INVESTIGACIÓN & DESARROLLO, No. **10**, Vol. **1**: 75 – 90 (2010)
ISSN 2518-4431

ADVANCED AND RAPID DEVELOPMENT OF DYNAMIC ANALYSIS TOOLS FOR JAVA

DESARROLLO AVANZADO Y RÁPIDO DE LAS HERRAMIENTAS DE ANÁLISIS DINÁMICO PARA JAVA

Alex Villazón*, Walter Binder, Danilo Ansaloni** and Philippe Moret****
******Centro de Investigaciones de Nuevas Tecnologías Informáticas* – CINTI
Universidad Privada Boliviana, Bolivia
***Faculty of Informatics, University of Lugano, Switzerland*
avillazon@upb.edu
(Recibido el 10 de octubre 2010, aceptado para publicación el 30 de noviembre 2010)

ABSTRACT

Low-level bytecode instrumentation techniques are widely used in many software-engineering tools for the Java Virtual Machine (JVM), that perform some form of dynamic program analysis, such as profilers or debuggers. While program manipulation at the bytecode level is very flexible, because the possible bytecode transformations are not restricted, tool development based on this technique is tedious and error-prone. As a promising alternative, the specification of bytecode instrumentation at a higher level using aspect-oriented programming (AOP) can reduce tool development time and cost. Unfortunately, prevailing AOP frameworks lack some features that are essential for certain dynamic analyses. In this article, we focus on three common shortcomings in AOP frameworks with respect to the development of aspect-based tools - (1) the lack of mechanisms for passing data between woven advices in local variables, (2) the support for user-defined static analyses at weaving time, and (3) the absence of pointcuts at the level of individual basic blocks of code. We propose @J, an annotation-based AOP language and weaver that integrates support for these three features. The benefits of the proposed features are illustrated with concrete examples.

RESUMEN

Las técnicas de instrumentación de bajo nivel de código intermediario (bytecode) son utilizadas ampliamente en herramientas de ingeniería de software para la Máquina Virtual Java (Java Virtual Machine - JVM) para realizar formas de análisis dinámico de programas, como por ejemplo perfiladores (profilers) o depuradores (debuggers). Si bien la manipulación de código al nivel de bytecode es muy flexible, ya que las transformaciones no tienen restricciones, las herramientas basadas en esta técnica son tediosas de implementar y son propensas a la inserción de errores. Como una alternativa prometedora, la especificación de la instrumentación de bytecode a un nivel más elevado, utilizando Programación Orientada a Aspectos (Aspect-Oriented Programming – AOP), puede reducir el tiempo y el costo del desarrollo de herramientas. En este artículo, nos concentramos en tres deficiencias comunes en los sistemas AOP existentes con respecto al desarrollo de herramientas basadas en aspectos – (1) la falta de mecanismos para pasar datos entre consejos tejidos (woven advices) en variables locales, (2) el soporte para análisis estáticos definidos por el usuario durante la integración de los aspectos (weaving time), y (3) la ausencia de puntos de corte para inserción de aspectos (pointcuts) a nivel de bloques individuales básicos de código. Proponemos @J, un lenguaje y un tejedor (weaver) AOP basado en anotaciones que integra soporte para las tres características mencionadas anteriormente, cuyos beneficios son ilustrados con ejemplos concretos.

Keywords: Aspect-Oriented Programming, Aspect Weaving, Analysis at Weaving Time, Bytecode Instrumentation, Dynamic Program Analysis.
Palabras Clave: Programación Orientada a Aspecto, Análisis durante el proceso de Tejido de Aspectos, Instrumentación de Bytecode, Análisis Dinámico de Programas.

1. INTRODUCTION

Java applications are compiled into hardware-independent code, known as bytecode, which ensures portability and is interpreted by the Java Virtual Machine (JVM). This representation has enough symbolic information to reconstruct the structure of the compiled application. Instead of modifying the application source code, which is often not available, low-level instrumentation techniques had been developed to modify the bytecode at compile-time, load-time and runtime. These bytecode instrumentation techniques are widely used for building software-engineering tools that perform some dynamic program analysis, such as profilers [21], [20], [27], [11], [32], memory leak detectors [49], [33], data race detectors [15], [16], or testing tools that preserve the conditions that caused a crash [5].

Java supports bytecode instrumentation using native code agents through the Java Virtual Machine Tool Interface (JVMTI) [43], as well as portable bytecode instrumentation through the **java.lang.instrument** API. Several bytecode engineering libraries have been developed, such as BCEL [44], ASM [34], Javassist [17], or Soot [45], to mention some of them.

However, because of the low-level nature of bytecode (similar to assembly language) and of bytecode engineering libraries (frequently representing every bytecode instructions as an object), the implementation of new instrumentation tools can be difficult and error-prone, often requiring high development and testing effort. For example, to add some instructions to invoke some profiling code, it is necessary to insert the new block of bytecode instructions, update the all the instruction offsets and references, add new references to additional classes, etc. A frequent mistake is the incorrect update of exception-handler tables, which may result in instrumented code that passes many tests, but may later fail under particular conditions. It is not unusual, that due to such errors, instrumented applications even crash the JVM, making the development and debugging extremely difficult. As another drawback of low-level instrumentation techniques, the resulting software-engineering tools are often complex and difficult to maintain and to extend.

Aspect-oriented programming (AOP) [29] is a powerful approach enabling a clean modularization of cross-cutting concerns, such as error checking and handling, synchronization, context-sensitive behavior, monitoring and logging. Instead of scattering the related code throughout methods, classes and components, AOP allows the separation of such concerns into separated and independent entities, called *aspects*. These concerns are considered as cross-cutting, because the “cut across” multiples abstractions in a program. For example, to perform logging in a program (e.g., to record information every time that some operations are performed), conventional programmers insert instructions to invoke a logging component in different locations of the program. With AOP, the programmer can implement a “logging aspect” that is independent from the application code. It specifies *when*, *how* and *were* a logging operation must be triggered. An *aspect weaver* reads the aspect description and the application code, and generates appropriate code with the aspect integrated, performing the logging functionality. Aspects specify *pointcuts* to intercept certain points in the execution of programs (so-called *join points*), such as method calls, field accesses, etc. *Advices* (additional behaviors) are executed *before*, *after*, or *around* the intercepted join points. Advices have access to contextual information of the join points. Aspect weaving can be performed at source code or compiled code level.

In Java, there are several AOP frameworks such as AspectJ [28], *abc* [7], AspectWerkz [6], Steamloom [14], PROSE [38], JBossAOP [26]. AspectJ is the standard de-facto AOP language for Java, from which most of the others derive. In AspectJ, aspects are compiled into standard Java classes and advices as methods. During the weaving process the weaver inserts code in the woven application class at the matching join points, to invoke these advice methods.

Traditionally, AOP has been used to avoid needless repetition of code and for improving maintainability of applications. More recently, AOP has also been successfully applied to the development of software-engineering tools, such as profilers, debuggers, or testing tools [37, 8, 48], which in many cases can be specified as aspects in a concise manner. Hence, in a sense, AOP can be regarded as a versatile approach for specifying some program instrumentation at a high level, hiding low-level implementation details, such as bytecode manipulation, from the programmer. Although it is possible to implement some instrumentation-based software-engineering tools with current AOP frameworks, unfortunately some important features are missing, thus limiting the program instrumentation that can be expressed as an aspect. This is because prevailing AOP frameworks have not been designed for this purpose.

