Software Manipulation with Annotations in Java

Vincenzo Gervasi Giacomo A. Galilei
Dipartimento di Informatica
Università di Pisa

Abstract

Annotations are a recent feature introduced in languages such as Java, C#, and other languages of the .NET family, which allow programmers to attach arbitrary, structured and typed metadata to their code. These languages run on top of so-called *virtual execution environments*, e.g. the JVM for Java, and the CLR for .NET languages, which allow for the run-time generation of executable code. In this paper we explore how annotations and the dynamic code generation capability can be used together to provide programmers with high-level methods for dynamic generation and modification of an application's code — at run-time. The paper introduces the `@Java` language, which is an extension to Java allowing annotation of arbitrary statements, and the JDAsm library, which is an infrastructure for bytecode manipulation which uses `@Java` annotations to pinpoint the locations and code fragments that are being manipulated. Together, they allow type-safe and fully symbolic runtime code modification and generation without any need to explicitly address bytecode instructions.

1 Introduction

The concept of *metadata*, which is data describing other data, is one of the mainstay in computer science, and has been used in a large variety of contexts, from defining database schema, to structuring digital annotations of medieval manuscripts. In this paper, we are mostly interested in *program* metadata, i.e. data describing programs. The concept of program metadata arises naturally in those languages where programs are data, e.g. LISP [16]. In these languages, the normal ways to describe relationships about different pieces of data can be used equally well to annotate programs with metadata. However, program metadata are common in more traditional languages as well, although mostly in a limited way.

The various incarnation of the concept of program metadata can be characterized by five features:

- **content:** what kind of information is carried by a metadata element
- **author:** who (person or tool) assigns a value to a metadata element
- **lifetime:** when a metadata element is attached to a program element, and when (if ever) it is discarded
- **location:** where is the metadata stored (e.g., together with the code, or in a separate location)

Metadata	Content	Author	Lifetime	Location	Target
comments	free text	programmer	source	source file	any lexical position as permitted by the language grammar
typing	types & signatures	compiler	source	source file	variables, functions, objects
compilation directive (e.g.#pragma)	instructions to the compiler	programmer	compile time	source	module
debugging symbols	symbol name, address, size, attributes	compiler	object	object file	module
identification tags	config tags, version numbers	compiler	object	object file	module or executable
API docs (e.g. JavaDoc)	API specifications	programmer	source, deploy to developers	source file (as comments)	functions, methods
Interface definitions	signatures	programmer	deploy to developers	IDL file (CORBA), WSDL file (web services)	functions (CORBA), methods (web services)
Versioning info (e.g., CVS)	release tags	revision control system	configuration release cycle	versioned source file	any lexical position
intellectual rights management (license info)	legal terms of use	programmer, lawyer	source (legal validity extends to executable)	source file (as comments)	module (typical), code fragment
development cycle control	links to design, rationale, tests, approvals, reviews, etc.	program manager	source to deploy	source file (as comments) or external management database	module, unit

Table 1: Characterization of some historical forms of metadata.

- **target:** to which program elements can a metadata element be attached

Historically, program metadata has been used, in a *ad hoc* fashion, to convey specific type of information across various tools, systems or activities in the development cycle, or across long time spans to different persons working on a system. For example, traditional forms of *comments* can be interpreted as free-form metadata, attached to a specific lexical position in the source code when the code is written by the programmer, and discarded upon compilation. Table 1 lists some forms of metadata which are commonly used in programming and system development practice.

The real weakness of these historical forms is the fact that each metadata type is defined in a different way, is set and processed by specific tools, and in most cases has no associated notion of *validity* of the content (e.g., there is no way to guarantee that a comment specifying some legal terms will be consistent with a predefined policy).

In recent times, the notion of program metadata has gained full citizenship both in the design of languages (e.g., C# [8], Java [14]) and in the corresponding execution environments (.NET CLR [9], JVM [15]). These new forms sport

important differences w.r.t. the historical forms we discussed above:

- they are **general purpose**, i.e. the schema for their content can be defined by the programmer, and the mechanism to set and retrieve the content is not specific to a particular schema;
- they can be applied to **generic program elements**, with the programmer being able to declare specific restrictions about which class of elements can be designated as targets
- they have **customizable lifetime and location**, encompassing all the range from source-only metadata (as comments) to run-time metadata (as typing information with reflection).

Another interesting development linked to the mainstream adoption of virtual execution environments is the comeback of modifiable code. With the exception of quasi-quotation mechanisms [7] in certain interpreted languages like LISP [16] or MetaML [17], the possibility of modifying the running code of an application has been ruled out in language design and by operating systems (usually with the assistance of hardware devices, e.g. by using an MMU to forbid writing in memory pages containing executable code) since the seventies, on the ground of security concerns.

However, the ability to synthesize, configure, customize or adapt the running code of an application at run-time, possibly without even requiring a shutdown of the application itself, is invaluable in many circumstances, as we will see in Section 5.

While it is certainly true that allowing the uncontrolled modification of executable code is unacceptable in terms of security, very prone to introducing bugs and potentially disastrous side effects, and can easily be abused or bring an entire application to an abrupt termination, the type safety of .NET IL [9] and of JVM bytecode [15] has allowed a safer approach to the issue. In fact, the standard library in .NET explicitly include means to generate IL code on the fly, and the class loading mechanism in the JVM provides similar (albeit less programmer-friendly) features to the same effect.

Both these approaches require however that the programmer synthesizes IL or bytecode fragments "by hand", listing instruction after instruction the contents of the fragment. In short, the programmer is required to be proficient in both a high-level language (e.g., C# or Java) to write the main bulk of the application in, and in IL or bytecode, in order to write high-level code which will emit at runtime the sequence of instructions appropriate to accomplish the task at hand. Given that programmers who can efficiently write assembly code are increasingly difficult to find (as a consequence of the demise of machine-code programming), this requirement is too stringent for most scenarios.

One solution which has been proposed (and implemented) has been to provide programmatic access to the compiler for the high-level language, so that applications can generate the source code for a high-level class, then ask the compiler to compile it, obtain a reference to the compiled class, and finally load it and invoke some of its methods (see, among others, [4]). However, this method suffers from a number of inconveniences, including a huge performance hit (both in time and space, as the whole compiler for the high-level language needs to be loaded and executed even for a small fragment), the difficulty of

```
/* a. an Annotation type to record the link between
   requirements and code */

public @interface Requirement {
   String id();
   String complianceStatement();
   String certifiedBy() default "John Doe, program manager";
   String date();
};


/* b. and its application to a method */
...
@Requirement(id="M1541", certifiedBy="Paul", date="12/5/2008")
public void applyStyle(TSpan span, Style s)
{
   ...
}
```

Figure 1: An example of Annotation declaration and use.

programmatically generating the source code, and the possibility of introducing errors which would cause the compilation of the fragment to fail at run-time.

