Profiler Instrumentation Using Metaprogramming Techniques

Ritu Arora, Yu Sun, Zekai Demirezen, Jeff Gray

University of Alabama at Birmingham
Department of Computer and Information Sciences
Birmingham, AL 35294

{ritu, yusun, zekzek, gray}@cis.uab.edu

Abstract

Software developers are frequently required to address evolving stakeholder concerns, which often result in changes to the source code of an application. Manually performing invasive modifications across a large code base can be tedious, time consuming, and error prone. Metaprogramming techniques assist a developer in specifying the changes needed to an application in a manner that does not require manual adaptation of source files. Various forms of metaprogramming exist, including compile-time metaobjects, load-time structural reflection, and aspect-oriented programming. In this paper, a profiler is implemented as a common case study using three different approaches to demonstrate the various mechanisms and benefits offered by metaprogramming.

1. Introduction

Successful software applications are often subjected to frequent change, which may come in the form of adaptive maintenance (i.e., migrating an application to a different execution platform) or perfective maintenance (i.e., improving the performance or other quality attributes of an application). Another driving force for change comes from evolving stakeholder requirements (i.e., customer change requests driven by new business rules or feature additions). Depending on the initial design of an application, new change requests may require a developer to invasively modify many source files to implement the change. In the presence of crosscutting concerns, where a concern is implemented across multiple modularization boundaries, implementing a new change can be tedious, error prone, and time consuming. For large applications, code evolution in the presence of crosscutting concerns cannot be performed in a reliable and productive manner using a manual invasive approach.

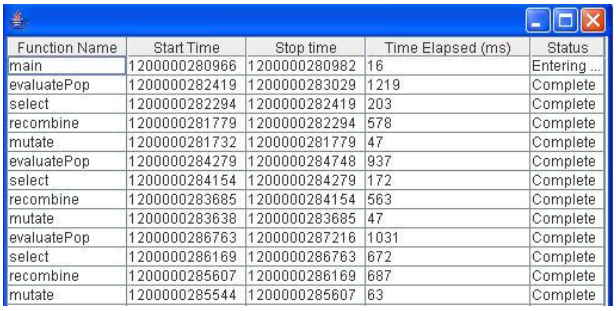
A metaprogram is a program that manipulates another program in order to transform or generate some new feature or analysis capability. A metaprogram can assist in the separation of a change request such that the required changes are isolated from the base features of the application. There are several approaches toward metaprogarmming, such as the following:

- *Compile-time metaobjects*: A metaobject is an object that defines the interpretation of another object; a metaobject is an instance of a metaclass. Metaobjects can be attached to the objects in a base application to enable some type of transformation that realizes a change request. Metaobjects can be processed at compile-time to influence the generation of code that modifies the meaning of a base application class. An example language for exploring compile-time metaobjects in Java is OpenJava [4], which is introduced in Section 2.1.
- *Load-time class adaptation*: By customizing the Java class loader, it is possible to modify the structure of a class (i.e., through bytecode manipulation) before it enters into the available class pool of an executing application. Javassist [9] (see Section 2.2) provides a library of transformation routines that can be used by a metaprogram to realize changes to an existing application represented in bytecode.
- *Aspect-oriented programming*: Aspect-oriented programming (AOP) [1] builds upon the objectives of metaprogramming by providing a language-based approach toward capturing crosscutting concerns into a new modularization unit called an aspect. AspectJ [3] is one of the most used AOP languages for Java and is discussed in Section 2.3.

The three approaches outlined above represent fundamentally different mechanisms for assisting in the separation of change requests that are crosscutting. The next section summarizes the common case study that will be used as a comparative analysis of the various approaches to metaprogramming.

1.1 Case Study: Profiler Instrumentation

A profiler is an essential tool for code optimization and performance enhancement. Profilers provide run-time information about the code (e.g., the amount of time spent in every method and the number of times a method was called). This information can be used for performance tuning and optimizations.



