

# Using AspectJ to Collect Value-Spectra

Mohamed Mansour  
CS 8803H Spring 2003  
College of Computing  
Georgia Institute of Technology  
mansour@cc.gatech.edu

## ABSTRACT

In this work we will explore the use of *Aspect Oriented Programming* (AOP) in program analysis. We will specifically explore the use of *AspectJ* for collecting runtime trace information about program execution. In addition to collecting profile data about program coverage, this work will also focus on collecting information about program state at each call site. By collecting the values of variables on the argument list at each call site we can construct a value-spectra. Value-spectra can be used in fault-localization [1] techniques and to increase regression fault-exposure probability. The classical approach to instrumenting Java code for coverage and program-state collection is to use byte-code rewriting tools. While these tools can be very powerful because they give full access to the Java byte-code, they require one to learn the intricacies of Java byte-code. AspectJ on the other hand offers a set of APIs that are very intuitive to Java programmers.

## 1. BACKGROUND

Code instrumentation is a technique that inserts extra statements in a program. These extra statements can provide additional functionality that was not designed in the original program. Instrumentation, for example, could be used to inject extra statements that collect coverage information, track memory allocation and de-allocation to detect memory leaks, or to collect performance profiling data<sup>1</sup>. Instrumentation can be done at the source code level, by using a preprocessor. For Java programs it can also be done at the byte-code level using byte-code-rewriting techniques. The instrumented probes could be added statically or dynamically [4]. Collecting program coverage data is used in a variety of fields. In software testing, coverage data can be a measure of the adequacy of the test cases [5]. Certain analysis techniques use coverage data and information about the changes made to a program to select the relevant tests from a regression test suite [2, 3].

<sup>1</sup><http://www.rational.com/products/pqc>

BCEL and SOOT are two common byte-code rewriting tools. *BCEL* (Byte Code Engineering Library), formerly known as *JavaClass*, is an open source tool that is part of the Apache Jakarta project<sup>2</sup>. *SOOT* is a similar tool that offers more intermediate representations of the byte-code<sup>3</sup>. SOOT is free software that is licensed under GNU's LGPL.

## 2. OVERVIEW OF ASPECTJ

AspectJ is a general purpose Aspect Oriented Programming (AOP) language that was developed at Xerox PARC. In December 2002 PARC decided to move AspectJ to the Eclipse open source project<sup>4</sup>. AspectJ was designed to be a seamless extension to the Java language. AspectJ allows developers to catch certain points during program execution and execute additional code at these points. These interception points are referred to as *join points*. A join point can be: calling a method (dynamic or static), object constructor, throwing an exception, reading or setting the value of a variable or a data member, and many more<sup>5</sup>.

Table 1 lists some pointcuts and their description. Exact names or pattern matching can be used in *Signature* and *Type*. The wild card \* matches zero or more characters except for .. The two dots (..) pattern matches any sequence of characters that start and end with a .. Modifiers such as *static* and *public* can also be used to narrow the scope of the pattern. Table 2 lists the pointcuts we used in this paper.

A *pointcut* is a collection of join points. An example of a pointcut is "all calls to method of class X" or "exception throw sites in package Y". The user can insert additional code before and/or after the join points in a pointcut. This additional code, which is plain Java code, is referred to as an *advice*. AspectJ provides reflection APIs for accessing the execution context from within the advice. So, the 'this' pointer, argument lists, source location of the join point, exception type, ... are all available. An *aspect* is a collection of join points, pointcuts and advices. Aspects are used to encapsulate a particular concern about the program. For example, one can add an aspect to intercept every method call in the system and log it to a trace file.

<sup>2</sup><http://jakarta.apache.org/bcel/>

<sup>3</sup><http://www.sable.mcgill.edu/soot/>

<sup>4</sup><http://www.eclipse.org/aspectj>

<sup>5</sup>for a complete reference of AspectJ 1.1 join points refer to <http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/apa.html>

A special compiler *ajc* is provided with AspectJ to compile the Java source code and the aspects. Currently, AspectJ supported compile time static weaving only. The development community is considering dynamic load-time class weaving.