In this paper we will investigate a different approach, using program metadata to drive in a semi-declarative way the run-time synthesis of executable code. We will refer to Java and the JVM throughout the paper, but the main ideas can be applied to .NET languages as well, as in part already done in [3, 11]. Section 2 will briefly introduce Java 5 *Annotations*, and is followed in Section 3 by a presentation of the `@Java` language we defined to extend Java 5 Annotations. Section 4 presents the code manipulation operations we have defined for `@Java`, while Section 5 discusses a number of applications for dynamic bytecode manipulation through annotations. Section 6 offers some conclusions and ideas for future work, and completes the paper.

2 Java Annotations

2.1 The annotations model in Java 5

The *Annotations*[1] introduced in Java 5 allow programmers to associate metadata to specific program elements. These metadata are characterized by an identifier (akin to a class or interface name) and by a signature (or schema), akin to the fields of a class, where each field has an identifier and a value. Custom Annotation types are declared with a syntax similar to that of a class, through the `@interface` keyword. Only field of basic types, String, Class, Enum, Annotation, or arrays of the same are allowed, and default values for them can be defined in the declaration of the Annotation type (see example in Figure 1-a.). More precisely, Annotation support in Java includes:

[1] In the following we will use the term Annotation, with a capital A, to refer to the specific form of annotations as used in Java.

- a syntax to declare Annotation types (Figure 1-a.);
- a syntax to annotate program elements with instances of Annotation types (Figure 1-b.);
- an API and library to inspect through reflection the annotations associated to program elements;
- a format specification, stating how annotations are stored in .class files;
- a tool (called `apt`, annotation processor tool) for generic processing of annotations in source code at pre-compile time;
- an API and associate library for generic programmatic processing of annotations through the `apt` facilities.

Since Annotation declarations are themselves program elements, Annotations can be annotated as well. Two of these meta-annotations (i.e., program meta-meta-data, data describing how data about the program should be interpreted) are of particular relevance for our purposes. The first is the *retention policy* of an annotation type, allowing the programmer to define its life time: source only (and discarded upon compilation), source and .class (and discarded upon class loading), or runtime (preserved in the running system). The second is the *target* of an annotation type, allowing the programmer to define to which program elements it can be attached: other annotations, constructors, fields, local variables, methods, packages, formal parameters, types.

It can be easily seen how with the ability to define the name, schema, lifetime, location and target of each custom annotation, all the features of our characterization of annotations from Section 1 have been placed under control of the programmer.

2.2 Limitations of the Java 5 annotation model

While the annotation model presented in the previous section is sufficiently comprehensive for the vast majority of applications, it suffers one major drawback for the purpose of dynamic code manipulation: a too coarse granularity level. In fact, while the target granularity for data is a single field, parameter, or local variables, the target granularity for code is a single method. This choice is reasonable in consideration of the fact that methods are the smallest code elements which can be found in class signatures[2], but any code manipulation system that can only manipulate entire methods could be expressed more easily by using typed "function pointers" (e.g., the delegate model in C#), and would not be suitable for fine grained optimization or configuration. We will discuss why fine grained manipulation is useful in Section 5.

Another minor limitation is that, unlike C#, only a single instance of a given annotation type can be applied to a given target, even when the Annotation fields would be different. For example, with reference to Figure 1, we cannot place multiple `Requirement` annotations on a method, to signify that it satisfies several requirements at once. This limitation (for which we could not find a documented design rationale, and that apparently could be easily lifted by

[2]But notice that the same principle has not been applied to data, in that local variables are not visible in class signatures, while all other possible targets are.

```
int i;
double t=0;
for (i=0; i<a.length; i++)
   t+=a[i];
@Parallel for (i=0; i<a.length; i++)
   a[i] /= t;
```

Figure 2: An example of statement annotation in @Java.

extending a few Reflection API methods) can be overcome by using array-typed fields, at the cost of some complication in the code. In our example, we could have used a `String` array for the `id`, `certifiedBy`, `complianceStatement` and `date` fields. This, however, is a solution which is somewhat contrived, more error-prone and less general than one could desire.

It should be noted that both these limitations have been identified in other contexts as well. For example, there is ongoing work on allowing annotations on each usage of a type which are being discussed for adoption in Java 7 [10] (the same document cites allowing multiple instances of an annotation on the same target as an example of possible developments).

3 The @Java language

Following the example set in [3], we propose to extend the Java language in order to allow Annotations to be placed on code fragments inside a method, or more precisely, on any statement.

The resulting language, called @Java, can be reduced to Java 5 by a preprocessor, which serves as compiler for the language.

3.1 Syntax extension

@Java differs from Java by a single syntax rule, namely

```
Statement ::= Annotations Statement
```

which allows annotations to be placed in front of any statement. We refer here to the Java 5 grammar as presented in [14]§18.1; a more concrete definition which exploits lookahead to optimize parsing time in the implementation is provided in [12]. A typical example of a fragment of @Java code is presented in Figure 2, where a statement annotation `@Parallel` is used to indicate that all iterations of a `for` loop could be executed in parallel.

3.2 Compilation strategy

The compilation of @Java to Java must satisfy two fundamental requirements:

1. the result of the compilation must be a valid Java 5 program;
2. it must be possible to retrieve which statements were annotated with which Annotations from the .class data produced by the Java compiler;

The first requirement can be satisfied by simply removing the statement annotations, which is easily performed by visiting the syntax tree of the `@Java` program while skipping the annotations nodes corresponding to the grammar rule above. The second requirement is more tricky, given that we want to use a standard Java compiler for the back-end compilation.

The strategy we implemented is as follows:

1. an annotated statement of the form $A(\vec{v})\ S$ or $A\ S$, where A is an Annotation and S is a statement, is replaced with a block of the form $\{\ K_b(k)\ S\ K_e(k)\ \}$ where $K_b(k)$ and $K_e(k)$ are special statements which serve as markers (these will be discussed in the following), and k is a unique identifier for the statement annotation instance.

2. an Annotation is generated for the method containing the annotated statement, with two fields: an array of identifiers ids and, in parallel, an array of Annotations $anns$. The contents of these arrays are initialized so that, for each index i, $id[i] = k$ and $anns[i] = A(\vec{v})$ (or A if the form of annotation without arguments was used).

Thus, statement annotations are lifted to the method (a legal target according to Java), stored in an array of Annotations (to overcome the problem with multiple instances of the same Annotation type on a single target), and linked by its index i to the unique identifier k in the parallel array of identifiers, which is also part of the method annotation. k in its turn is used to link to the marker statements $K_b(k)$ and $K_e(k)$. These statements must be such that (i) their presence does not alter the semantics of the program, (ii) they can be localized in compiled bytecode, together with their unique key k, (iii) they cannot be optimized away or otherwise corrupted by any Java compiler.