In our research, where we have tried applying AOP for recasting various instrumentation-based software-engineering tools as aspects, we found the following three essential missing features, which are not supported by existing AOP frameworks:

- *Data passing between advices that are woven into the same method using local variables.* In many instrumentation-based tools, local variables are allocated to pass data between different instrumentation sites in the code. While AspectJ’s around advice allows passing data generated by code inserted before a join point to code inserted after a join point, it is not possible to pass data in local variables between different join points, such as from a “before call” advice to an “after execution” advice. This missing feature is illustrated in Figure 1(a).
- *Execution of custom analysis code, which only depends on static information, at weaving time.* Many instrumentation-based tools perform some specific analysis to determine whether and how a particular join point shall be instrumented. For example, the listener latency profiler LiLa [27] analyses the class hierarchy to determine whether an invoked method is declared in a listener interface; only in that case the method invocation is profiled. Without user-defined analysis at weaving time, it is required to perform complex verifications and to navigate the class hierarchy at runtime. This results in high overhead, making listener latency profiling impracticable[1]. The lack of support for user-defined analyses at weaving time is illustrated in Figure 1 (b).

[1] Because the goal of listener latency profiling is to detect performance issues in the response time of event listeners, it is paramount that the overhead of the profiling code execution remains minimal.

- *Pointcuts to intercept the execution of basic blocks of code (BBCs) and context information on static BBC properties, such as the number of bytecodes in an intercepted BBC, or any user-defined statically computed bytecode metrics.* Many instrumentation-based software-engineering tools compute bytecode metrics, such as the dynamic metrics collector *J [21], the platform-independent profiler JP [11, 32], or the cross-profiler CProf for embedded Java [13]. As a means to express such tools as aspects, it is essential to aggregate bytecode metrics at the BBC level. The lack of support for BBCs is shown in Figure 1 (c).

In order to provide these missing features in an AOP framework, we propose @J (Aspect Tools in Java), an annotation-based aspect language and weaver, especially intended for easing the implementation of instrumentation-based software-engineering tools. Annotations in Java are a special form of syntactic metadata that can be added to the source code, which avoids modifying the programming language and the compiler. It is therefore a well-suited mechanism for extensions. @J supports many AspectJ constructs in AspectJ's annotation version, which we will call @AspectJ[2]. In this article, we focus on the new constructs offered by @J that are not available in @AspectJ.

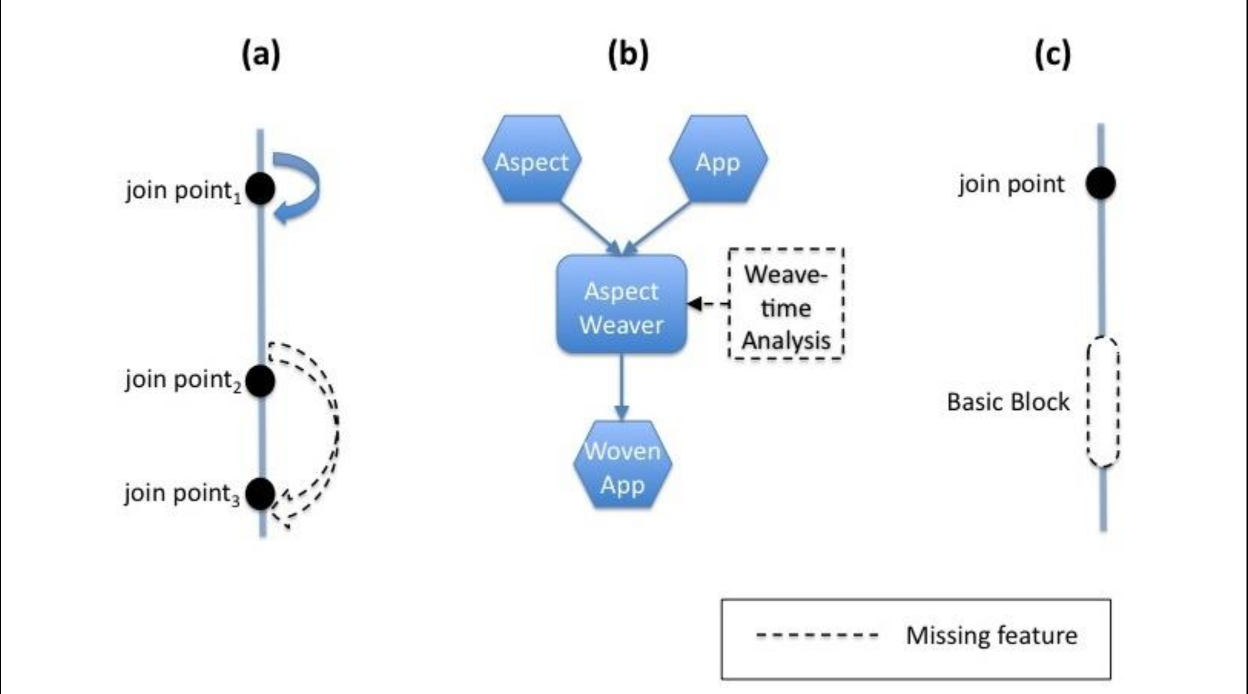

Figure 1 - Limitations of prevailing AOP frameworks.

In @J, an instrumentation is expressed as *code snippets* (i.e., small regions of re-usable code) which are woven at bytecode positions specified by the snippet programmer, including the aforementioned pointcuts at the BBC level. Note that in @J, we always use the term "snippet" instead of "advice" for the code to be executed at an intercepted join point, because this terminology is more intuitive for programming instrumentation-based software-engineering tools. By default, snippets are inlined in the woven code. @J supports invocation-local variables [46], allowing snippets that are woven into the same method body to pass data in local variables. @J snippets may access context information, such as static or dynamic join point instances, in the same way as in @AspectJ. In addition, static properties of BBCs are exposed through an extensible set of pseudo-variables.

In order to allow the expression of custom static analyses that are executed at weaving time, @J supports *executable snippets*. An executable snippet may only access static context information. By storing values in invocation-local variables, an executable snippet can pass the results of a static analysis to other inlined snippets. Apart from writing to invocation-local variables, executable snippets must not have any side effects. An executable snippet is woven by inserting a bytecode sequence that assigns the values to invocation-local variables that the snippet has generated upon execution at weaving time. As snippets can be composed, the code inserted in a woven method at a particular bytecode

[2] AspectJ is an extension of the Java programming language with AOP constructs, which requires a special compiler. @AspectJ in contrast can be compiled with standard Java compiler. The AOP constructs are expressed as annotations.

position may consist of an arbitrary sequence of inlined and executable snippets. Hence, custom static analyses can be embedded within inlined code snippets.

The rest of this article is structured as follows: Section 2 summarizes the design goals underlying @J. Section 3 discusses the distinguishing language features of @J. Section 4 gives some examples of @J programs, illustrating the use of @J's special features. Section 5 discusses related work, and Section 6 concludes this article. The work presented in this article extends and complements the description of @J in [4].