Function Name	Start Time	Stop time	Time Elapsed (ms)	Status
main	1200000280966	1200000280982	16	Entering ...
evaluatePop	1200000282419	1200000283029	1219	Complete
select	1200000282294	1200000282419	203	Complete
recombine	1200000281779	1200000282294	578	Complete
mutate	1200000281732	1200000281779	47	Complete
evaluatePop	1200000284279	1200000284748	937	Complete
select	1200000284154	1200000284279	172	Complete
recombine	1200000283685	1200000284154	563	Complete
mutate	1200000283638	1200000283685	47	Complete
evaluatePop	1200000286763	1200000287216	1031	Complete
select	1200000286169	1200000286763	672	Complete
recombine	1200000285607	1200000286169	687	Complete
mutate	1200000285544	1200000285607	63	Complete

Figure 1. Profiler GUI used in the comparative implementations

To assist in comparing the three metaprogramming techniques investigated in this paper, we have implemented a very basic profiler in each technique. A common case study based on the same profiler classes enables a comparative evaluation of the techniques. A set of classes that implement the profiler GUI are shared by all three metaprogramming implementations of the profiler. The profiler simply captures the details about all the methods that are called in a program and the time spent in each method. All the information is presented to the user through a graphical user interface (GUI). Figure 1 shows the GUI for displaying the profiling information. The GUI shows the function names, time spent in each function, execution status and presents a call graph.

The challenge of implementing a profiler comes from the need to instrument an existing legacy application with the hooks needed to record the amount of time spent in each method. Typically, at every point where a method call can be made, the existing application must be modified to add the profiler concern. For a large application, it is not feasible to manually modify the location of every method call. Metaprogramming provides the capability to isolate the profiling concern into a single module.

The rest of this paper is organized as follows. Section 2 introduces the three metaprogramming techniques that are compared in the paper, where each sub-section provides an overview of the profiler implementation in each respective technique. Section 3 offers a discussion of lessons learned from the implementation experience described in Section 2. Concluding remarks and a summary of our findings are offered in Section 4.

2. Profiler Implementation Examples

This section introduces three different approaches to metaprogramming. Each sub-section provides an outline of the profiler implementation written in each approach.

2.1 OpenJava

OpenJava is an extensible meta-language based on Java, developed at the University of Tsukuba. The OpenJava MOP (Metaobject Protocol) is an extension interface of the Java language, which enables programmers to customize the language to implement new language constructs to enhance the reflective ability of Java.

2.1.1 Introduction to OpenJava

The key objective of using OpenJava is to produce a metaobject that represents the logical structure of a class definition for each class in the source code. In addition, metaobjects also contain instructions to modify the definition and behavior of a class. Figure 2 is an overview of OpenJava usage, where the base program is separated from the metaprogram using a pre-processing technique that generates intermediate Java code that is compiled by a traditional Java compiler [4].

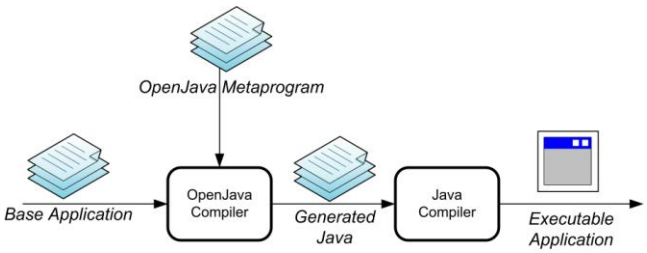


Figure. 2: Overview of OpenJava for compile-time metaobjects

A simple example of using OpenJava to add tracing methods in a Java program is presented in Figure 3. This metaclass can be associated with any base program (in this case, a simple "Hello World" program in Figure 4). Line 1 of the code in Figure 4 has a clause after the class definition (i.e., "`instantiates TracingClass`"), which means that the `Hello` class specifies an instance of `TracingClass` as its metaobject.