**Table 1: A sample of AspectJ pointcuts**

Pointcut	Description
<code>call(<i>Signature</i>)</code>	Picks out each method call join point whose signature matches <i>Signature</i>
<code>target(<i>Type</i>)</code>	Picks out each join point where the target object (object which a call or field operation is applied to) is an instance of type. Will not match any join points from static contexts.
<code>within(<i>TypePattern</i>)</code>	Picks out each join point where the executing code is defined in a type matched by <i>TypePattern</i>
<code>!<i>Pointcut</i></code>	Join points that are not picked out by <i>Pointcut</i>
<code><i>Pointcut0</i>&amp;&amp;<i>Pointcut1</i></code>	Join points that are picked out by both <i>Pointcut0</i> and <i>Pointcut1</i>
<code><i>Pointcut0</i>  <i>Pointcut1</i></code>	Join points that are picked out by either <i>Pointcut0</i> or <i>Pointcut1</i>

**Table 2: Pointcuts used in this work and their explanation**

pointcut	Description
<code>call (static * * (..))</code>	picks calls to static methods that return any type with any name and any argument list. i.e. all static methods
<code>call (!static * * (..))</code>	picks calls to all non-static methods
<code>within(java..*)</code>	inside any method in the java package
<code>call(* java..*(..))</code>	picks any call to any method in the java package

### 3. ENVIRONMENT

We used AspectJ 1.1rc1 with Sun's JDK 1.4.1\_01. All experiments were executed on the College of Computing machines at GA Tech. We used Intel boxes running Linux 7.3, and NFS mounted file systems. The machines we used were helsinki, a Dell PowerEdge 1400 (2 x 800 MHz Pentium III, 1 GB memory) and mikkeli (1.2 GHz Pentium III, 896 MB memory). Performance results presented in this paper use the Unix time utility, we used the following three measure for a program run:

*real* Elapsed real time (in seconds).

*user* Total number of CPU-seconds that the process spent in user mode.

*sys* Total number of CPU-seconds that the process spent in kernel mode.

All experiments were run during a period of low activity when the machines had almost no additional load. Each experiment was repeated 20 times and the results presented are the average of these runs.

### 4. COLLECTING VALUE-SPECTRA COVERAGE

Collecting value-spectra information amounts to capturing partial program state at different execution points over the program life time and can result in large amounts of trace information. We have developed a set of aspects to collect value-spectra for a program execution. These aspects use a range of techniques to encode and compress the value-spectra data for efficient storage. We present in this section the different techniques used and performance measurements of each technique.

#### 4.1 Aspect for collecting value-spectra

Figure 1 shows the skeleton of trace aspect we used to collect value-spectra.

The `methodExec` pointcut select all entry points to any method in the NanoXML package. The `!myTrace` condition makes sure we do not intercept calls to methods defined in the aspect, the `!javaCode` condition filters off calls to standard Java libraries. The aspects define an advice that will be executed at the entry of each method and another one to be executed after returning from each method call. In the `before()` advice, we extract the value-spectra signature and push it on a stack. In the `after() returning(Objectt)` advice we again capture the value-spectra signature, we also pop the entry-point signature and log both to an external file. The code for capturing the signature is not shown here for space reasons. Full source code is submitted with this report and can be made available on request.

## 4.2 Log format

In this section we will present the format we used for logging the value-spectra and function entry and exit. Function name is the fully qualified function name and signature. For arguments of basic types, we log their values directly. For arguments of class types, we log the values of their data members using the following format:

```
[Type]<m_1>, <m_2>, ... [Type].
```

where `Type` is the fully qualified name of the class type and `m_1` and `m_2` are the data member of the class. `<m_1>` denotes the value of the data member `m_1`. are values of the data members of the class (regardless of access level). If a data member is an object itself, we recursively encode its value.

For example, a simple class that has the following declaration:

```
Class SimpleClass {
    int a;
    int b;
}
```

and assuming an object of `SimpleClass` with `a=1` and `b=2`, it is logged as:

```
[SimpleClass]1, 2[SimpleClass]
```

As another example, consider the class below.

```
ComplexClass {
    float c;
    SimpleClass d;
}
```

Assume we have an object of this class with `c=3.5`, and `d` is the object presented above. Its value is logged as:

```
[ComplexClass]3.5, [SimpleClass]1, 2[SimpleClass][ComplexClass]
```

We order class data member by the order implied by the Java reflection APIs. So, to illustrate the whole process, assume a call to `int foo(SimpleClass sc)`:

```
public int foo (SimpleClass s) {
    s.a = 0;
    s.b = 0;
    return 0;
}
```