All these properties can be obtained by using as markers method calls to non-final, static methods of a special dummy class, with empty bodies, and having k as their single argument. In particular, since their bodies are empty calling these methods does not alter the semantics, per (i). The method call sequence consisting of a `iconst_`n, `bipush`, `sipush`, `ldc` or `ldc_w` instruction to push k on the stack, followed by a `invokestatic` instruction to a distinguished method is easily identifiable in the code, satisfying (ii). And finally, since the dummy class could be changed after compilation of the invocation, the compiler cannot optimize away the call by inlining the body, which guarantees (iii).

A few observations are in order. First, it should be noted that the bytecode sequence is easily identifiable, but not unique. A similar snippet, consisting of a push followed by a method call, could also be generated in the course of evaluating an expression like $k + o.M()$, where it would be followed by an `add` instruction. However, since we define only a version of the method M with a single argument k, cases like the above would be flagged as errors by the compiler, so the risk if erroneously identifying the bytecode fragments for the $K_b(k)$ and $K_e(k)$ sequences is minimal, and in practice confined to hand-crafted bytecode.

Second, the method calls $K_e(k)$ and $K_b(k)$ do not alter the functional semantics of the program, but they could alter its performance, and possibly adversely impact the meeting of non-functional requirements, since method invocation add a small performance penalty. However, while the Java compiler cannot inline or optimize away the method calls, an adaptive optimizing JIT

```
public void M()
{
  ...
  @A while (...)  {
    cnt++;
    @B(c=1) for (T i:  coll) {
      ...
    }
  }
}
```

```
import jcodebrick.Fragment;
import jcodebrick.MultiA;
...
@MultiA(
  ids={1,2},
  value={@A,@B(c=1)}
)
public void M()
{
   ...
   Fragment.begin(1);
   while (...)  {
     cnt++;
     Fragment.begin(2);
     for (T i:  coll) {
       ...
     }
     Fragment.end(2);
   }
   Fragment.end(1);
}
```

Figure 3: An example of the source-to-source translation performed by the `@Java` compiler. On the left, the source `@Java` code; on the right, the result of the translation.

compiler can, and usually will, so in practice even non-functional semantics is preserved.

Third, since method calls in Java can have side effects, and the compiler cannot be sure which body will be executed for non-final methods (as already discussed above), it is extremely unlikely that even an aggressively optimizing compiler will move code across $K_b(k)$ and $K_e(k)$ borders[3], so we can rely on the fact that the compiled bytecode contained between $K_b(k)$ and $K_e(k)$ markers is indeed the complete and only code for the annotated statement S.

Figure 3 shows an example of how an `@Java` code fragment is translated to Java by the `@Java` precompiler. As a side note, observe how the compilation scheme can be applied to the empty statement `;` (e.g., `@Pos;`), hence `@Java` annotations can be used to assign symbolic names to specific positions in the source code. On the other hand, statement annotations cannot be applied to `return`, `throw`, `break` and `continue` statements, because in that case, the $K_e(k)$ end markers would be flagged as unreachable code by the Java compiler.

[3]With the potential exception of deferred stack pops, which however would not affect the semantics, as the operand stack is supposed to be stable on statement boundaries, and of the evaluation of side-effect free expressions which only read local variables and assign to local variables, which could be moved around by the compiler: see [13] for a discussion of such cases.

4 Manipulating annotated code

As we have seen in the previous section, the statement annotations introduced by `@Java` can be used in three capacities:

- to express metadata about program fragment, serving all the needs we introduced in Section 1 (but with a finer granularity, so that metadata can be more precisely attached to code w.r.t. the standard model of Java 5);
- to assign symbolic names to specific positions in the source code, with a single-statement granularity; such symbolic references will be available also at runtime, in executable code.
- to assign symbolic names to code fragments, both in source and corresponding bytecode, and again available at runtime.

We will not discuss in this paper applications of the first role that statement annotations can serve, focusing instead of using the other two roles for *dynamic bytecode manipulation*.

In fact, given the availability at runtime of a system of symbolic names for places and fragments, established in the source code (or even programmatically, in more contrived cases), and coupling that with the dynamic class loading system provided by the JVM, it becomes possible to insert, delete or move around parts of the program, and immediately execute the resulting code.

4.1 The JDAsm library

The code manipulation operations are offered to the programmer through the API provided by a library called JDAsm [12]. Similar in spirit to other code manipulation libraries like BCEL [1] or JavaAssist [5, 4], JDAsm was developed with the goal of offering ease of use through the use of statement annotations, insulation from the actual bytecode, and good performances, to allow extensive use at run-time.

We use a lazy evaluation strategy; code manipulation operations requested by the program are queued and not evaluated, until a `build` operation is invoked; at that point, the queued operations are applied in order, and a new class is generated in-memory hosting the resulting code. In addition to increasing performance, since no intermediate code or classes have to be generated in the course of the manipulation, this lazy strategy offers an opportunity for optimizing the operation queue (e.g., all operations modifying a fragment which is later deleted can be skipped altogether) before the actual build[4].

In the rest of this section, we will describe the operations offered by the library, together with the formal definition of some of them (other operations are defined in a similar way, see [13] for a fuller account), and an example of application.

[4]The current implementation does not apply any optimization; these issues are scheduled for future work.

4.2 Notation and definitions

Every method of a Java class stores the local variables into the *local variable array*. We use $\mathcal{L} \subset \mathbb{N}$ to indicate it, treating a variable just as the index of its position in $\mathcal{L}$; since the variables are stored in $\mathcal{L}$ in growing order starting from index 0, it will be $\mathcal{L} = [0, \ldots, n)$. We next introduce the domains of the variables $\mathbb{V}$ and of the instructions $\mathbb{I}$, and define the following functions to obtain the variables an instruction can read, and those it can write:

$$rv \colon \mathbb{I} \to \mathcal{P}(\mathcal{L})$$

$$wv \colon \mathbb{I} \to \mathcal{P}(\mathcal{L})$$

Let $\mathbb{M}_C$ be the set of all the methods of a Java class C, and let $i \in \mathbb{I}$ be the instance of an instruction, we use $IL = \langle i_1, \ldots, i_n \rangle \in \mathbb{IL}$ to indicate an instruction list (either the body of a method or just a part of it). Then we define the following:

$$\mu \colon \mathbb{M}_C \to \mathbb{IL}$$

as the function that given a method $m \in \mathbb{M}_C$ returns all its bytecode as a list of instructions; with a slight abuse of notation we will write $IL \subseteq m$ to indicate that IL is a contiguous sublist of $\mu(m)$. We use the function ι to retrieve the index of an instruction i in an instruction list:

$$\iota \colon \mathbb{IL} \times \mathbb{I} \to \mathbb{N}$$

to simplify the notation, we will overload ι as follows:

$$\iota \colon \mathbb{M}_C \times \mathbb{I} \to \mathbb{N}$$

$$\iota(m, i) = \iota(\mu(m), i)$$

The set of local variables referred to by an instruction or an instruction list (again, overloading the notation for simplicity) is defined as

$$loc(i) = rv(i) \cup wv(i)$$

$$loc(IL) = \bigcup_{i \in IL} loc(i)$$

Let α be a statement annotation inserted in the source code to mark a statement (typically a block statement) inside a method $m \in \mathbb{M}_C$. Then we define a *Fragment* f as the section of bytecode of m identified by the triple $r = \langle id, \alpha, m \rangle$, where the id is the unique identifier generated by the pre-compilation parser. A fragment is the smallest part of code that the user can manipulate by moving it and deleting it. It is defined as:

$$f = \langle i_b, i_e, r \rangle \;\text{ where }\; i_b \in r.m,\; i_e \in r.m,\; b < e$$