The metaprogram (i.e., `TracingClass` shown in Figure 3) contains a method called `translateDefinition`, whose purpose is to examine every method in the class of the base program and add an output statement to display the name of the method that is being executed. In Figure 3, `getDeclaredMethods` returns the list of methods within a specific class in the base program. The `makeStatement` method creates a new program statement that can be inserted into a specific location within a method of the base program.

Figure 5 shows the general Java program generated by the OpenJava compiler. The two print statements in Line 3 and Line 7 are added automatically by the OpenJava compiler after translating the metaprogram of `TracingClass` as applied to the `Hello` class.

```
public class TracingClass instantiates Metaclass
                          extends OJClass {
  public void translateDefinition()
  throws MOPException {

    OJMethod[] methods = getDeclaredMethods();
    OJMethod m;

    for (int i = 0; i < methods.length; ++i) {
      m = methods[i];
      Statement printer =
        makeStatement("System.out.println( \"" +
                      m + " is called.\" );" );
      m.getBody().insertElementAt(printer,0);
    }
  }
}
```

Figure 3. OpenJava program to add tracing (adapted from [4])

```
1.public class Hello instantiates TracingClass {
2.   public static void main(String[] args) {
3.      hello();
4.   }
5.   static void hello() {
6.      System.out.println( "Hello, world." );
7.   }
8. }
```

Figure 4. Base-level program

```
1. public class Hello {
2.   public static void main(String[] args) {
3.     System.out.println("main is called.");
4.     hello();
5.   }
6.   static void hello() {
7.     System.out.println("hello is called." );
8.     System.out.println( "Hello, world." );
9.   }
10. }
```

Figure 5. Generated Java program

2.1.2 Profiler Implementation Using OpenJava

The class that implements the profiler GUI provides two static methods that are called to record the time spent in each method of the base application; the names of these profiler methods are `Profiler.start(`*`MethodName`*`)` and `Profiler.end()`. To enable profiling on the base application, the `Profiler.start` method must be inserted at the beginning of each method of the base application to start the timer. The `Profiler.end()` method should be inserted at the end of every method to stop the timer. Similar to the tracing metaobject shown in Figure 3, a new metaobject can be defined to insert the calls to the profiler statements, as shown in Figure 6.

```
public class ProfilerMeta instantiates Metaclass
                      extends OJClass{
public void translateDefinition()
throws MOPException {

    OJMethod[] methods = getDeclaredMethods();
    OJMethod m;  // each method in application
    int msize;   // number of statements in method

    for (int i = 0; i < methods.length; ++i) {
      m = methods[i];

   // Show Profiler in main method
    if(m.getName().equalsIgnoreCase("main"))
     Statement showGUI =
         makeStatement("{Profiler.show();}");
      m.getBody().insertElementAt(showGUI, 0);

   // Insert Profiler.start() at method front
     Statement start =
         makeStatement("{Profiler.start(\"" +
                        m.getName() + "\");}");
     m.getBody().insertElementAt(start, 0);

   // Insert Profiler.end() at method end
      Statement end =
        makeStatement("{Profiler.end();}");
      m.getBody().insertElementAt(end, msize);

    }
  }
}
```

Figure 6. Profiler instrumentation in OpenJava

In Figure 6, the `ProfilerMeta` metaclass obtains the list of all methods in the associated application class (stored in the `methods` array), iterates over each method and adds the `start` and `end` calls of the `Profiler`. The `main` method of an application is modified to initiate the display of the profiler GUI. The metaclass shown in Figure 6 is actually a subset of the instrumentation required, but for brevity, this listing captures the essence of using OpenJava to instrument a base application with profiling information. In the complete implementation of this metaobject, if the return type of the method is `void`, the profiler `end` statement can be inserted after the last statement of the method. However, if there is a `return` statement somewhere within the method (perhaps even in the middle of the method), the profiler `end` statement must be inserted before the `return` statement, so that the profiler can be stopped properly. This adds additional complexity to an OpenJava solution. More explanation on this limitation will be given in Section 3.