2. DESIGN GOALS FOR @J

In this section we summarize the design goals underlying @J

- **Expressiveness:** @J is designed to allow the expression of a wide range of instrumentation-based software-engineering tools. We have explored a large variety of case studies in the profiling and debugging domains in order to determine the necessary features. Examples include the dynamic metrics collector *J [21], profilers and memory leak detectors such as the NetBeans Profiler [33], the latency listener profiler LiLa [27], the testing tool ReCrash [5], the Eclipse plugin Senseo that collects various dynamic metrics and runtime type information [40], the resource management framework JRAF2 [12], [9], the platform-independent profiler JP [11], [32], and the cross-profiler CProf [13]. @J allows recasting the considered case studies as compact snippets that can be easily extended.
- **Efficiency:** @J shall enable the construction of efficient software-engineering tools that offer equivalent level of runtime performance as tools programmed with low-level bytecode engineering libraries.
- **Full method coverage:** For many instrumentation-based dynamic analysis tools, such as profilers or memory leak detectors, it is essential that the instrumentation covers all methods executing in the JVM (which have a bytecode representation), including methods in dynamically generated or loaded classes, as well as in the Java class library. Otherwise, the resulting profiles are incomplete and potentially useless. To ensure full method coverage, @J is based on the FERRARI framework[3] [10], which is also the basis of the MAJOR aspect weaver [47], [48]. FERRARI prevents the execution of inserted code during JVM bootstrapping. That is, during the bootstrapping phase, @J snippets are not executed. However, full method coverage is guaranteed for the whole execution of a program's main thread, and for all threads spawned by the program.
- **Java annotations:** Leveraging Java annotations, we do not introduce any new language constructs in Java. As a consequence, any standard Java compiler may be used to com- pile @J snippets. Note that in addition to @AspectJ, several other AOP frameworks, such as JBossAOP [26], AspectWerkz [6], SpringAOP [41], or Spoon [36], also support annotation-based aspect specifications, which confirms the soundness of the approach.

3. @J FEATURES

In this section, we firstly summarize the features of @AspectJ that are also supported in @J, and secondly explain the new features of @J that are complementary to @AspectJ.

3.1. Supported @AspectJ Features

@J supports @AspectJ pointcuts (e.g., "execution", "call", "get", "set", etc.), as well as *before* and *after* advices. Static and dynamic join points are supported in the same way as in @AspectJ. That is, context information about the well-defined points in the execution of the program is accessible to the aspect programmer (e.g., though reflective join point information provided by `JoinPoint` and `JoinPoint.StaticPart`).

@J does not support non-singleton aspect instances using `per*` clauses (e.g., per-object or per-control flow aspect association), because @J snippets are either inlined or executed at weaving time.

@AspectJ's *around* advice is not supported in @J. The @AspectJ weaver implements the around advice by inserting wrapper methods in woven classes [25], which can cause problems when weaving the Java class library. For instance, in Oracle's HotSpot JVMs there is a bug that limits the insertion of methods in `java.lang.Object`[4], which prevents the creation of additional (wrapper) methods. Moreover, wrapping certain methods in the Java class library breaks stack introspection in many recent JVMs, including Oracle's HotSpot JVMs and IBM's J9 JVM [32], [48]; usually, there is

[3] http://www.inf.usi.ch/projects/ferrari
[4] http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6583051

no public documentation indicating those methods in the class library that must not be wrapped. Hence, the use of around advices would compromise weaving with full method coverage in many common, state-of-the-art JVMs. Nonetheless, with the aid of invocation-local variables [46], it is possible to emulate a common use of around advices as a combination of before and after advices. We show the use of such combination in @J, to emulate the around advice in the examples in Section 4.

Static cross-cutting (inter-type declarations) [28] enables explicit structural modifications, such as changes of the class hierarchy or insertions of new fields and methods. In contrast to AspectJ without annotations, @AspectJ restricts the possibilities of static cross-cutting. For instance, in @AspectJ, it is not possible to insert fields in existing classes. @J does support some kind of structural modification, with the insertion of special fields, as a mean to achieve performance and not as a general-purpose static cross-cutting. This is because, regarding the development of software-engineering tools performing dynamic analyses, we found it essential to have efficient support for thread-local variables, which are often constantly accessed by such tools. To this end, we have used to insert thread-local variables as fields into **`java.lang.Thread`**, in order to avoid the overhead of accessing thread-local variables through the **`java.lang.ThreadLocal`** API (which involves a hashtable lookup, and therefore impacts performance). As @AspectJ does not support field insertion, @J provides dedicated support for efficient thread-local variables through **`@ThreadLocal`** annotations (see Section 3.6).

3.2. Snippets and their Composition

While an aspect in `@AspectJ` starts with an **`@Aspect`** annotation, an `@J` class is annotated with **`@J`**, which can take some extra annotation parameters.

Snippets are public static methods with void return type annotated with **`@BeforeSnippet`**, **`@AfterSnippet`**, **`@AfterReturningSnippet`**, or **`@AfterThrowingSnippet`**. These @J annotations correspond to **`@Before`**, **`@After`**, **`@AfterReturning`**, respectively **@**`AfterThrowing` advice methods in @AspectJ, but may take some additional annotation parameters. In @J, snippets may be woven only *before* or *after* a join point; in contrast to @AspectJ, @J does not support weaving *around* a join point. The @J snippet annotations support the optional boolean parameter execute for indicating whether a snippet is to be inlined (default: **`execute=false`**) or executed at weaving time (**`execute=true`**).

In contrast to advices in @AspectJ, snippets are always static in @J. Since snippets are inlined or executed at weaving time, it is not possible to change the snippets associated with a program at runtime. In contrast, the standard @AspectJ weaver inserts invocations of advice methods instead of inlining their bodies. The approach taken by @AspectJ has the benefit that the aspect association can be changed at runtime. However, for the purpose of @J, we consider static snippets appropriate, because snippet inlining is a prerequisite for passing data between snippets woven into the same method using local variables [46].

If multiple snippets match a join point, the @J programmer must specify the precedence of snippets. To this end, the @J snippet annotations support an optional integer parameter **`order`** (snippets with smaller order value come first). Weaving produces an error, if the order of multiple matching snippets is insufficiently specified.

3.3. Invocation-local Variables

@J supports the notion of invocation-local variables [46], in order to allow efficient data passing in local variables between snippets. The term “invocation-local” was chosen to imply that the scope of an invocation-local variable is one invocation of a woven method. Invocation-local variables are accessed through public static fields with **`@InvocationLocal`** annotations. Within snippets, invocation-local variables can be read and written as if they were static fields. For each invocation-local variable accessed in a woven method, a local variable is allocated and the bytecodes that access the corresponding static field are simply replaced with bytecodes for loading/storing from/in the local variable.

Each invocation-local variable is initialized in the beginning of a woven method with the value stored in the corresponding static field, which is assigned only during execution of the static initializer.[5] This implies that the @J class holding the snippets and the invocation-local variables is also loaded in the JVM, although the snippets are never invoked at runtime, and the static fields corresponding to invocation-local variables are assigned values only by the corresponding static initializers. As an optimization, the initialization of local variables corresponding to invocation-local variables in woven methods is skipped, if the weaver can statically determine that the first access is a write operation.

[5] The Java memory model [24], [31], [23] ensures that the value assigned by the static initializer is visible to all threads.

3.4. Snippet Execution at Weaving Time

In many instrumentation-based software-engineering tools, we found optimizations where some static analysis is performed at instrumentation time in order to decide whether and how to instrument a particular location in the bytecode. Standard AspectJ does not support the execution of custom analysis code at weaving time, making it impossible to recast such optimizations in aspects.

@J introduces *executable snippets*, which are executed at weaving time. Executable snippets produce weavable results by writing to invocation-local variables. In the woven method, a byte- code sequence is inserted that reproduces the values of the written invocation-local variables. Executable snippets may access only static context information, such as static join points, and must not have any side effects apart from writing to invocation-local variables of primitive type or of type **`java.lang.String`** (i.e., the written values are constants that can be stored in the constant pool or in the bytecode of a woven class). Executable snippets must not read any invocation-local variable.