Each fragment f is delimited by two *markers* called starting marker K^f_b (immediately preceding i_b) and ending marker K^f_e (immediately following i_e). Each marker is a two-instruction sequence, $K^f_{b_1}$ and $K^f_{b_2}$, $K^f_{e_1}$ and $K^f_{e_2}$, which are the result of compiling the marker method calls inserted by the `@Java` compiler in place of a statement annotation. They include:

- An instruction $K^f_{b_1} = K^f_{e_1}$ to push onto the stack the value of $f.r.id$
- A static call to an empty method, one for any $K^f_{b_2}$ and another one for any $K^f_{e_2}$

Between the markers, f includes $l >= 0$ inner instructions, and we use this function to get them:

$$\nu(f) = IL$$

Thus, given a method $m \in \mathbb{M}_C$ of n instructions, and a fragment f of length l in m, we will have

$$\mu(m) = \langle i_1, \ldots, K^f_{b_1}, K^f_{b_2}, i_j, \ldots, i_{j+l-1}, K^f_{e_1}, K^f_{e_2}, \ldots, i_n \rangle$$

A fragment is *valid* if it does not contain any jump instruction targeting an instruction outside of the fragment, with the exception that a jump immediately after the end of the fragment (i.e., to the first instruction following the last instruction in f) is considered valid. This condition excludes as valid fragments any part of code which contains a `break` or `continue` instruction which would continue the execution to locations not included in the fragment, and, depending on the compilation scheme used by the Java compiler, certain statements with `return` or `throw` clauses embedded in an outer `try-catch-finally` statement (in all these cases, the compiled code would include a jump to the code for the `finally` clause). In the following, we concern ourselves only with valid fragments.

It should be noted that multiple fragments in m never overlay each other and are always correctly nested, i.e. they are either disjoint, or one is entirely contained in the other. This is guaranteed by the grammar of `@Java` under the assumption that the compiler preserves the nesting structure of blocks in the compiled code (an assumption which holds true for all current major compilers). For instance, given two fragments f' and f'' (appearing in this order) in the same method m, their markers K'_s, K'_e, K''_s,K''_e, and the index in the $IL \subseteq m$ of such markers, $a = \iota(m, K'_s)$, $b = \iota(m, K'_e)$, $c = \iota(m, K''_s)$, $d = \iota(m, K''_e)$, then either $a < b < c < d$ or $a < c < d < b$.

4.3 Operations

We define four operations over fragments:

- op_{src}, to *search* and retrieve fragments;
- op_{ins}, to *insert* a fragment at the start or end of another fragment;
- op_{del}, to *delete* a fragment from the code in which it appears;
- op_{xtr}, to *extrude* a fragment, and execute it outside its context.

In the following, we provide a full formal definition only for op_{ins}, while for other operations we provide only a partial definition to support the intuition, omitting some details due to space considerations.

4.3.1 Search

Through the search operations the user is able to retrieve and get a reference to the fragments declared in a Class C. The operation is offered in several overloaded forms, allowing searches according to different criteria. Remembering that $r = \langle id, \alpha, m \rangle$, then we have this four overloaded operations (all forms take a class or a single method as argument, and then more arguments to specify which annotated fragments in the class should be retrieved):

$op_{src} \colon \mathbb{C} \times \mathbb{N} \rightarrow F$	given a class and an id, returns the fragment with that id in the class;
$op_{src} \colon \mathbb{M}_C \rightarrow \langle F_1, \ldots, F_n \rangle$	given a method, returns the list of fragments defined in that method;
$op_{src} \colon \mathbb{C} \times \mathbb{A} \rightarrow \langle F_1, \ldots, F_n \rangle$	given a class and an annotation type, returns the list of fragments annotated with that type in the class;
$op_{src} \colon \mathbb{M}_C \times \mathbb{A} \rightarrow \langle F_1, \ldots, F_n \rangle$	given a method and an annotation type, returns the list of fragments annotated with that type in the method.

For brevity we omit here a formal definition of these operations, which are clerical in nature; the interested reader can refer to [13] for the details.

4.3.2 Insertion

Through the insertion operation the user can inject the bytecode of a source fragment f_s into a specific position in a method m of a class C. The destination position is related to a destination fragment f_d, and it can be one of *before_start*, *after_start*, *before_end*, *after_end*, which indicate, respectively, that f_s is to be inserted before the starting marker of f_d, after the starting marker of f_d, before the ending marker of f_d, and after the ending marker of f_d. The possibility of inserting code inside and outside the destination markers has consequences in concatenated operations that involve the destination fragment f_d more than once. For instance, given four fragments A, B, T, Z, by inserting A into T in position $before\_start$, then inserting B into T in position $after\_start$, and finally inserting T into Z, the code of B will be carried into Z through T, but not the code of A, which has been inserted outside the markers of T.

Given a fragment f, let $IL = \nu(f)$ be its instruction list. IL can use and modify local variables, so we need to consider the source method $m_s = f_s.r.m$, the destination method $m_d = f_d.r.m$ and their respective local variables. Any variable has its own scope; the following function:

$$scope(IL, v) = (j, k) \quad | \, v \in \mathbb{V} \quad j, k \in \mathbb{N}$$

is defined to return the pair j and k as the boundary index of the instructions in IL where the scope of v is valid (this information is provided by the Java compiler among the metadata carried with Java classes, in the table `LocalVariableTableAttribute`).