2.2 Javassist

Java supports a weaker form of reflection known as introspection, which allows inquires into the structure of a program but does not allow structural changes to be made to a program [5]. Several extensions to Java have been proposed to allow the more powerful form of introspective reflection [6, 7, 8]. These extended capabilities are categorized according to structural and behavioral reflection, where structural reflection provides the ability to alter data structures in programs and behavioral reflection permits a program to alter the behavior of operations. Javassist is a class library for structural reflection in Java [9].

2.2.1 Introduction to Javassist

In Javassist, to avoid performance degradation, structural reflection is performed by bytecode transformation at either compile-time or load-time. Although Javassist can perform bytecode transformation, those who write Javassist programs do not have to deal with Java bytecode representation because the Javassist API provides source level abstraction over the complexities of lower level bytecode manipulation.

Because intercepting and instrumenting class files are based on bytecode transformation at load-time, Javassist performs its adaptation through a customized class loader. The programmer of a class loader can alter the existing Java class structure or can define new classes. Figure 7 and Figure 8 enumerate some of the Javassist methods for introspection and instrumentation.

	Method Name	Description
`CtClass`	`CtField[] getDeclaredFields()`	Gets the fields of a specific class
`CtField`	`CtMethod[] getDeclaredMethods()`	Gets the methods declared in the class
`CtMethod`	`CtClass[] getParameterTypes()`	Returns the type of method parameters

Figure 7. Subset of Javassist introspection methods

	Method Name	Description
`CtClass`	`void addMethod(…)`	Add a new method to a class
`CtField`	`void bePublic()`	Make the field public
`CtMethod`	`void setBody(…)`	Substitute a method body

Figure 8. Subset of Javassist class instrumentation methods

The following tracing example illustrates the usage of the Javassist APIs. In Figure 9, `TracingClassLoader` is a metaclass where the structure of classes is altered at load-time. The method `loadClass` is inherited from the `ClassLoader` and represents the mechanism for adaptation.

```
public class TracingClassLoader
             extends ClassLoader {

  public Class loadClass(String arg0)
         throws ClassNotFoundException {

    CtClass c = ClassPool.getDefault().get(arg0);
    CtMethod [] methods = c.getDeclaredMethods();

    for (int i = 0; i < methods.length; i++) {
        CtMethod m = methods[i];
        m.insertBefore("System.out.println(\"" +
                    m.getName()+" is called\");");
  }
  return c.toClass();
}
```

Figure 9. Javassist example to add tracing

The first step in using Javassist is to load the desired class from the class pool, which is a heap representing all classes that have been loaded. In Figure 9, a specific class name is passed as an argument and the `ClassPool` is queried. After the specific application class has been obtained, the remainder of the `loadClass` method iterates over all declared methods within the loaded class and inserts a tracing statement at the beginning of each method in the loaded class. The loaded class is then converted to a standard Java reflective `Class`.

2.2.2 Profiler Implementation Using Javassist

```
public class ProfilerClassLoader
             extends ClassLoader {
public Class loadClass(String arg0)
       throws ClassNotFoundException {

  CtClass c = ClassPool.getDefault().get(arg0);
  CtMethod[] methods = c.getDeclaredMethods();

  for (int i = 0; i < methods.length; i++) {
    CtMethod m = methods[i];

    if(m.getName().equalsIgnoreCase("main"))
      m.insertBefore("new Profiler.show();");

    m.insertBefore("Profiler.start(" +
                   m.getName() + "\");");

    m.insertAfter("Profiler.end();"; +

  }

  return c.toClass();

 }
}
```

Figure 10. Profiler Implementation in Javassist

The Javassist profiler metaclass is shown in Figure 10. The `ProfilerClassLoader` implementation extends the `ClassLoader` class and overrides the `loadClass` method to implement structural bytecode transformation at load-time. The `loadClass` method first obtains a `ClassPool` object and retrieves a corresponding `CTClass`, which specifies the representation of the class file and enables modification of the class definition. The `getDeclaredMethods` method is called to obtain a list of all the defined methods of the class. All method implementations are modified using Javassist's API to place the profiler start and end method calls in the appropriate location.