Instead of inlining an executable snippet, the @J weaver creates an environment that enables snippet execution for matching join points at weaving time. To this end, the weaver generates a class holding the executable snippets (in a transformed version) and the invocation-local variables. The transformed snippets are instrumented so as to provide the set of written invocation-local variables upon completion. The resulting class is loaded at weaving time, and for each matching join point, the corresponding transformed snippet method is called, passing the needed static context information of the join point as arguments. Note that this may require allocating static join point instances at weaving time, if an executable snippet makes use of it. After snippet execution, the weaver inlines a bytecode sequence in the woven method that assigns the written invocation-local variables with the respective constants (which are added to the constant pool of the class holding the woven method)

A typical use of an executable snippet is to run some static analysis at weaving time, producing a boolean value in an invocation-local variable indicating whether the join point shall be instrumented. The executable snippet is composed with a normal (inlined) snippet, which is a conditional statement on the value of the boolean invocation-local variable. Evidently, in case the condition is false, the inlined snippet is dead code, which is likely to be eliminated by the just-in-time compiler of the JVM. @J does not perform any bytecode optimization, such as dead code elimination, since we assume that the snippet-based software-engineering tools will execute on standard, state-of-the-art JVMs, which already include sophisticated optimizations upon bytecode compilation. A detailed example involving an executable snippet is presented in Section 4.3.

Figure 2 depicts a comparison between the compilation and weaving process for @AspectJ and @J. In both cases, since aspects are implemented as conventional Java classes with annotations, they can be compiled with any standard Java compiler. In the case of @AspectJ, advices are compiled into methods. The AspectJ weaver uses the compiled aspect code together with the compiled application as input, to weave the aspect into the application code. The weaver finds the matching join points (shown as black circles ● in Figure 2 (a)), an insert bytecodes to invoke the corresponding advice methods, according to the pointcuts described in the aspect [25].

In the case of @J, the weaving process has additional steps to support executable snippets. The @J weaver also takes as input the compiled aspect class and the compiled application. Similar to advices, snippets are compiled into methods, but used differently. If snippets are marked as executable (see "Snippet code$_{exec}$ " in Figure 2(b)), the @J weaver generates an auxiliary class to hold the executable snippet, so as to enable its execution at weaving time. The @J weaver locates the matching join points, then executes the executable snippet and produces the code to store the result of the execution of the executable snippets into invocation-local variables, according to the pointcut description, as described before. In the case of non-executable snippets (i.e., by default), the weaver inlines the code of the snippets at the corresponding locations. Note that in @J all the necessary code is contained in the woven application, whereas, in the case of @AspectJ, the aspect class needs to be instantiated when the woven application is executed.

3.5. BBC Pointcuts

In @J, BBCs are join points where snippets can be woven. To this end, @J introduces the new pointcut designator **`bbc`**. For instance, the pointcut **`bbc(* java.util..*(..))`** matches all BBCs in methods in classes in packages whose qualified name starts with "**`java.util.`**". Snippets can be woven only *before* a BBC pointcut; that is, only the **`@BeforeSnippet`** annotation supports the **`bbc`** pointcut designator.

The definition of BBC is highly customizable. The BBC analysis algorithm, which builds the control-flow graph (CFG) of a method, can be provided by the user; it must implement predefined BBC and CFG interfaces. @J comes with default BBC and CFG implementations that are appropriate for a variety of case studies we have considered. The BBC analysis algorithm to be used in an @J class is specified with the optional @J annotation parameter `bbc`. For example, the annotation **`@J(bbc = "org.atj.DefaultBBCAnalysis")`** selects @J's default implementation.

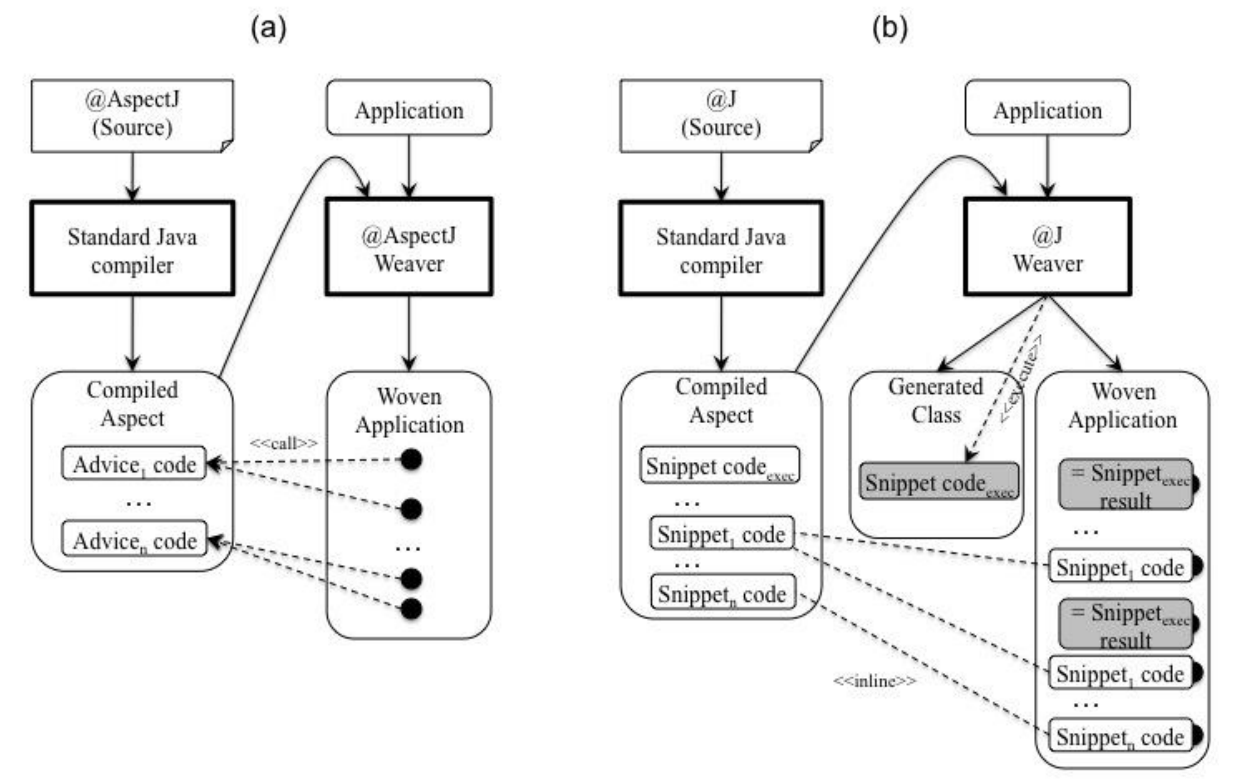

Figure 2 – Compilation and weaving processes for (a) @AspectJ and (b) @J.

Two kinds of properties can be associated with each BBC: (1) matching properties for pointcut matching, and (2) data properties that can be accessed within snippets through pseudo-variables. Both kinds of properties can be customized and extended by the programmer.

@J's default BBC implementation defines the following BBC matching properties: **`FirstInMethod()`** (first BBC in a method), **`FirstInHandler()`** (first BBC in an exception handler), **`FirstInLoop()`** ("first" BBC in a loop), **`EndsWithReturn()`** (BBC ends with a return bytecode), **`EndsWithThrow()`** (BBC ends with an athrow bytecode), **`EndsWithJump()`** (BBC ends with an unconditional jump bytecode), and **`EndWithBranch()`** (BBC ends with a conditional branch bytecode). In BBC pointcuts, a boolean expression (where the literals are BBC matching properties) can constrain the matching BBCs. For instance, the pointcut

```
bbc(* *.*(..)) && (FirstInMethod()|| FirstInHandler())
```

matches the first BBC in each method and in each exception handler. User-defined BBC matching properties are introduced with the annotation **`@MatchingProperty`**, which takes as argument a string with the property name. Typically, new BBC matching properties are defined in conjunction with the implementation of a new BBC analysis algorithm.