A call to `foo` with `s (3, 4)` will be logged as:

```
public int foo (SimpleClass)
[SimpleClass]3, 4[SimpleClass]
[SimpleClass]0, 0[SimpleClass]
0
```

**Figure 1: Source list for value-spectra coverage aspect**

```
public aspect BaseAspect {
    pointcut myTrace(): within(BaseAspect) ;
    pointcut javaCode(): within(java..*)
        || call(* java..*(..));
    pointcut methodExec():
        execution(* net..*(..))
        && !javaCode()
        && !myTrace();

    before (): methodExec() {
        String enter = getValueSpectra(thisJoinPoint);
        stack.push(enter);
    }
    after () returning (Object t): methodExec() {
        String enter = stack.pop();
        String exit = getValueSpectra(thisJoinPoint, t);
        log(enter, exit);
    }
}
```

## 4.3 Different techniques to collect value spectra

In this section we will present the different techniques we propose for collecting value-spectra and efficiently storing them.

1. This is the baseline for our experiment, this aspect is useful if you want readable log files. We will refer to this aspect as `value1`. For the example cited before for a call to `foo()`, it will be logged as:

```
public int foo (SimpleClass)
[SimpleClass]3, 4[SimpleClass]
[SimpleClass]0, 0[SimpleClass]
0
```

2. In this aspect, we log the hash value of function signatures. All other values are logged as in `value 1`. The hash tables used are saved to an external file at the end of the run. We will refer to this aspect as `value2`. For the example cited before for the call to `foo()` it will be logged as:

```
123
[SimpleClass]3, 4[SimpleClass]
[SimpleClass]0, 0[SimpleClass]
0
```

Here 123 happens to be the index for `publicintfoo(SimpleClass)` in the function signatures file.

3. In this aspect, we log the hash value of class signatures. All other values are logged as in `value2`. The hash tables used are saved to an external file at the end of the run. We will refer to this aspect as `value3`. For the example cited before for the call to `foo()` it will be logged as:

```

123
[45]3, 4[45]
[45]0, 0[45]
0

```

Here, 45 happens to be the index for `SimpleClass` in the class signatures file.

- In this aspect, we log the hash value of object signatures. All other values are logged as in value2. The hash tables used are saved to an external file at the end of the run. We will refer to this aspect as value4. For the example cited before for the call to `foo()` it will be logged as:

```

123
<1>
<2>
0

```

Here, 1 happens to be the index of the `[45]3,4[45]` in the object signatures file and 2 is the index of `[45]0,0[45]`.

- This aspect is identical to value4. The output is compressed using the LZ77 compression algorithm. This is useful if log file sizes are of concern. To read the log files one will have to decompress them first. Compression is done on the fly using Java's `GZIPOutputStream` class. We will refer to this aspect as value5.

Table 3 summarizes the above techniques.

**Table 3: Various ways to collect value-spectra**

aspect	Description
No aspect	The program runs with no aspects. This is the baseline case
value1	All value spectra is logged as is, no hashing or compression used
value2	Same as value1, function signatures are hashed
value3	Same as value2, Class names are hashed
value4	Same as value3, object signatures are hashed
value5	Same as value4, the Java GZIP compression APIs are used to compress the output

To evaluate this technique we experimented with NanoXML<sup>6</sup>. NanoXML is a small XML parser for Java. We applied the various techniques listed above to the entire regression test suite for NanoXML 2.2.3. Table 4 lists the total output log file size for each technique. Table 5 lists the runtime performance.

<sup>6</sup><http://nanoxml.n3.net>

**Table 4: Output file sizes for the different value-spectra collection techniques**

aspect	Output filesize
value1	824KBytes
value2	688 KBytes
value3	430 KBytes
value4	305 KBytes
value5	92 KBytes

**Table 5: Runtimes for the different techniques to collect value-spectra.**

aspect	real	user	sys
No aspect	2.2805	1.6955	0.1245
value1	7.8745	6.5495	0.297
value2	8.5745	6.768	0.3395
value3	8.2125	6.6895	0.2925
value4	8.186	6.6365	0.291
value5	7.889	6.621	0.2705

## 5. BENCHMARKING

The results in the previous section show over 150% increase in run-time when using AspectJ to collect value-spectra. In this section we will perform a series of experiments to identify the source of this overhead. We have developed a set of three aspects for our experiments, all three aspects have identical pointcuts to `BaseAspect` in figure 1.