2.3 AspectJ

When a concern such as profiling is spread across all methods within multiple modules of an application, it is known as a crosscutting concern [1]. Such concerns are difficult to modularize using traditional object-oriented languages because each concern is scattered across modularity boundaries, making it hard to maintain and reuse. In order to make a single change in a crosscutting concern, it becomes necessary to replicate the changes at multiple places. Aspect Oriented Software Development (AOSD) [2] techniques provide an alternative to this type of invasive reengineering. Aspect-oriented programming (AOP) offers a new type of modular construct that separates crosscutting concerns, which may improve the coupling and cohesion of an application. The crosscutting concerns are isolated in modular units called aspects that can be weaved into an application as needed. AspectJ [3] is a popular AOP language for Java.

2.3.1 Introduction to AspectJ

The following sub-sections give a brief introduction to the basic concepts related to AOP using AspectJ [3] and show how AspectJ can be used to instrument a program with profiling information without making manual changes to existing code. As modular units, aspects may contain attributes and methods. Like classes, an aspect can inherit from other aspects. The following are key language concepts in AspectJ:

- A *join point* is a location in a program where a crosscutting concern emerges. Example join points in AspectJ are method call and method execution points.
- A *pointcut expression* is a logical predicate that captures a set of join points. A pointcut quantifies over the code base to identify the common location of a specific set of join points. For example, a pointcut expression that contains the predicate "`execution(public * *(..))`" would capture all join points on public methods where the return type, method name, and parameters are wild cards that match to anything.
- An *advice* specifies the actions that capture the implementation of a crosscutting concern. In AspectJ, an advice is associated with pointcut expressions that are contained within a named aspect. Advice may execute *before*, *after*, or *around* a join point.

Figure 11 illustrates the use of AspectJ to specify the tracing example (AspectJ keywords are in bold). The `thisJoinPoint` construct is used to expose the context of a specific join point. In Line 1 of the code shown in Figure 11, an aspect called `TraceClasses` is declared. Line 2 and Line 3 define the pointcuts named `myClass` and `myMethod`, where `myClass` is a "within" type of pointcut that matches all join points within the class `ClassToTrace`. Similarly, `myMethod` is an "execution" pointcut that has wildcards matching all method executions. In Line 4,

an "around" advice is specified that has an `Object` return type. The code in this advice (Lines 6, 7, 8) is executed instead of the actual code defined at the join point. However, Line 8 continues with a "proceed" that permits the execution to continue from the original join point. In summary, "around" advice obtains the method signature (`thisJoinPoint.getSignature`) and prints the method name before and after the execution of the original method.

```
1: aspect TraceClasses {
2:  pointcut myClass(): within(ClassToTrace) ;
3:  pointcut myMethod(): myClass() &&
4:                       execution(* *(..));
5:  Object around (): myMethod() {
6:    System.out.println("Entering"  +
7:        thisJoinPoint.getSignature() +
8:         "method");
9:    Object o = proceed();
10:   System.out.println("Exiting"  +
11:         thisJoinPoint.getSignature() +
12:         "method");
13:   return o;
14:  }
15: }
```

Figure 11. AspectJ code for tracing

2.3.2 Profiler Implementation using AspectJ

```
1.  public aspect ProfileClasses {
2.
3.  pointcut everyMethod() : call(* *.*(..));
4.
5.   before(): call(void *.main(..)) {
6.     new Profiler.show();
7.  }
8.
9.   before() : everyMethod() {
10.    Signature s = thisJoinPoint.getSignature();
11.    Profiler.start(s.getName());
12.  }
13.
14.  after() returning : everyMethod() {
15.    Profiler.end();
16.  }
17.
18. }
```