@J's default BBC implementation provides the following BBC data properties as pseudo-variables: **`org.atj.BBC.ID`** (integer uniquely identifying a BBC within a method), **`org.atj.BBC.OPS`** (integer giving the number of opcodes in a BBC), **`org.atj.BBC.BYTES`** (integer giving the number of bytes in a BBC),

org.atj.BBC.LC (integer indicating the corresponding line-of-code in the source, or zero if not available), and **org.atj.BBC.POS** (integer giving the position where a BBC begins in the bytecode).

The @J programmer may define custom pseudo-variables as static final fields with the **@PseudoVariable** annotation. These fields must be of primitive type or of type **java.lang.String**, such that access to a pseudo-variable in a snippet can be easily replaced with the corresponding constant value of a matching BBC join point. For example, in order to recast the cross-profiler CProf [13] as an @J class, it is necessary to compute a static metric for each BBC, according to a given target processor architecture; the statically computed metric represents an estimate of the CPU cycles used for executing the same BBC on the target processor. To this end, the programmer may define the new pseudo-variable **org.cprof.BBC.CPU** CYCLES of integer type.

In order to tell the @J weaver how to compute a custom BBC data property for a given BBC, each user-defined **@PseudoVariable** annotation must refer to a static method that takes a BBC as argument and returns a value of the pseudo-variable's type. The BBC interface provides access to the BBC bytecodes through the API of a bytecode engineering library, such as BCEL [44]. When programming in @J, defining new BBC data properties or implementing a custom BBC analysis algorithm are the only situations where the programmer is confronted with a low-level bytecode engineering API.

When pseudo-variables are accessed in an executable snippet then this complicates the snippet transformation for execution at weaving time. The accessed pseudo-variables are added as method arguments, such that the @J weaver can pass the appropriate constants for each matching BBC join point. Note that @AspectJ requires the aspect programmer to specify required context information, such as static or dynamic join points, as argument of advice methods. This is possible, because each kind of context information corresponds to a special type recognized by the weaver. In the case of BBC data properties, such an approach would not work, because the data properties are not distinguished by type. For instance, a snippet may access both **org.atj.BBC.ID** and **org.atj.BBC.OPS**, which are of integer type.

3.6. Efficient Thread-local Variables

Although Java provides dedicated support for thread-local variables through the **java.lang.ThreadLocal** API, directly inserting thread-local variables as instance fields into **java.lang.Thread** can improve performance, if the thread-local variables are accessed very frequently. In AspectJ without annotations, the inter-type declaration mechanism can be used for inserting fields into any class. As inter-type declarations are restricted in @AspectJ (because aspects are Java classes to be compiled with any standard Java compiler) such that field insertion is impossible, @J offers a special mechanism for inserting thread-local variables as instance fields into **java.lang.Thread.**

In @J, a static field annotated with **@ThreadLocal** acts as a thread-local variable. The static field corresponding to a thread-local variable must be initialized to the default value of the field's type. The weaver replaces access to the static field with bytecodes that get the current thread and access the inserted instance field. Weaving of **java.lang.Thread** is treated specially, as the extra instance fields corresponding to thread-local variables are inserted.

In comparison with standard thread-local variables of type **java.lang.ThreadLocal**, our approach offers two benefits: (1) direct access to thread-local variables without any hashtable lookup, and (2) support for thread-local variables of primitive types (without wrapping). Regarding limitations, our approach would fail if a JVM does not support the insertion of instance fields into **java.lang.Thread**. We have successfully tested our approach with various versions of Oracle's HotSpot JVMs and IBM's J9 JVMs on different platforms.

4. EXAMPLES

In this section, we discuss three simple example @J programs. The first example shows a simple tool to profile the execution time of every method that is invoked. It shows the usage of invocation-local variables. The second, recasts the JRAF2 resource management instrumentation [12, 9], illustrating an application of the BBC pointcut. The third example recasts the listener latency profiler LiLa [27] and shows the use of invocation-local variables and of an executable snippet for running a custom static analysis at weaving time. The two latter examples rely on snippet composition.

4.1. A Simple Execution Time Profiler

To illustrate the use of @J, let's start with a simple tool that profiles the elapsed wall clock for every invoked method. This kind of simple profiling tool is often useful to find performance bottlenecks. It also shows the use of invocation-local variables and basic @J constructs. The main idea is to capture the beginning and the end of every method execution; to store the start time at the beginning; to compute the elapsed time at the end of the method execution; and to profile this elapsed time together with details of the executed method (e.g., location in the source code, class, method name, and signature).

Figure 3 shows the **`ExecutionTimeProfiler`** tool that implements the profiling functionality in few lines of @J code. The **`ExecutionTimeProfiler`** is a conventional Java class with @J annotations (which start with the "**@**" character). The **`@Pointcut`** annotation is used to express the pointcut definition (provided as a string argument). The pointcut expressions correspond to conventional AspectJ's pointcut descriptions, which are supported by @J. In our example, the "**`execution(* *.*(..))`**" pointcut allows to capture all method executions in all classes. The **`@Pointcut`** annotation is used to mark the **`allExecs()`** empty method, which name can be used in the snippets, to refer to which pointcut they apply to. Thus, the **`takeStartTime()`** and **`takeEndTimeAndProfile(...)`** snippets, as specified by the **`@BeforeSnippet`** and **`@AfterSnippet`** annotations, will be inlined before and after every method body respectively.

The first snippet stores the start time into the **`start`** variable. By declaring **`start`** as **`@InvocationLocal`**, we ensure that the value stored in the **`@BeforeSnippet`** is accessible by the inlined code in the **`@AfterSnippet`** to compute the elapsed time. The context information of the captured join point (e.g., the signature and the source location that gave rise to the join point), is provided by AspectJ's **`JoinPoint.StaticPart`** and used to profile the execution time in the **`profileExecTime(...)`** method. Thus, it is possible to precisely identify the actual execution time for every method that is executed in the application. This example shows how invocation-local variables can be used in @J to emulate @AspectJ's *around* advice.

```
@J
public class ExecutionTimeProfiler {

  @InvocationLocal
  public static long start; // stores starting time

  // pointcut matching the execution of any method
  @Pointcut( "execution(* *.*(..))" )
  void allExecs() {}

  // stores the starting time
  @BeforeSnippet( pointcut = "allExecs"; )
  public static void takeStartTime() {
     start = System.nanoTime();
  }

  // profiles the elapse time after the execution of every methods
  // jpsp provides method details (package, class, name, signature)
  @AfterSnippet( pointcut = "allExecs";)
  public static void takeEndTimeAndProfile(JoinPoint.StaticPart jpsp) {
     profileExecTime(jpsp, System.nanoTime – start); // not shown here
  }
  ...
}
```

Figure 3- A simple Execution Time Profiler in @J

4.2. Recasting JRAF2

JRAF2 enables resource management in a platform-independent manner by accounting respectively limiting the number of bytecodes that a thread (or component) may execute. To this end, a thread-local bytecode counter is updated in each BBC according to the number of bytecodes in the BBC, and in some strategic locations (begin of methods, exception handlers, and loops), polling code is inserted to determine whether the bytecode counter has exceeded a given threshold. In this case, a user-defined resource management policy is invoked by the exceeding thread.

Figure 4 shows an @J class recasting JRAF2. The **`bytecodeCounter`** field is declared as **`@ThreadLocal`** and therefore added as instance field to **`java.lang.Thread`**. In the beginning of each BBC, the **`updateCounter()`**

snippet increments **`bytecodeCounter`** by the length of the BBC, provided by the pseudo-variable **`org.atj.BBC.OPS`**. The **`polling()`** snippet, which matches the first BBC in each method, exception handler, or loop, is inlined after the **`updateCounter()`** snippet, according to the specified order. The **`polling()`** snippet checks whether the **`bytecodeCounter`** value exceeds a given threshold and invokes a user-defined resource management policy in that case.