- Benchmark0** The first aspect contains empty advices. The purpose of this aspect is to measure the overhead imposed by the pointcuts in our value-spectra aspect without the extra overhead of collecting the value-spectra and logging them.

```

before (): methodExec() {
}
after () returning (Object t): methodExec() {
}

```

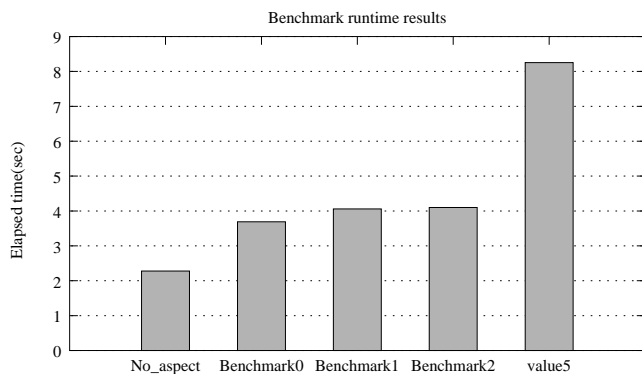
- Benchmark1** The third aspect only makes a reference to `thisJoinPoint`.

```

before (): methodExec() {
    JoinPoint jp = thisJoinPoint;
}
after () returning (Object t): methodExec() {
    JoinPoint jp = thisJoinPoint;
}

```

`thisJoinPoint` is also defined by AspectJ and offers full reflective access to the join point. `thisJoinPoint` gives you access to the `this` pointer of the currently executing object, the source location corresponding to this join point, the signature at this join point, and the argument list, target object, and current object. The value-spectra aspect must make a reference to this object to access functions' argument list. The overhead imposed by accessing this object is the penalty we have to pay for using AspectJ.



**Figure 2: Elapsed real time for benchmarking experiments**

3. Benchmark2 This aspect uses `thisJoinPoint` and additionally access the argument list. This aspect will incur the overhead of marshaling the arguments.

```

before (): methodExec() {
    Object[] args = thisJoinPoint.getArgs();
}
after () returning (Object t): methodExec() {
    Object[] args = thisJoinPoint.getArgs();
}

```

We ran four experiments, the first one did not use any aspects, the purpose was to get a baseline measurements. Each of the remaining experiments used one the aspects described above. We used the same subject, NanoXML 2.2.3 and its regression test suite, as in the previous section. We also reran the value5 experiment and included its results here for comparison. Table 6 lists the results of these experiments. The elapsed real time is plotted in figure 2

**Table 6: Benchmarking of AspectJ overhead**

aspect	real	user	sys
No aspect	2.2785	1.729	0.1065
Benchmark0	3.6905	2.461	0.17
Benchmark1	4.0595	2.991	0.182
Benchmark2	4.099	2.988	0.178
value5	8.2525	6.7275	0.2655

From these results we can see that the total overhead of collecting value-spectra in our technique is roughly 270%. The instrumentation added by AspectJ contributes 30% of the overhead and the remaining 70% is due to actual logging of the value-spectra (includes using Java reflection APIs to collect object signatures).

## 6. SUMMARY AND CONCLUSION

We have presented a case study for using AspectJ in program analysis. The language in its current state is rich enough to allow for many interesting possibilities. We would like to see more features added to the language, e.g. join points at the statement level. The overhead of the current AspectJ implementation can be prohibitive in some cases. We encourage

the AspectJ community to reduce runtime overhead in future releases.

## 7. FUTURE WORK

This project opens a wide range of possibilities for future work. The value-spectra collection techniques collect function arguments only. This can be extended to collect all accessible static variables. This list of accessible static variables at each function entry can be easily obtains through static analysis. The value-spectra data can be used in further analysis, like invariant detection and bug localization. The format we used to represent the value-spectra should be adapted to the application using the data.

## 8. REFERENCES

- [1] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. ACM Press, 2002.
- [2] Sandra Rapps and Elaine J. Weyuker. Data flow analysis techniques for test data selection. In *Proceedings of the 6th international conference on Software engineering*, pages 272–278. IEEE Computer Society Press, 1982.
- [3] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.
- [4] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the international symposium on Software testing and analysis*, pages 86–96. ACM Press, 2002.
- [5] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.