Given an instructions list IL, a *free variable* $v' \in loc(IL)$ is a variable whose scope is defined outside IL:

$$v' \in loc(\nu(f_s)) \quad | \, (j, k) = scope(\mu(m_s), v'), j < \iota(m, k^f_{s_1}) \wedge k > \iota(m, k^f_{s_2})$$

When we want to deal with insertion of a source fragment f_s that uses the free variable v', we need the user to specify a valid mapping among all the free variables in f_s with the variables in m_d whose scope covers the insertion point. We use a function to get the subset of $loc(IL)$ of all the free variables in IL:

$$floc(IL) = \{v \in loc(IL) \mid v \text{ is free}\}$$

Let $V_m = \{v_{s_1} \rightarrow v_{d_1}, \ldots, v_{s_n} \rightarrow v_{d_n}\}$ be a user defined mapping that associates to any free variable v_{s_i} of f_s a valid variable v_{d_i} of f_d (valid variables are those that are in-scope at the insertion point and have the appropriate type; the mapping is specified by name in the implementation for ease of use, but here we will only refer to the variable indexes), then we define the operation of *insertion* as the function that given a source fragment f_s, a destination fragment f_d, a position p and a mapping V_m, inserts the new fragment in the same method m_d of f_d, and returns m'_d to indicate that the instruction list IL of m_d has been modified.

$$op_{ins} \colon \mathbb{F} \times \mathbb{F} \times P \times \mathbb{V}_m \rightarrow \mathbb{M}_C$$

Other aspects have to be considered in addition to free variable mapping in implementing this operation. In particular, there are cases where the insertion cannot be performed in a type-safe way. If the source bytecode contains a return instruction, we have to check that the return type is compatible with the return type of the destination method. To model this, we introduce the following functions:

$$ret \colon \mathbb{I} \rightarrow Type$$

$$ret \colon \mathbb{M}_C \rightarrow Type$$

(where $Type$ is one of the basic types of the JVM) defined as:

$$ret(i) = \begin{cases} t & \text{if } i \text{ is the } \texttt{RETURN} \text{ instruction for type } t \\ \emptyset & \text{otherwise} \end{cases}$$

$$ret(m) = \bigcup_{i \in \mu(m)} ret(i)$$

with $|ret(m)| \leq 1$, that is, since we are working on an already loaded class, guaranteed by the bytecode verifier.

If $\exists i \in \nu(f_s)$ such that $ret(i) \neq \emptyset \wedge ret(i) \neq ret(m_d)$ (i.e., a return instruction whose type differs from that of the method it is being injected into), then the fragment is not compatible with the method and the insert operation fails returning an error. As we have already seen, free variables are renumbered through the user-supplied mapping V_m; all other variables need to have their index shifted so that they do not conflict with the local variables of f_d. Since all variables in m_d use their own index into the local variable array $\mathcal{L}$ and the variables $V \in loc(\nu(f_s))$ with $V \notin floc(\nu(f_s))$ might use the same indexes, to avoid the risk of overlaying the two sets, we compute the higher index h used by m_d and add h to any index used in V.

Furthermore we consider the possibility that $IL = \nu(f_d)$ is included inside a *try-catch* block. Since we cannot determine by looking at IL alone if its instructions can raise an exception, we conservatively assume that they can,

```
       goto end
catch: new jcodebrick/FragmentRTE
       dup_x1
       swap
       invokespecial jcodebrick/FragmentRTE."<init>":(Ljava/lang/Throwable;)V
       athrow
end:
```

Figure 4: The IL code for the `catch` blocks appended at the end of fragments for the insertion operation.

and surround the inserted code with a brand new *try-catch* block that will catch any exception, and handle it by throwing a new `RuntimeException` (having the original exception in its `cause` field) in the catch block.

It should be noted that our choice is not the only possible one. Another possibility would be to update the signature of m_d to accommodate for the additional exceptions which could be raised by the inserted fragment. This choice however would violate the API contract between the method and its callers, and make seamless replacement of code difficult, while our approach, based on the unchecked `RuntimeException`, does not suffer from this difficulty.

We define an exception as the tuple $exc = \langle ExcType, j, k, h \rangle$ with its type, the indexes j and k as delimiters of the scope of the *try* block and the index h of the first instruction of the *catch* block. This information is held in the Java class file into the `exception_table` field of the `Code_attribute` for the method. We will indicate with $et(m)$ the exception table of a method m, according to its `Code_attribute`, containing metadata about the type and indexes of all try/catch blocks, and with $te(m)$ the set of $ExcType$s thrown by a method m, according to its signature.

The set of exception types which might be thrown by a fragment $f = \langle i_b, i_e, \langle id, \alpha, m \rangle \rangle$ is defined as follows:

$$tc(f) = te(m) \cup \{ET \mid \langle ET, j, k, h \rangle \in et(m) \wedge j \leq b \wedge e \leq k\}$$

The code that will be inserted at the end of f_s in case we have to add the *catch* block will be that shown in Figure 4; we will denote that instruction list with IL_{RT}.

With the above definitions, we say that an insertion operation $op_{ins}(f_s, f_d, p, V_m)$ is *valid* if the following conditions are met:

1. $floc(f_s) = domain(V_m)$;
2. $\forall v \in range(V_m), scope(\nu(f_d), v) = (j, k) \implies j \leq ip(f_d, p) \leq k$;
3. $\forall (v \to w) \in V_m, type(v) = type(w)$;
4. $\forall i \in \nu(f_s), ret(i) = \emptyset \vee ret(i) = ret(f_d.m)$.

where $domain(m)$ and $range(m)$ are, respectively, the set of keys and of values in a mapping m; $ip(f, p)$ returns the index of the insertion point for a fragment f with a position p (it will be the index of the begin or end marker of f, depending on p), and $type(v)$ is the VM type of a variable v.

An invalid insertion operation results in an `InvalidBuildException` being thrown at build time, and the operation is aborted. If the operation is valid, the insertion proceeds as follows.

First, the local variables in the IL associated with the source fragments are renumbered, to avoid clashes with the variable already used in the destination fragment. Then, free variables are mapped according to V_m, and finally a *try-catch* block is added, if needed, to capture and turn into `RuntimeException` all exception thrown by the source fragment which are not handled in the destination method. Formally, this process is described in the following.

Let $h = \max_{v \in loc(\mu(f_d.m))}(v)$ be the index of the highest-numbered local variable in the destination method. Then a new instruction list $IL' = \langle i'_1, \ldots, i'_n \rangle \cdot \theta$ is obtained by copying and modifying the instruction list of the source fragment $IL = \langle i_1, \ldots, i_n \rangle$ in such a way that

$$i'_j = \begin{cases} i_j \left[{v+h}/{v} \right] & \text{if } v \in loc(i_j) \text{ and } v \text{ is not free} \\ i_j \left[{w}/{v} \right] & \text{if } v \in loc(i_j) \text{ and } (v \to w) \in V_m \\ i_j & \text{otherwise} \end{cases}$$

and

$$\theta = \begin{cases} IL_{RT} & \text{if } tc(f_s) \setminus tc(f_d) \neq \emptyset \\ \langle\rangle & \text{otherwise} \end{cases}$$

The resulting method m'_d will be such that its instruction list will be updated to insert IL' at the location specified by p and f_d; its exception table is updated to include the possible addition of *try-catch* blocks for the inserted fragment; and its `LocalVariableTableAttribute` is updated to include the new local variables carried into the method by the inserted fragment. In all other respects (e.g., signature, `throws` clause, debug attributes, etc.) m'_d is identical to m_d.