Compared to the original implementation of JRAF2, which has about 10 000 lines of code (dealing also with low-level issues of instrumenting the Java class library), the @J version is very compact, as shown in Figure 4.

```
@J( bbc = "org.atj.DefaultBBCAnalysis"; )
public class JRAF2 {
  // approximate number of bytecodes to be executed between subsequent
  // invocations of the user-defined resource management policy by the same thread
  public static final int THRESHOLD = ...;

  @ThreadLocal
  public static int bytecodeCounter;

  // pointcut matching all basic blocks in all methods
  @Pointcut( "bbc(* *.*(..))" )
  void allBBCs() {}

  @BeforeSnippet( pointcut = "allBBCs"; order = 1; )
  public static void updateCounter() {
    bytecodeCounter += org.atj.BBC.OPS;
  }

  @BeforeSnippet( pointcut = "allBBCs && ( FirstInMethod()|| FirstInHandler()|| FirstInLoop()
)";
                  order = 2; )
  public static void polling() {
    if (bytecodeCounter >= THRESHOLD) {
      runResourceManagementPolicy(Thread.currentThread(), bytecodeCounter); // not shown here
      bytecodeCounter = 0;
    }
  }
...
}
```

Figura 4 - JRAF2 instrumentation for resource management expressed in @J

4.3. Recasting LiLa

Listener latency profiling [27] helps developers locate slow operations in interactive applications, where the perceived performance is directly related to the response time of event listeners. LiLa[6] is an implementation of listener latency profiling based on ASM [34], a low-level bytecode engineering library.

The response time for handling an event relates to the execution time of an invoked method on an instance of a class implementing the **`java.util.EventListener`** interface. In order to reduce profiling overhead, LiLa does not instrument all methods in each subtype of **`java.util.EventListener`**, but restricts the instrumentation to those methods that are declared in an interface. Hence, LiLa analyzes the class hierarchy to determine which methods to instrument. This optimization reduces profiling overhead at runtime, because less methods are instrumented.

Even though it is possible to recast the basic profiling functionality of LiLa as an aspect in AspectJ, for example, using the around advice to measure response time by surrounding the execution of every event-related method, the optimization that reduces the number of instrumented methods cannot be performed at weaving time.

In Figure 5, we show how the static analysis at weaving time is implemented in @J with the executable snippet **`analyzeNeedsProfiling(...)`**. The result of the snippet execution at weaving time is stored in the invocation-local variable **`needsProf`**. The **`takeStartTime()`** snippet, which is inlined after the bytecodes that result from

[6] **`http://www.inf.usi.ch/phd/jovic/MilanJovic/Lila/Welcome.html`**

executing **`analyzeNeedsProfiling(...)`**, records the starting time only if the static analysis determined that profiling was needed. The invocation-local variable start stores the starting time for later use in the same woven method. The **`takeEndTimeAndProfile(...)`** snippet intercepts (both normal and abnormal) method completion. The listener object is made accessible within the body of the snippet through the expression "**`this(listener)`**" in the pointcut declaration. Whenever the execution time exceeds the given threshold, the method **`profileEvent(...)`** (not shown in the figure) logs an identifier of the intercepted method (conveyed by the static join point), the target object, and the execution time. This information helps developers locate the causes of potential performance problems due to slow event handling.

```
@J
public class LiLa {
  // listeners executing less than 100 ms (100,000,000 ns) are not logged
  public static final long THRESHOLD_NS = 100L * 1000L * 1000L;

  @InvocationLocal
  public static long start; // stores starting time of listener execution

  @InvocationLocal
  public static boolean needsProf; // stores result of static analysis

  // pointcut matching the execution of any method in any subtype of the EventListener interface
  @Pointcut( "execution(* java.util.EventListener+.*(..))" )
  void listenerExec() {}

  // static analysis at weaving time (result stored in invocation-local variable);
  // jpsp provides method details (package, class, name, signature)
  @BeforeSnippet( pointcut = "listenerExec"; execute = true; order = 1; )
  public static void analyzeNeedsProfiling(JoinPoint.StaticPart jpsp) {
     needsProf = isInterfaceMethod(jpsp); // not shown here
  }

  // store starting time upon listener entry, if the static analysis
  // considers profiling necessary
  @BeforeSnippet( pointcut = "listenerExec"; order = 2; )
  public static void takeStartTime() { if (needsProf) start = System.nanoTime(); }

  // profile listener execution upon completion, if the static analysis
  // considers profiling necessary and the execution time exceeds the threshold
  @AfterSnippet( pointcut = "listenerExec && this(listener)"; )
  public static void takeEndTimeAndProfile( JoinPoint.StaticPart jpsp,
                                           java.util.EventListener listener) {
  if (needsProf) {
     long exectime = System.nanoTime() - start;
     if (exectime >= THRESHOLD_NS)
       profileEvent(jpsp, listener, exectime); // not shown here
  }
```

Figure 5 - Listener latency profiler LiLa expressed in @J

The example in Figure 6 illustrates the weaving of an **`EventListener`** implementation. For the sake of easy readability, we show the transformations conceptually at the Java level, whereas the @J weaver operates at the bytecode level. The method **`actionPerformed(ActionEvent)`** is declared in the implemented interface and needs to be profiled, whereas the method **`notDeclaredInInterface()`** does not require profiling. Figure 6(b) shows the result of weaving. The interesting part is how the result of the static analysis is stored in the invocation-local variable **`needsProf`**. The woven code is quite long, since there is a significant amount of dead code. A state-of-the-art compiler will detect and eliminate the dead code, yielding the optimized code shown in Figure 6(c).

Similar to the previous example, we can see that the implementation of an advanced dynamic analysis tool can be done in few lines of code with @J (compared to the several thousand lines of code of the original LiLa implementation using low-level bytecode instrumentation techniques). Thanks to the proposed execution of custom analysis code at weaving time (through executable snippets), we are able to prevent costly analysis at runtime, which may impact performance and therefore make the tool impractical. This confirms the soundness and applicability of our approach.

(a) Before weaving:

```
class ExampleListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
     doSomething();
   }

   public void notDeclaredInInterface() {
     doSomethingElse();
   }
  ...
}
```

(b) Woven code:

```
class ExampleListener implements ActionListener {
   private static final JoinPoint.StaticPart
   jpsp1 = ..., // actionPerformed
   jpsp2 = ...; //  notDeclaredInInterface

  public void actionPerformed(ActionEvent e) {
    long start = 0L;
    boolean needsProf = true;
    if (needsProf) start = System.nanoTime();
    try {
      doSomething();
    } finally {
      if (needsProf) {
        long exectime = System.nanoTime() - start;
        if (exectime >= LiLa.THRESHOLD_NS) LiLa.profileEvent(jpsp1, this, exectime);
      }
    }
  }

  public void notDeclaredInInterface() {
    long start = 0L;
    boolean needsProf = false;
    if (needsProf) start = System.nanoTime();
    try {
      doSomethingElse();
    } finally {
      if (needsProf) {
        long exectime = System.nanoTime() - start;
        if (exectime >= LiLa.THRESHOLD_NS) LiLa.profileEvent(jpsp2, this, exectime);
      }
    }
  }
  ...
}
```

(c) Optimized code (e.g., by a just-in-time compiler that eliminates dead code):

```
class ExampleListener implements ActionListener {

  private static final JoinPoint.StaticPart
    jpsp1 = ..., //  actionPerformed
    jpsp2 = ...; // notDeclaredInInterface

  public void actionPerformed(ActionEvent e) {
    long start = System.nanoTime();
    try {
      doSomething();
    } finally {
      long exectime = System.nanoTime() - start;
      if (exectime >= LiLa.THRESHOLD_NS) LiLa.profileEvent(jpsp1, this, exectime);
    }
  }

  public void notDeclaredInInterface() {
    doSomethingElse();
```

```
   }
   ...
}
```

Figure 6 - Weaving and optimization of an example **`EventListener`** implementation

5. RELATED WORK

The AspectBench Compiler (abc) [7] eases the extension of AspectJ with new pointcuts [1, 15, 19]. Similar to @J, abc uses inlining of advice code, but for optimizing advice execution [50]. Even though the new pointcuts of @J could be implemented as an extension using abc, we opted for an annotation-based snippet development style in order to rapidly prototype @J features and therefore focus on the weaving part, rather than on the aspect language front-end. In addition, adapting the @AspectJ weaver to use FERRARI [10] for full method coverage turned out to cause less development effort than modifying abc.