Figure 12. AspectJ code for profiling

Profiling is a crosscutting concern that can be modularized using AspectJ by weaving the profiler code into the appropriate join points representing method boundaries, as shown in Figure 12. The `ProfileClasses` aspect records the time spent in each function and presents a call graph to the user. In this aspect, the `everyMethod` pointcut expression on Line 3 captures all method calls made in a particular application (i.e., all method calls are within the purview of the profiling aspect). The advice on lines 5 through 7 initializes the display of the profiler GUI in the `main` method of an application. Line 5 represents an anonymous pointcut (i.e., the pointcut is not formally named, such as the one on Line 3) associated with the "before" advice. In Line 9 of the code shown in Figure 12, a "before" advice is specified. This "before" advice on method call join points is executed before the execution of each method. The "before" advice inserts the `Profiler.start` method before the beginning of each method of the base application to start the timer. In Line 14, an "after" advice is specified that executes after the program returns from the methods matched in the join point. This "after" advice is used to insert the `Profiler.end` method at the end of every method to stop the timer. This profiling aspect is like a pluggable module that can be used or removed as needed. In this case, the profiling concern is separated into a single module, the aspect called `ProfileClasses`, which allows a developer to add or remove the profiling concern by modifying a single module, rather than manually modifying each method in every class.

3. Results and Discussion

There are several observations and lessons learned from our experience in creating profilers using various metaprogramming approaches. Each approach offered its own set of advantages and limitations. The primary distinction among the approaches relates to the power of adaptation versus the conciseness of expressing the crosscutting profiling concern.

3.1 Benefits and Limitations in Using OpenJava

The modularization boundary of an OpenJava program is very clear because the base program and the metaprogram are separate. However, a major problem with OpenJava is that it cannot automatically catch every return point for each method, and therefore the function of the profiler implemented by OpenJava is limited. As noted in Section 2.1.2, it can be observed that OpenJava cannot find the return statement nested in a statement or block. The only way to deal with this limitation is to parse the code within the OpenJava metaprogram, which is overkill in many cases. However, Javassist and AspectJ do not have such a problem. They can guarantee that the inserted statement is executed before the return of a method no matter where the return statement is located, which enables Javassist and OpenJava to fully realize the function of the profiler.

Another problem with OpenJava is that in order to use the profiler, the definition of each class in the source file must add the keyword "`instantiates`" and the name of metaclass into the header of each application class that needs to be adapted. For example, in our tracing example of Section 2.1, "`instantiates TracingClass`" is added to every class in the source file. If there are a large number of classes that need to be profiled, this would be a tedious and time consuming work that also requires invasive manual modification (albeit limited to the class signatures). By comparison, Javassist and AspectJ can define which class needs to be profiled without changing the base level source file.

An advantage of OpenJava is that it has more flexibility to change the source code. An OpenJava metaprogram can change any line of a Java application at a low-level of control. However, Javassist only supports changing the code in binary form and AspectJ is not able to change the source code at such a low-level of granularity.

3.2 The Comparative Expressiveness of AspectJ

AspectJ is more expressive as compared to OpenJava and Javassist. The code written in AspectJ is easier to read and comprehend because AspectJ adopts a language-based approach compared to the library-based approach of OpenJava and Javassist. There is also integrated tool support for AspectJ - Eclipse has an AspectJ plugin that assists in making the development of aspects more transparent to a programmer.

AspectJ and OpenJava are extensions of Java whereas Javassist is a set of APIs. All three techniques help in changing the struc-

ture and behavior of the base program in a separate module. Although AspectJ is easy to use, OpenJava provides fine-grained control on the program. In OpenJava, more power is given to the programmer to alter the semantics of the language. In AOP, power is captured in language constructs that provide safer use. For AOP-based techniques like AspectJ, it is important to clearly recognize and specify the join points [10]. This requirement is a challenge because certain code constructs like "for loops" are difficult to specify as join points.