We only define fully the case for $p =$ *before_start* (the other cases are totally analogous), where if

$$\mu(m_d) = \alpha \cdot \langle K^{f_d}_{b_1}, K^{f_d}_{b_2} \rangle \cdot \beta \cdot \langle K^{f_d}_{e_1}, K^{f_d}_{e_2} \rangle \cdot \gamma$$

then the result of the insertion is m'_d such that

$$\mu(m'_d) = \alpha \cdot \langle {K'}^{f_s}_{b_1}, K^{f_s}_{b_2} \rangle \cdot IL' \cdot \langle {K'}^{f_s}_{e_1}, K^{f_s}_{e_2} \rangle \cdot \langle K^{f_d}_{b_1}, K^{f_d}_{b_2} \rangle \cdot \beta \cdot \langle K^{f_d}_{e_1}, K^{f_d}_{e_2} \rangle \cdot \gamma$$

and

$$et(m'_d) = et(m_d) \cup E$$

where ${K'}^{f_s}_{b_1}, {K'}^{f_s}_{e_1}$ are similar to $K^{f_s}_{b_1}, K^{f_s}_{e_1}$, respectively, except in that they have a fresh unique id (a larger id may require a different opcode), and

$$\begin{aligned} E \quad = \{ \quad & (ET, j, k, k) \mid ET \in tc(f_s) \setminus tc(f_d), \\ & j \text{ is the initial index of } IL' \text{ in } \mu(m'_d), \\ & k \text{ is the index of the } \texttt{catch} \text{ label from } \theta \text{ in } \mu(m'_d) \qquad \} \end{aligned}$$

As a final technicality, the `max_stack`, `max_locals`, `code_length`, `code`, `exception_table_length`, `exception_table`, `attribute_info` of the `Code_attribute` for m'_d are updated as needed, and a copy of the Annotation for f_s with the new fresh id used in ${K'}^{f_s}_{b_1}$ and ${K'}^{f_s}_{e_1}$ is added to the annotations for m'_d.

4.3.3 Deletion

To delete a fragment f from a method m means to re-emit the bytecode of m without the instructions delimited by K_b^f and K_e^f. We define three types of deletion: *delete_without_markers*, *delete_with_markers*, *delete_only_markers* where respectively the bytecode included by f is deleted but the markers are not, the bytecode is deleted and the markers are too, and only the markers are deleted while the bytecode included in f is left untouched.

The operation of deletion is defined as the function that, given a method m, a fragment f in m, and a type t of deletion, returns m' which is identical to m except that part or all of $\mu(m)$ is not present in it anymore:

$$op_{del} : \mathbb{M}_C \times \mathbb{F} \times T \rightarrow \mathbb{M}_{C'}$$

Since the `@Java` compiler inserts fragment markers only at the begin and at the end of a statement, we are guaranteed that a deletion cannot overlap a *try-catch* block nor the scope for a variable, and that the corresponding fragment cannot contain an instruction which is a target from an external jump instruction. Furthermore, since the Java compiler always adds an explicit `return` instruction at the end of a void method, we are assured that the return type from a method's code cannot be changed by a deletion. Hence, a deletion does not need any structural change to a method.

We only define fully the case for $t =$ *delete_without_markers* (the other cases are totally analogous), where if

$$\mu(m) = \alpha \cdot \langle K_{b_1}^f, K_{b_2}^f \rangle \cdot \beta \cdot \langle K_{e_1}^f, K_{e_2}^f \rangle \cdot \gamma$$

then the result of the deletion of f is m'_d such that

$$\mu(m') = \alpha \cdot \langle K_{b_1}^f, K_{b_2}^f, K_{e_1}^f, K_{e_2}^f \rangle \cdot \gamma$$

We also need to remove from the exception table all the *try-catch* blocks which were entirely contained in the removed fragment f, and possibly compact the local variable table by removing all variables whose scope was entirely within f. Again, addresses in $\mu(m')$ are renumbered, and the various `Code_attribute` fields are recomputed as needed. These operations are similar to those we already described for op_{ins}, and for brevity we do not provide all the details here (the interested reader can refer to [13] instead).

4.3.4 Extrusion

The extrusion operation makes it possible to execute the code of a fragment as a self-sufficient method, outside of its original context. The result of the operation is a new class, containing a single static method `exec` (and the default empty constructor), whose body is the IL of f.

$$op_{xtr} : \mathbb{F} \rightarrow \mathbb{C}$$

The signature of the `exec` method is synthesized by looking at the return type of instructions in f (which determine the return type of the method), at the set of free local variables (which determine the arguments number and types), at whether $f.m$ was `static` or an instance method (to determine whether to add an additional `this` parameter), and finally at the exception table for $f.m$ (to determine the `throws` clause for `exec`). In particular:

- The return type for the method is given by $ret(f.m)$, subject of course to the condition that $|ret(f.m)| \leq 1$ (which, however, is already guaranteed by the bytecode verifier). Since the bytecode verifier also guarantees the absence of unreachable code in the source method, it is always the case that the last instruction of a fragment is not a T`return` (remember that we have added two instructions at the end of such fragment, K_e). To further guarantee that any branch in `exec` terminates with a T`return` instruction, in synthesizing the method we append such an instruction at the end of $K_b^f(k) \cdot \nu(f) \cdot K_e^f(k)$, with a fresh k, and optionally preceded by an instruction to push the default value for the return type (see [15]§2.5.1).

- All free variables in $floc(\nu(f))$ are lifted to method arguments, with the appropriate[5] types. As part of this lifting of free variables to arguments, all references to variable indexes in $\nu(f)$ are renumbered accordingly (so that the $n = |floc(\nu(f))|$ free variables lifted to argument occupy indexes $0 \ldots n-1$ and local variables whose scope is entirely contained within f occupy indexes $\geq n$.

- The set of exception thrown by `exec` is determined as

  $$te(\texttt{exec}) = tc(f)$$

  which indicates that any exception which is declared to be thrown by the source method, or caught by a `try-catch` surrounding f, is added to the `throws` clause of `exec`.

It should be noted that changes to local variables of extruded fragments are lost upon return from the synthesized method. This is a limitation of our approach, which derives from the lack of `out` variables in Java.

4.4 Examples

Let us consider an application which has to perform frequently some check on given conditions. These checks can be very thorough and complex, and computationally expensive, but in most cases a more basic and more efficient approximation might be sufficient, depending on environmental conditions. As will be described in Section 5, we envision a situation where the checks have to be performed in real-time, so we do not want to pay the penalty for an indirect method call each time, and decide to use runtime code manipulation instead.

The method performing the checks could be as follows:

[5]Notice that types inferred this way may differ from those in the source code; for example, `short` local variables will be promoted to `int` when lifted as arguments, according to the standard type conversion rules of the JVM [15]§3.11.1.