Nu [22] enables extensions using an intermediate language model and explicit join points [39]. Nu adopts a fine-grained join point model. Similar to @J, it allows expressing aspect-oriented constructs in a flexible manner. While Nu is based on a customized JVM, @J is compatible with standard JVMs and uses standard Java compilers.

Steamloom [14] provides AOP support at the JVM level, which results in efficient runtime weaving. Steamloom enables the dynamic modification and reinstallation of method bytecodes and provides dedicated support for managing aspects. Steamloom uses its own aspect language and provides a parser to support AspectJ-like pointcuts. Steamloom is based on the Jikes RVM [2] and supports thread-locally deployed aspects. In order to support thread safety, Steamloom uses code snippets that are inserted before every call to advices, so as to verify whether the advice invocation for the current thread should be active or not. Similar to Steamloom, PROSE [38] also provides aspect support within the JVM, which may ease the implementation of low-level pointcuts thanks to the direct access to JVM internals. PROSE combines bytecode instrumentation and aspect support at the just-in-time compiler level with an extension of the Jikes RVM. Unfortunately, these approaches require of a customized JVM, thus limiting extensibility and portability.

Prevailing AspectJ weavers do not support the execution of custom analysis code at weaving time, which typically only depends on static information. SCoPE [3] is an AspectJ extension that partially solves this problem by allowing analysis-based conditional pointcuts. Similarly, the approach described in [30] enables customized pointcuts that are partially evaluated at weaving time. @J supports custom analysis through snippets that are executed during weaving.

In [18], AOP is explored as an approach to enable runtime monitoring. Since AspectJ lacks low-level pointcuts to capture enough details to cover all monitoring needs (e.g., weaving of statements, BBCs, loops, or local variable accesses), the authors present two new pointcuts to intercept BBCs and loops. Unfortunately, no concrete use case is described. The extension is implemented with abc and supports only minimal context information. In contrast, @J provides customizable BBC pointcuts and access to detailed BBC context information.
In [1], the notion of *region pointcut* is introduced as an extension of AspectJ, and implemented in abc [7]. Similar to the notion of a *block* in @J's BBC pointcuts, the region pointcut groups a range of code (e.g., for synchronization), for which conventional AspectJ join "points" are not well suited. Because a region pointcut potentially refers to several compound but spread join points, the around advice poses a problem. The proposed solution is based on an external object, shared between the join points inside the region, which holds the values to be passed between them. This is similar to the notion of invocation-local variables, but with a different scope.

Maxine [42] is a meta-circular research VM implemented in Java. Maxine uses a layered compiler with different intermediate representations. Instead of writing the code in a particular intermediate representation to add a runtime feature, Maxine allows developers to write snippets directly in Java, which are compiled into the corresponding intermediate representation. This approach decouples runtime features from compiler work. The Ovm [35, 4] virtual machine follows a similar approach, where a high-level intermediate representation eases the customization for building language runtime systems, so as to define new operations and to modify the semantics of existing ones.

In prior work [47, 48], we extended the AspectJ weaver with new features by transforming bytecode that has been previously woven with the original, unmodified AspectJ weaver. This approach has the benefit that new versions of the AspectJ weaver can be easily integrated. However, snippet weaving in @J differs significantly from aspect weaving in @AspectJ, such that we found it easier to directly modify the @AspectJ weaver. Note that for the weaving of BBC pointcuts, post-transformations after aspect weaving would not work, because aspect weaving changes the bytecode (which may result in spurious BBCs and incorrect BBC data properties). Similarly, pre-transformations before aspect weaving would not work either, because the inserted code may constitute spurious join points upon aspect weaving.

6. CONCLUSION

The implementation of tools that perform some kind of dynamic analysis, such as profiling, is largely based on low-level bytecode instrumentation techniques. Because the implementation of such tools at the bytecode level is tedious and error-prone, specifying dynamic analysis tools with high-level AOP is a promising approach. It reduces tool development costs, improves maintainability, and simplifies the extension of the tools.

Unfortunately, many prevailing AOP frameworks, such as AspectJ, lack certain features that are important for developing efficient dynamic analysis tools for certain purposes. We identified three missing features in AspectJ that we consider essential for tool development: efficient data passing between woven advices in local variables, the execution of custom static analyses at weaving time, and pointcuts at the level of individual BBCs.

In this article, we propose the annotation-based AOP framework @J, which is based on @AspectJ and incorporates support for these three features. We presented in detail the compilation and weaving process of @J. As examples, we presented a simple profiler tool that collects execution times, and we recast two existing tools based on low-level bytecode manipulation as @J classes, illustrating the use of @J's distinguishing features. The resulting tools are compactly implemented within a few lines of code.

Regarding limitations, @J suffers from the same problem as any other framework relying on Java bytecode instrumentation. The JVM imposes strict limits on certain parts of a class file (e.g., the method size is limited); these limits may be exceeded by the code inserted upon aspect weaving. Our approach aggravates this issue by inlining snippets, usually increasing the code bloat. Nonetheless, we have not yet encountered any problem due to code growth in practice.

7. ACKNOWLEDGEMENTS

Part of this work has been supported by the Swiss National Science Foundation.

8. REFERENCES

[1] S. Akai, et al. *Region pointcut for AspectJ*. In ACP4IS '09: Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software, pages 43– 48, New York, NY, USA, 2009. ACM.
[2] B. Alpern et al. "The Jalapeño virtual machine." *IBM Systems Journal*, vol. 39, no. 1, pp. 211–238, 2000.
[3] T. Aotani and H. Masuhara. "SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts." In AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development, pages 161–172, New York, NY, USA, 2007. ACM.
[4] W. Binder et al. "@J - Towards Rapid Development of Dynamic Analysis Tools for the Java Virtual Machine." In VMIL '09: Proceedings of the 3th Workshop on Virtual Machines and Intermediate Languages, pages 1–9, New York, NY, USA, 2009. ACM.
[5] S. Artzi et al. "ReCrash: Making Software Failures Reproducible by Preserving Object States." In J. Vitek, editor, ECOOP '08: Proceedings of the 22th European Conference on Object-Oriented Programming, volume 5142 of Lecture Notes in Computer Science, pp. 542–565, Paphos, Cyprus, 2008. Springer- Verlag.
[6] Aspectwerkz. "AspectWerkz - Plain Java AOP." Internet: http://aspectwerkz.codehaus.org/.
[7] P. Avgustinov et al. "abc: An extensible AspectJ compiler." In AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development, pp. 87–98, New York, NY, USA, 2005. ACM Press.
[8] L. D. Benavides et al. "Debugging and testing middleware with aspect-based control-flow and causal patterns." In Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware, pages 183–202, New York, NY, USA, 2008. Springer-Verlag New York, Inc.
[9] W. Binder and J. Hulaas. "A portable CPU-management framework for Java." *IEEE Internet Computing*, vol. 8, no. 5, pp. 74–83, Sep./Oct. 2004.
[10] W. Binder et al. "Advanced Java Bytecode Instrumentation." In PPPJ'07: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, pp. 135–144, New York, NY, USA, 2007. ACM Press.

[11] W. Binder et al. *Platform-independent profiling in a virtual execution environmen*t. Software: Practice and Experience, vol. 39, no. 1, pp. 47–79, 2009.

[12] W. Binder et al. *Portable resource control in Java*. ACM SIGPLAN Notices, vol. 36, no. 11, pp. 139–155, Nov. 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[13] W. Binder, et al. "Cache-aware cross-profiling for Java processors." Presented at International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES-2008), pp. 127–136, Atlanta, Georgia, USA, Oct. 2008. ACM.

[14] C. Bockisch et al. "Adapting virtual machine techniques for seamless aspect support." In OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp. 109–124, New York, NY, USA, 2006. ACM.

[15] E. Bodden and K. Havelund. "Racer: Effective Race Detection Using AspectJ." In International Symposium on Software Testing and Analysis (ISSTA), Seattle, WA, July 20-24 2008, pp. 155–165, New York, NY, USA, 07 2008. ACM.

[16] F. Chen et al. "jPredictor: A Predictive Runtime Analysis Tool for Java." In ICSE '08: Proceedings of the 30th international conference on Software engineering, pp. 221– 230, New York, NY, USA, 2008. ACM.

[17] S. Chiba. "Load-time structural reflection in Java." In Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000), volume 1850 of Lecture Notes in Computer Science, pages 313–336. Springer Verlag, Cannes, France, June 2000.

[18] J. Cook and A. Nusayr. "Using AOP for Detailed Runtime Monitoring Instrumentation." Presented at WODA 2008: the sixth international workshop on dynamic analysis, New York, NY, USA, jul 2009. ACM.

[19] B. De Fraine et al. "StrongAspectJ: flexible and safe pointcut/advice bindings." In AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development, pp. 60–71, New York, NY, USA, 2008. ACM.

[20] M. Dmitriev. "Profiling Java applications using code hotswapping and dynamic call graph revelation." In WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance, pp. 139–150. ACM Press, 2004.

[21] B. Dufour et al. "*J: A tool for dynamic analysis of Java programs." In OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 306– 307, New York, NY, USA, 2003. ACM Press.

[22] R. Dyer and H. Rajan. "Nu: A dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation." In AOSD '08: Proceedings of the 7th International Conference on Aspect-oriented Software Development, pp. 191–202, New York, NY, USA, 2008. ACM.

[23] B. Goetz et al. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[24] J. Gosling et al. *The Java Language Specification, Third Edition.* The Java Series. Addison-Wesley, 2005.

[25] E. Hilsdale and J. Hugunin. "Advice weaving in AspectJ." In AOSD '04: Proceedings of the 3rd International Conference on Aspect- Oriented Software Development, pp. 26–35, New York, NY, USA, 2004. ACM.

[26] JBoss. "Open source middleware software." Internet: http: //labs.jboss.com/jbossaop/.

[27] M. Jovic and M. Hauswirth. "Measuring the performance of interactive applications with listener latency profiling." In PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java, pp. 137–146, New York, NY, USA, 2008. ACM.

[28] G. Kiczales et al. "An overview of AspectJ." In J. L. Knudsen, editor, Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP-2001), volume 2072 of Lecture Notes in Computer Science, pp. 327–353, 2001.

[29] G. Kiczales et al. "Aspect-oriented programming." In M. Akşit and S.Matsuoka, editors ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming, volume 1241 of Lecture Notes in Computer Science, pages 220–242, Jyväkylä, Finnland, June 1997. Springer-Verlag.

[30] K. Klose et al. *Partial evaluation of pointcuts*. PADL, pp. 320–334, 2007.

[31] J. Manson et al. "The Java Memory Model." In POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 378–391, New York, NY, USA, 2005. ACM.

[32] P. Moret et al. "CCCP: Complete calling context profiling in virtual execution environments." In PEPM '09: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, pp. 151–160, Savannah, GA, USA, 2009. ACM.

[33] NetBeans. "The NetBeans Profiler Project." Internet: http: //profiler.netbeans.org/.

[34] OW2 Consortium. "ASM – A Java bytecode engineering library." Internet: http://asm.ow2.org/.

[35] K. Palacz et al. "Engineering a customizable intermediate representation." In IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators, pp. 67–76, New York, NY, USA, 2003. ACM.

[36] R. Pawlak. "Spoon: Compile-time annotation processing for middleware." *IEEE Distributed Systems Online*, vol. 7, no. 11, pp. 1, 2006.

[37] D. J. Pearce et al. “Profiling with AspectJ.” *Software: Practice and Experience*, vol. 37, no. 7, pp. 747–777, June 2007.

[38] A. Popovici et al. “Just-in-time aspects: efficient dynamic weaving for Java.” In AOSD ’03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, pp. 100–109, New York, NY, USA, 2003. ACM Press.

[39] H. Rajan. “A Case for Explicit Join Point Models for Aspect-Oriented Intermediate Languages.” In VMIL ’07: Proceedings of the 1st Workshop on Virtual Machines and Intermediate Languages for Emerging Modularization Mechanisms, pp. 4, New York, NY, USA, 2007. ACM.

[40] D. Röthlisberger et al. “Augmenting static source views in IDEs with dynamic metrics.” In ICSM ’09: Proceedings of the 25th IEEE International Conference on Software Maintenance, pp. 253–262, Edmonton, Alberta, Canada, 2009. IEEE Computer Society.

[41] Spring Framework. “Open source application framework.” Internet: http://www.springsource.org/.

[42] Sun Microsystems, Inc. “The Maxine Virtual Machine.” Internet: http://research.sun.com/projects/maxine/.

[43] Sun Microsystems, Inc. “JVM Tool Interface (JVMTI) version 1.1.” Internet: http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html, 2006.

[44] The Apache Jakarta Project. “The Byte Code Engineering Library (BCEL).” Internet: http://jakarta.apache.org/bcel/.

[45] R. Valleée-Rai et al. “Optimizing Java bytecode using the Soot framework: Is it feasible?” In Compiler Construction, 9th International Conference (CC 2000), pp. 18–34, 2000.

[46] A. Villazón et al. “Advanced Runtime Adaptation for Java.” In GPCE ’09: Proceedings of the Eighth International Conference on Generative Programming and Component Engineering, pp. 85–94. ACM, Oct. 2009.

[47] A. Villazón et al. “Aspect Weaving in Standard Java Class Libraries.” In PPPJ ’08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, pp. 159–167, New York, NY, USA, Sept. 2008. ACM.

[48] A. Villazón et al. “Flexible Calling Context Reification for Aspect-Oriented Programming.” In AOSD ’09: Proceedings of the 8th International Conference on Aspect-oriented Software Development, pp. 63–74, Charlottesville, Virginia, USA, Mar. 2009. ACM.

[49] G. Xu and A. Rountev. “Precise memory leak detection for Java software using container profiling.” In ICSE ’08: Proceedings of the 30th international conference on Software engineering, pp. 151–160, New York, NY, USA, 2008. ACM.

[50] P. Avgustinov et al. “Optimising AspectJ.” In PLDI ’05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 117–128, New York, NY, USA, 2005. ACM.