3.3 Javassist as a Bytecode Transformation API

Javassist is a bytecode reengineering toolkit that does not require source code if the bytecode level APIs are used. This technique is extremely useful in a scenario in which third party software needs to be reengineered and the source code is unavailable. Javassist does not require any special compiler and can perform transformations at both compiler-time and load-time. Javassist is a popular metaprogramming approach that has been adopted in commercial middleware.

4. Conclusion

Software developers are continually asked to modify software to accommodate new change requests. For some types of requests (e.g., those that are crosscutting), manual invasive changes to a source code base provide many challenges. Metaprogramming can be used to help isolate change requests into a single module without the need to invasively change code across modularity boundaries.

There are several approaches and techniques to realize the benefits of metaprogramming and aspect-oriented programming. This paper explored three separate approaches toward modularizing the concerns of a profiler. The paper provides a summary discussion of profiler implementation using OpenJava, Javassist and AspectJ. A discussion of the lessons learned offer insight into the benefits and limitations of each approach.

The purpose of this paper was not an investigation into profiling, per se; there are several commercial profilers such as JProbe [11] that perform much more advanced functionality than the simple profiling demonstrated in this paper. The adoption of profiler construction as a common case study was due to the intrinsic crosscutting behavior of profiler implementation. Such a case study served as an excellent artifact for comparative evaluation.

Note to reviewer: Due to double-blind review, we did not disclose the URL of our website that contains all of the complete source code and experimental results from the investigation. If accepted, the URL for this website would be disclosed in a final revision.

References

[1] Kiczales, G., Lamping,J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin J. *Aspect-oriented programming*. European Conference on Object-Oriented Programming (ECOOP), LNCS 2072, Jyvaskyla, Finland, June 1997, pp. 220-242.

[2] Filman, R. E.; Tzilla E., Siobhàn C, and Mehmet A., *Aspect-Oriented Software Development*, Addison-Wesley, 2004.

[3] Kiczales, G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G., *An Overview of AspectJ*, European Conference on Object-Oriented Programming (ECOOP), LNCS 2072, Budapest, Hungary, June 2001, pp. 327-355.

[4] Tatsubori M., Chiba S., Killijian M., and Itano K., *OpenJava*: *A Class-Based Macro System for Java*, Reflection and Software Engineering, LNCS 1826, Denver, CO, November 1999, pp. 117-133.

[5] McManis, C., *Take an in-depth look at the Java Reclection API*, Java World, September 1997, http://www.javaworld.com/javaworld/jw-09-1997/jw-09-indepth.html

[6] Wu, Z., *Reflective Java and a Reflective-Component-Based Transaction Architecture*, OOPSLA'98 Workshop on Reflective Programming in C++ and Java, Vancouver, Canada, October 1998.

[7] Welch, I. and R. Stroud, *From Dalang to Kava :The Evolution of a Reflective Java Extension*, Reflection and Software Engineering, LNCS 1826, Denver, CO, November 1999, pp. 155-167.

[8] Kleinoder, J. and M. Golm, *MetaJava*: *An Efficient Run-Time Meta Architecture for Java*, International Workshop on Object Orientation in Operating Systems, Seattle, WA, October 1996, pp. 54-61.

[9] Chiba, S. *Load-time Structural Reflection in Java*, *E*uropean Conference on Object-Oriented Programming (ECOOP), *L*NCS 1850, Cannes, France, June 2000, pp. 313-336.

[10] Harbulot, B., Gurd, J., *Using AspectJ to Separate Concerns in Parallel Scientific Java Code*, International Conference on Aspect-Oriented Software Development, Lancaster, UK, March 2004, pp. 122-131.

[11] *JProbe*, http://www.quest.com/jprobe/