@Java *Source*	@Java *compiled code*

class C1 {
...
public void m()
{
  ...
  @ComplexChecks {
  /* complex check code */
  }
  ...
}
...
}
	import jcodebrick.Fragment;
	import jcodebrick.MultiA;
	
	class C1 {
	...
	@MultiA(
	  ids={1},
	  value={@ComplexChecks})
	public void m()
	{
	  ...
	  {
	  Fragment.begin(1);
	  /* complex checks code */
	  Fragment.end(1);
	  }
	  ...
	}
	...
	}

The compiled @Java code will be in turn compiled by the Java compiler into the following bytecode:

```
@MultiA{ids={1}, value={@ComplexChecks}}
method m():
   Code:
     // Initial method code
     ...
     // Starting marker K_b
     iconst_1
     invokestatic jcodebrick/Fragment.begin
     ...
       // complex checks code
     ...
     // Ending marker K_e
     iconst_1
     invokestatic jcodebrick/Fragment.end
     ...
     // More method code
     ...
     // End of method
     return
```

The code to replace at runtime the complex checks fragment with the basic checks one, and invoke the modified method, could be as follows:

```
CbClass c = new CbClass( C1.class );
Fragment complex = c.getFragment("ComplexChecks");
Fragment basic = c.getFragment("BasicChecks");
...
complex.insertFragment(Fragment.BEFORE_START, basic);
complex.delete();
C1 cc=(C1)c.build().newInstance();
cc.m();
```

The bytecode of the modified method `m` is obtained through these two steps:

After the insertion	After the deletion

`Code:`
`// Initial method code`
`...`
`iconst_4`
`invokestatic jcodebrick/Fragment.begin`
`...`
`// Basic Fragment code`
`...`
`iconst_4`
`invokestatic jcodebrick/Fragment.end`
`iconst_1`
`invokestatic jcodebrick/Fragment.begin`
`...`
`// Complex Fragment code`
`...`
`iconst_1`
`invokestatic jcodebrick/Fragment.end`
`...`
`// More method code`
`...`
`// End of method`
`return`
	`Code:`
	`// Initial method code`
	`...`
	`iconst_4`
	`invokestatic jcodebrick/Fragment.begin`
	`...`
	`// Basic Fragment code`
	`...`
	`iconst_4`
	`invokestatic jcodebrick/Fragment.end`
	`...`
	`// More method code`
	`...`
	`// End of method`
	`return`

4.5 Performance

Given that one of the major advantages of our proposal over previous research is its ability to perform code manipulation at runtime, we are particularly concerned about its performances.

We have compared the execution times of typical @Java operations using different libraries for bytecode engineering. In particularly, JDAsm performances have been compared to that of BCEL [1] and JavaAssist [5, 6], using the latter both at source level and at bytecode level. In particular, we have measured the performances of the three libraries in the synthesis of a new Java class (as in our build operation), containing a single "Hello world" method.

The experimental results, obtained by averaging 20 runs of the equivalent generating code for the three libraries are shown in Figure 5. As can be seen, JDAsm is substantially faster than both BCEL and JavaAssist in source mode, and offers performances comparable (and slightly better) with those of JavaAssist used in bytecode mode, but with the advantage of being able to compose the method symbolically, rather than having to handle each individual bytecode instruction.

Library	*Time*
BCEL	172ms
JavaAssist (source level)	188ms
JavaAssist (bytecode level)	78ms
JDAsm	62ms

Figure 5: Execution times for the class synthesis benchmark.

5 Applications

The ability to modify the running code of an application in a structured, symbolic and type-safe way, while leaving the programmer able to express code fragments in source form, opens the way to a vast number of novel applications. In the following we will only list a few examples, serving as conceptual scenarios but with no aim of completeness. Before going into the details, it is worth remarking that similar techniques have been used already in the past, albeit typically in an ad hoc fashion, and often at the source level (e.g., classical aspect-oriented programming), or at program installation time (e.g., configuration-selecting installers, as in a OS installer that only installs drivers needed for the actual hardware). In contrast, our proposed technique is totally general, annotation can be used both at the source level and at the bytecode level, and operations can be performed at any stage of the life cycle of the application, even while the application is running and without requiring a restart.

5.1 Logging

At installation time, a program could contain statements whose purpose is to compute and log to some external file certain values describing the state of the application during its execution, as a way of monitoring its performances and correctness. After monitoring the system's logs for a while, it can be determined that the system is behaving correctly, and that there is no longer a need for a detailed log.

Current logging frameworks (e.g., Log4J [2]) can enable or disable the output to the log file dynamically, but cannot avoid computing the values, which might be costly or have other undesired side effects. In contrast, with `@Java` the logging statements (or blocks) can be marked with an annotation such as `@Log(level)`, and when it is determined that logging is no longer required, all the logging blocks below a given severity level can be removed from the running code, thus avoiding any associated computation and possibly improving performances significantly.

As a related example, the `@Log` fragments could be removed leaving the markers in place, and stored in a data structure, together with a reference to their original location. This way, it becomes also possible to reinstate them if at a later time logging is desired again.

5.2 Environment-based reconfiguration

It is often the case that a system has to react differently to certain events based on changing environment conditions. For example, a heavy-load dispatcher for

a web server farm could operate normally under standard operating conditions, while monitoring the response times of the system. If these become too high, it could install in its running code a fragment to monitor incoming requests especially to identify denial-of-service attacks (this might entail maintaining and updating complex data structures, to perform pattern matching on the requests data and to identify sets of IP addresses from which a potential distributed-DoS attack is coming). If no DoS attack is recognized, the dispatcher would go back to the standard dispatch code. On the other hand, if such an attack is identified, the dispatcher could further substitute its request-dispatching code with a more precise, but less efficient, version which would guard against requests coming from potential DoS sources. The assumption here is that the more precise dispatching code, rejecting DoS requests upfront, will save processing costs later on in the requests handling chain. If, after some time, it is determined that the DoS attack has ended, the original, optimistic but faster code can be replaced again inside the dispatcher.

In a more flexible implementation, both attack-detecting code and hardened dispatching code could be loaded dynamically based on the type of attack, thus making the system able to detect and respond optimally to different threats.

Similar behavior could be obtained by calling virtual, abstract or interface methods to perform the monitoring, detection and dispatching functions, and switching to different implementations of the same when appropriate. However, this standard technique would leave several method invocations in place even when they are not needed, which might be undesirable for a very high-performance system. On the contrary, with @Java the mutable code is substituted in-place, with no need for indirection, thus guaranteeing better performances both in the optimistic case and in the hardened one.

5.3 Dynamic optimization

A numeric application could include some heavy computation, which could be performed either in floating point (e.g., using `double`s) or in fixed point (e.g., using `int`s and then scaling the results by a fixed amount). At install time, the application could measure the performances of both, and then insert into its own computation code the version which offers better performances.

Again, similar results could be obtained by guarding the computation with an `if` statement, or by calling a method, but if the variable fragment has to be executed a relevant number of times (which is not uncommon, e.g. with large matrix operations), the cumulative cost of evaluating the flags or calling the methods, multiplied by millions or billions of invocations, could become significant. In contrast, with @Java the insertion of the proper fragment in-line would be performed only once, regardless of the number of times the fragment is run.

It is also worth remarking that the choice between different versions of a code fragment could be done dynamically, possibly switching between multiple versions based on external conditions. For example, using a floating point version can be too costly if another numerical application is running concurrently (e.g., due to the need of storing and retrieving all the FPU registers at every context switch), but may be more convenient otherwise, so the application could periodically re-check the performances of the various versions of the code available, and choose a different one to execute based on current performances (again,

saving on indirection costs as the chosen fragment would be inserted in-line).

5.4 Adaptable declarative security

The native security model in Java is *operational*, meaning that code performing a protected function has to call specific methods to check whether the caller has the right permission to invoke the given function. This might be inconvenient and error-prone, and moreover the entire security model of an application is wired-in once the application is written and compiled[6].

With `@Java`, a programmer can mark relevant sections of code with annotations such as `@GrantPermission(perm)`, `@AcquirePermission(perm)` and `@RequirePermission(perm)`, thus moving to a declarative model instead. One of the advantages is that in `@Java` permission-related annotations can be placed on statements and blocks, thus providing finer control over which sections of the code are critical (and satisfying Denning's principles). Another advantage is that the operational code needed to actually grant, acquire and check permissions can be injected at the appropriate places automatically, and – moreover – it can be changed, at runtime, to suit different security models as appropriate from time to time.

5.5 Parallelization

In parallel applications, it is customary to use dialects of common programming languages extended with keywords used to declare properties relevant for the parallel execution of the code. This approach typically requires custom compilers, which produce parallelism-handling code based on the custom keywords.

As we have seen in Figure 2, we could use a `@Parallel` annotation placed on a `for` statement to declare that the iterations of the `for` are independent and could potentially be executed in parallel. Then, an application could inject in those places code to actually realize the parallelism, choosing whatever implementation is more appropriate for the JVM/OS/hardware combination the program is running on (e.g., no parallelism at all, or creating a certain number of threads or processes based on how many CPUs are available on the machine, etc.).

Even more interesting, with the emergence of virtualization systems, it is becoming increasingly common that an application can be run on a virtualized server, and in that case the server could be dynamically reconfigured to allocate or simulate a variable number of CPUs - in which case, the application can react by changing its parallelization strategy and injecting different thread-handling fragments at `@Parallel` locations.

6 Conclusions and future work

In this paper we have introduced `@Java`, a variant of Java which permits to manipulate an application code at runtime in a structured, symbolic and type-

[6]The Java security model provides for that by externalizing policy decisions in a text file which can be edited by the user, but with limited flexibility, essentially implementing a source-based permission policy.

safe way, by using annotations placed on single statements or blocks to define code fragments and locations in the code.

While sharing similarities in its scope with traditional aspect-oriented programming techniques, our contribution places a greater emphasis on the possibility of manipulating the code at run-time, whereas aspect weaving is typically performed at compile-time only. This important distinction opens the way to a number of applications for which standard AOP techniques are not flexible enough.

The techniques we presented, building on top of execution technology provided by virtual execution environments and on novel language features such as custom annotations, change in a fundamental way the notion of lifecycle of a program. Whereas customarily writing, compiling, linking, shipping, deploying, installing, loading and running a program were considered completely distinct phases, the ability to identify and process annotations both in source and in object (.class) form, and at runtime in executable code, in a sense blends this phases. Now, program code can be written at run-time; compilation can execute user-provided code based on annotations found in source files, an installer can manipulate the object code that has been deployed based on a specific machine architecture, etc.

The `@Java` language and its code-manipulation capabilities are a contribution towards reaching this vision, in which code manipulation and program re-writing is a substantial part of execution. The language itself could be extended to address annotation of (sub-)expressions, to cover cases where one might want to manipulate, say, a `new` expression, or a method invocation. We intend to address this issue as part of future work.

More work is also needed in two other directions: (i) on the application side, by providing run-time support and case studies for common needs (e.g., logging, security, parallelism), and (ii) on the technological side, by providing more flexible and more efficient implementations of the code-manipulation primitives we have defined.

The `@Java` source-to-source compiler and the associated JDAsm code manipulation library have been released as open source, and are currently available, respectively, at `http://at-java.sourceforge.net` and `http://jdasm.sourceforge.net`.

Acknowledgments. The authors would like to thank Antonio Cisternino for providing invaluable input and ideas, and Cristian Dittamo and Nicola Giordani for working on the implementation of related technologies as part of their M.Sc. work.

References

[1] Apache Software Foundation. Bcel: Bytecode engineering library. `http://jakarta.apache.org/bcel`.

[2] Apache Software Foundation. Apache log4j, 2007. `http://logging.apache.org/log4j`.

[3] W. Cazzola, A. Cisternino, and D. Colombo. [a]C#: C# with a customizable code annotation mechanism. In *Proceedings of the 10th Annual ACM*

Symposium on Applied Computing (SAC'05), pages 1274–1278, Santa Fe, New Mexico, USA, Mar. 2005. ACM Press.

[4] S. Chiba. JavaAssist. `http://www.csg.is.titech.ac.jp/~chiba/javaassist`.

[5] S. Chiba. Load-time structural reflection in Java. In *Proc. of ECOOP 2000*, volume 1850 of *LNCS*, pages 313–336. Springer-Verlag, 2000.

[6] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03)*, volume 2830 of *LNCS*, pages 364–376. Springer-Verlag, 2003.

[7] A. Cisternino and V. Gervasi. Meta-programming without quasi-quotation. In *Proceedings of the 2nd MetaOCaml Workshop*, Tallin, Estonia, Sept. 2005.

[8] ECMA. *Standard ECMA-334 – C# Language Specification.* European Computer Manufacturer Association, Geneva, 4th edition, 2006.

[9] ECMA. *Standard ECMA-335 – Common Language Infrastructure (CLI).* European Computer Manufacturer Association, Geneva, 4th edition, 2006.

[10] M. D. Ernst. JSR 308: Annotations on Java types, 2007. (updated on March 2008).

[11] A. K. G. Attardi, A. Cisternino. CodeBricks: code fragments as building blocks. *SIGPLAN Notices*, 38(10):306–314, Oct. 2003.

[12] G. A. Galilei. Applicazioni delle annotazioni alla manipolazione a runtime di codice su macchine virtuali. Master's thesis, University of Pisa, 2007. (in Italian).

[13] G. A. Galilei. Design and implementation of the `@Java` system. Technical Report TR-08-19, Dipartimento di Informatica, University of Pisa, July 2008.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison-Wesley, 3rd edition, 2005.

[15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 2nd edition, 1999.

[16] G. L. Steele. *Common LISP – The language.* Digital Press, 2nd edition, 1990.

[17] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.