automatic exposed context for variable m.

5 Related Work

LogicAJ2 [16] introduces the concept of fine-grained genericity for aspect languages.

Fine-Grained Genericity allows modularization of code coverage and program optimization concerns if aspects are able to address base language entities at the finest level of granularity like individual statements and expressions [8].

LogicAJ provides meta-variables ranging over code blocks but currently it only supports their use for matching entire method bodies. Thus, their designators are not capable for selecting specific join points for local variables neither show source code information. In comparison, our FGA Model allows Fine-Grained Genericity through Local Variable Crosscutting.

Bugdel [18] is an aspect-oriented system designed exclusively for debugging. It extends the thisJoinPoint variable to provide information about the specific line of source code in which a local variable occurs. Our FGA Model takes a step further with a more general albeit precise pattern matching mechanism.

The JSR-308 [6] proposal describes the extensions to the Java language and classfile format to support annotations on all kind of Java classes. In this sense, our FGA 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 our JPM to support annotation processing (as advice) for all proposed extensions in JSR-308.

6 Conclusions

The paper proposes a JPM for FGA to approach crosscutting on local variables, one of the finest granularity inside a method.

The proposed FGA Model uses path expressions to implement fine-grained concerns and provide fine-grained genericity by means of path expressions to locate the fragment of the method body containing the joinpoints of interest.

This novel approach provides new insights in the pointcut description languages currently used in AOP.

References: