Composition of Dynamic Analysis Aspects

Éric Tanter
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Chile
etanter@dcc.uchile.cl

Philippe Moret Walter Binder
Danilo Ansaloni
Faculty of Informatics
University of Lugano – Switzerland
firstname.lastname@usi.ch

Abstract

Aspect-oriented programming provides a convenient high-level model to define several kinds of dynamic analyses, in particular thanks to recent advances in exhaustive weaving in core libraries. Casting dynamic analyses as aspects allows the use of a single weaving infrastructure to apply different analyses to the same base program, simultaneously. However, even if dynamic analysis aspects are mutually independent, their mere presence perturbates the observations of others: this is due to the fact that aspectual computation is potentially visible to all aspects. Because current aspect composition approaches do not address this kind of computational interference, combining different analysis aspects yields at best unpredictable results. It is also impossible to flexibly combine various analyses, for instance to analyze an analysis aspect. In this paper we show how the notion of *execution levels* makes it possible to effectively address these composition issues. In order to realize this approach, we explore the practical and efficient integration of execution levels in a mainstream aspect language, AspectJ. We report on a case study of composing two out-of-the-box analysis aspects in a variety of ways, highlighting the benefits of the approach.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Algorithms, Languages, Measurement

Keywords Aspect-oriented programming, execution levels, aspect composition, dynamic program analysis, AspectJ

1. Introduction

Dynamic crosscutting with the pointcut and advice mechanism is particularly well-suited for defining different kinds of dynamic analyses, such as profiling [2, 6, 26], data race detection [1, 9], or memory leak detection [21, 35]. The advantage of using aspects for dynamic analysis stems from the convenient high-level model offered by join points (representing specific points in the execution of a program), pointcuts (denoting a set of join points of interest), and advice (code to be executed whenever a join point of interest happens). A dynamic analysis aspect is easier to define, tune and extend, compared to an equivalent implementation based on low-level code instrumentation tools.

A practical impediment to define these analyses as aspects is that most aspect-oriented programming (AOP) frameworks do not support aspect weaving in the core libraries of the language because of bootstrapping issues, as well as to avoid infinite regression when advice invoke methods in those libraries. In [35], the authors addressed this issue for Java and AspectJ, using a particular implementation technique based on *code duplication within method bodies* and on *bypasses* to allow reverting to the original bytecode of a method (before aspect weaving). Therefore, it now becomes possible to cast existing dynamic analyses as aspects. However, this raises the need for proper composition of such *dynamic analysis aspects* (analysis aspects, for short).

Aspect composition is a multi-faceted topic [4, 10, 13]. A language like AspectJ only addresses the *shared join point* problem with a limited pragmatic mechanism to declare precedence between aspects. Other proposals have targeted semantic interactions, whereby the effects of two aspects conflict with each other [28, 36]. However, even if two aspects are totally independent from each other (no shared join point, no direct semantic interaction), they may still interfere with each other because their own computation can be observed by the other aspect. Because analysis aspects tend to rely on fairly exhaustive pointcuts (*e.g.,* all invocations, all instantiations, etc.) visibility is critical. Depending on the actual visibility of aspectual computation, different scenarios can be achieved, as will be explained in Section 2. For instance, it is crucial to be able to apply different analysis aspects and to ensure a stability property according to which the set of join points observed by each of them is unaffected. It is also important to be able to analyze analysis aspects (*e.g.,* profiling object allocation in a data race detection aspect); and even to apply a given analysis to itself!

Recently, Tanter proposed *execution levels* [31] as a means to structure aspect-oriented programs in order to avoid problems such as infinite regression and unwanted interference between aspects. Inspired by work on reflective architectures [12, 15], the default semantics of execution levels is that aspectual computation is considered "meta", and therefore occurs at a higher-level than that of the base program. As a consequence, aspectual computation is itself invisible to other aspects. Aspects that need to observe the activity of other aspects can be *deployed* at higher levels. Every aspect observes and potentially affects the computation at the level below it. The language proposed by Tanter includes the possibility for application and aspect programmers to explicitly shift execution up and down if needed, in order to address specific visibility requirements.

So far, execution levels are implemented in AspectScript [33] and in an extension of AspectScheme [17], both of which rely on dynamic weaving. This work addresses the practical and efficient integration of execution levels in a mainstream aspect language like AspectJ, where pointcuts are partially evaluated at compile time [20, 24]. In particular, this work focuses on the integration and composition of analysis aspects.

The original scientific contributions of this work are:

- The design of a coherent subset of execution levels suited to the AspectJ compilation and programming models, which takes into account the specificities of analysis aspects.
- The integration of execution levels in AspectJ, with complete bytecode coverage (including the Java class library), and support for most features of AspectJ. Our tool, called MAJOR2, relies on a generalization and extension of previously developed techniques.
- A validation of our approach in different use case scenarios, as well as an evaluation reporting on the size of woven code and presenting a performance comparison of code woven with MAJOR2 respectively with the standard AspectJ weaver.

This paper is structured as follows: Section 2 motivates the need for execution levels by discussing different ways of composing two independently-developed analysis aspects. Section 3 presents our model of execution levels for AspectJ, and Section 4 details our implementation. Section 5 presents evaluation results. Section 6 discusses related work, and Section 7 concludes.

2. Motivating Scenarios

As motivating scenarios, we consider different ways of composing two analysis aspects, the data race detector `Racer` [9] and the object allocation profiler `Prof`, and different composition scenarios.

The aspects. Figure 1 shows the (refactored and simplified) `Racer` aspect, which is a refined implementation of the Eraser data race detection algorithm [29]. `Racer` reports a potential[1] data race if two or more threads access the same field without holding any common lock, and if at least one of these threads is writing to the field. With each field f, `Racer` associates a set of locks $L(f)$ and a finite-state machine to keep track of read and write operations performed by different threads. The first time f is accessed, $L(f)$ is initialized with all the locks held by the accessing thread. Subsequently, each time f is accessed, $L(f)$ is replaced by the intersection of the locks in $L(f)$ and the locks held by the accessing thread. If $L(f)$ becomes empty and the field has been accessed by more than one thread, at least one of them having performed a write access, a potential data race is reported. `Racer` intercepts acquisition and release of intrinsic locks (expressed with `lock` and `unlock` pointcuts) in order to keep track of the locks held by each thread (the thread-local variable `locksHeld`). It checks for potential data races upon each field access (the `staticFieldSet`, `fieldSet`, `staticFieldGet`, and `fieldGet` pointcuts). Details of the `Racer` implementation are presented in [9].

Figure 2 illustrates the simplified `Prof` aspect. It intercepts all object allocations and keeps some statistics on the number and size of allocated objects (method `profileAllocation`, not shown here). Note that in order to avoid infinite regression, both aspects declare specific scope pointcuts (`scopeRacer` and `scopeProf`) and use them repeatedly at each advice declaration. This ensures that each aspect does not see join points produced by its *own* computation[2]. Such pointcuts are simply redundant with execution levels [31], as will be made clear in Section 3.

Applying both aspects with AspectJ. Applying both aspects out of the box is impossible with AspectJ, yielding a

```
public aspect Racer {
  final ThreadLocal<Bag> locksHeld = new ThreadLocal<Bag>(){ ... };
  pointcut staticFieldSet()   : set(static * *);
  pointcut fieldSet(Object o) : set(!static * *) && target(o);
  pointcut staticFieldGet()   : get(static * *);
  pointcut fieldGet(Object o) : get(!static * *) && target(o);

  pointcut scopeRacer() : !within(Racer) && !cflow(...) && ... ;

  before(Object l) : lock() && args(l) && scopeRacer() {
    Bag locks = locksHeld.get(); locks.add(l);
  }
  after(Object l) : unlock() && args(l) && scopeRacer() {
    Bag locks = locksHeld.get(); locks.remove(l);
  }
  before(Object owner) : fieldSet(owner) && scopeRacer() {
    JoinPoint.StaticPart jpsp = thisJoinPointStaticPart;
    String id = jpsp.getSignature().toLongString().intern();
    SourceLocation loc = jpsp.getSourceLocation();
    checkRaceUponFieldSet(owner, id, loc, locksHeld.get(),
                          Thread.currenThread());
  }
  ... // similar advice for fieldGet, staticFieldGet and staticFieldSet
}
```

Figure 1. Simplified aspect for data race detection.

```
public aspect Prof {
  pointcut scopeProf() : !within(Prof);
  after() returning(Object o) : call(*.new(..)) && scopeProf() {
    profileAllocation(o);
  } ...
}
```

Figure 2. Simplified aspect for object allocation profiling

`NoAspectBoundException`. The reason is that in its initialization, `Racer` instantiates some objects; this triggers `Prof` and starts its initialization, which happens to access a field. This field access provokes the retrieval of the `Racer` instance, whose initialization is however not complete![3]

Even if we manually modify the aspects so as to avoid this initialization circularity, the obtained results are not consistent[4]. The issue is that each aspect “sees” the computation of the other aspect. Concretely, the execution of advice in `Racer` produces join points that are matched by `Prof` (*i.e.* method `checkRaceUponFieldSet` allocates objects), while the execution of advice in `Prof` has join points that are matched by `Racer` (*i.e.* method `profileAllocation` accesses some fields). Hence, the output of each aspect is affected by the presence of the other.

It is actually not even possible to achieve consistent semantics according to which `Racer` analyzes the execution of both the base program and advice in `Prof`, whereas `Prof` profiles the execution of both the application and advice in `Racer`; this is because the scope pointcuts introduced in both aspects to avoid infinite regression rule out all computation that happens in the control flow of each aspect. This means that `Racer` sees `Prof` advice computation only when the advice is triggered from application code, but not from `Racer` code (and vice versa). In order to properly “separate” the aspects such that they analyze only the base program, both aspects need to be modified; in this case, the pointcuts `scopeRacer` and `scopeProf` must be changed to exclude the computation of both aspects from weaving.

[1] The algorithm employed by `Racer` is prone to false positives.

[2] Note that `scopeRacer` includes a control flow check in order to skip all join points produced by its own computation, including its callees, while `scopeProf` only rules out join points that are lexically in its source code. The kind of scope pointcuts to use depends on the specificities of each aspect—but it can have an important impact on performance.

[3] The control flow condition in the `scopeRacer` pointcut does not avoid the issue because the AspectJ compiler generates code that retrieves the aspect instance *prior* to evaluating the pointcut residue.

[4] In AspectJ, `precedence` declarations allow us to control the order of advice that match the same join point. In this example, an eventual `precedence` declaration has no impact, because the sets of join points matched by the two aspects are disjoint.

Clearly, requiring changes of both aspects to weave their composition is bad software engineering practice, violating basic black-box composition and reuse principles. For each composition of different analysis aspects, the involved aspects need to be modified.

Composition scenarios. In addition to the sensible composition semantics described above, there are other interesting scenarios of aspect composition; for example, profiling object allocation in `Racer` with `Prof` (while `Racer` is analyzing a base program), or checking for data races in `Prof` with `Racer` (while `Prof` is profiling a base program). For the development, testing, and optimization of analysis aspects, these scenarios are important. While in some special cases it may be possible to weave the advice methods of one compiled aspect with another aspect, AspectJ provides no proper support for flexibly composing aspects in different ways.

In this paper we present an implementation of execution levels for AspectJ that enables flexible composition of (analysis) aspects in order to support, amongst others, the following scenarios of aspect composition:

1. `Racer` and `Prof` are simultaneously applied to a base program, without perturbating each other.
2. `Racer` is applied to a base program, and `Prof` is applied to `Racer`. That is, `Prof` profiles object allocation in the `Racer` implementation.
3. As above, `Racer` is applied to a base program, but now `Prof` is applied to *both* the base program and `Racer`, thereby profiling all object allocations in the composed program.
4. One instance of `Racer` is applied to a base program, and a *second instance* of `Racer` is applied to code executed by the first `Racer` instance. That is, the second `Racer` instance checks for data races in the `Racer` implementation itself.

These scenarios, as well as many others that are not explicitly addressed in this paper (*e.g.,* applying `Racer` to `Prof`, applying the same instance or different instances of `Prof` to a base program and to `Racer`, etc.) can all be easily expressed with our implementation of execution levels for AspectJ, as we will show in Section 3.

3. Levels for Analysis Aspects in AspectJ

The purpose of this section is to provide background on execution levels [31] and to expose the design of execution levels support in AspectJ. The design focuses on supporting dynamic analysis aspects and therefore does not cover the full-fledged proposal of execution levels as originally formulated by Tanter.

3.1 Execution Levels in a Nutshell

Aspectual computation and meta-circularity. An aspect observes the execution of a program through its pointcuts, and affects it with its advice. An advice is like a method, and therefore its execution also produces join points. Similarly, pointcuts as well can produce join points. For instance, in AspectJ, one can use an `if` pointcut designator to specify an arbitrary Java expression that ought to be true for the pointcut to match. The evaluation of this expression is a computation that produces join points. In higher-order aspect languages like AspectScheme [17], AspectML [14], or AspectScript [33], all pointcuts and advice are standard functions, whose application and evaluation produce join points as well.

The fact that aspectual computation produces join points raises the crucial issue of the *visibility* of these join points. In all languages, by default, aspectual computation is visible to all aspects—including themselves. This of course opens the door to infinite regression issues (*a.k.a.* meta-circularity in reflective architectures). These are typically addressed with ad-hoc checks (like

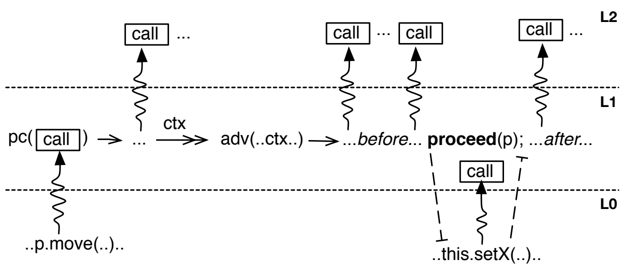

Figure 3. Execution levels in action: pointcut and advice are evaluated at level 1, `proceed` goes back to level 0.

`!cflow(adviceexecution() && within(A)))` in AspectJ, or primitive mechanisms (like AspectScheme's **app/prim** and AspectML's **disable**). However, all these approaches eventually fall short for they fail to address the fundamental problem, which is that of *conflating* levels that ought to be kept separate [12].

Execution levels. In order to address this issue, a program computation is structured in *levels*. Computation happening at level 0 produces join points observable at level 1. Aspects are *deployed* at a particular level, and observe only join points at that level. This means that an aspect deployed at level 1 only observes join points produced by level-0 computation. In turn, the computation of an aspect (*i.e.* the evaluation of its pointcuts and advice) is reified as join points visible at the level immediately above: therefore, the activity of an aspect standing at level 1 produces join points at level 2.

Crucially, an aspect that acts *around* a join point can eventually invoke the original computation. For instance, in AspectJ, this is done by invoking `proceed` in the advice body. The original computation ought to run at the same level at which it originated![5] In order to address this issue, it is important to remember that when several aspects match the same join point, the corresponding advice are chained, such that calling `proceed` in advice k triggers advice $k + 1$. Therefore, the semantics of execution levels guarantees that the *last call* to `proceed` in a chain of advice triggers the original computation at the lower original level.

This is shown in Figure 3. A call to a `move` method in the program produces a call join point (at level 1), against which a pointcut `pc` is evaluated. The evaluation of `pc` produces join points at level 2. If the pointcut matches, it passes context information `ctx` to the advice. The advice execution produces join points at level 2, except for `proceed`: control goes back to level 0 to perform the original computation, before going back to level 1 for the after-part of the advice.

Crucially, execution levels do support the fact that aspects may call into library code (such as the Java class library). This is because execution levels are a property of a flow of control, not of a static entity like a class. Therefore, the same method in the same object may be executed at different levels depending on who calls it.

Level shifting. The default semantics of execution levels are safe by default: computation shifts up whenever a pointcut or advice executes, and shifts down when returning to the original computation. This ensures that infinite regression can never happen, and also ensures that the computation of an aspect is invisible to other aspects at the same level. This safe default may however be too strict in certain cases. The proposal of Tanter includes *level-shifting operators*, **up** and **down** to allow for fine-grained control of visibility.

First, an aspect can be deployed at a higher level, *e.g.,* level 2, in order to observe computation of other aspects. Note that the same

[5] This issue is precisely why using control flow checks in AspectJ in order to discriminate advice computation is flawed. See [31] for more details.

aspect instance can be deployed at different levels if needed. Also, one can use **down** to push some part of an advice to the level below, so that it becomes visible to other aspects standing at the same level. Finally, in order to handle deferred (and possibly concurrent) advice execution, the proposal also includes *level-capturing functions*, *i.e.* functions that are run at the level at which they are defined (irrespective of the level at which they are actually applied). As we will see, only a subset of these features are needed for tackling the issue of composition of dynamic analysis aspects.

3.2 Language Design

We now derive a design for AspectJ with execution levels, based on the specificities of AspectJ, as well as the characteristics of analysis aspects.

Analysis aspects. Aspects that are used for dynamic analysis belong to a peculiar category. According to the classification system of Rinard *et al.* [27], analysis aspects have only *augmentation* advice because after crosscutting, the original computation always executes. This is because analysis aspects are monitoring and gathering information about the execution of a base program, something which is clearly orthogonal to the functionality being advised. Also, analysis aspects are *independent* aspects in the sense that they may read some fields from the base computation but are not writing to fields that the base program (and other analysis aspects) may read or write to.

These specificities of analysis aspects perfectly match the default semantics of execution levels, according to which aspect computation should not be visible to each other. In addition, there is no need for an analysis aspect to "push down" some of its computation. As a consequence, the fine-grained control over execution levels provided by level-shifting operators is *not* required in the language per se when dealing with analysis aspects. It is enough for the weaver to ensure that aspectual computation is run at a higher level. Therefore, our AspectJ extension does not support explicit level shifting at the moment.

Finally, we support before, after and around advice, as well as `if` pointcut designators, as these may be used in analysis aspects.

Levels and threads. The original formulation of execution levels includes a mechanism for defining functions (or in the Java setting, objects) that capture the level at which they are created and always run at that level. While this is definitely an interesting feature, for instance to properly support scheduling threads that execute runnable objects originating from different levels, we do not need it for this case study; its integration is left for future work.

However, the original proposal of Tanter does not address concurrency explicitly: it does not specify at which level newly-created threads should run. In this work, we do need to support multi-threading because it is common for analysis aspects to create some threads, such as "shutdown hooks" to emit the gathered statistics before the JVM terminates (*e.g.,* the profiling aspects in DJProf [26]), cleanup threads to process weak references that have been cleared by the garbage collector (*e.g.,* the memory leak detection aspect of Villazón *et al.* [35]), or thread pools for performing analysis tasks in parallel with program execution (*e.g.,* the calling-context profiling aspect of Binder *et al.* [6] and the parallelized version of `Racer` [1]).

We extend the semantics of execution levels to support multiple threads in the following manner: *a thread runs at the level at which it is created*. Therefore, if an aspect running at level 1 creates a thread, this thread also runs at level 1. This means that it produces join points at level 2, just like the aspect that spawned it.

All threads initially started in the JVM run at level 0. This includes the base program main thread of course, as well as system threads, such as finalizer threads.

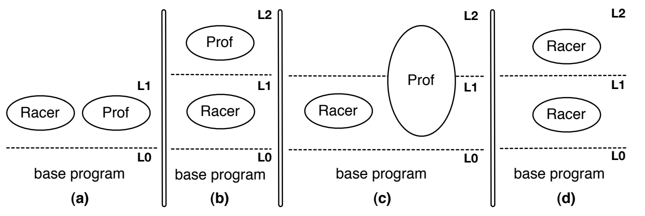

Figure 4. Different deployment and instantiation scenarios.

Aspect deployment. The language proposed in [31] is an aspect language for a higher-order procedural core language with dynamic deployment of aspects. Dynamic deployment means that the language features an explicit `deploy` operator that, when run at level n, dynamically installs an instantiated aspect at level $n + 1$. In contrast, AspectJ does not support dynamic deployment of aspects. Rather, aspects are specified at build time, or at program start up, if load-time weaving is used. Without dynamic deployment, we must rely on some static specification of aspect deployment that includes the level at which aspects are deployed. This moves us outside of the realm of the programming language per se into the realm of the actual weaver used to effectively apply aspects.

Our MAJOR2 weaver operates statically to transform the Java class library, and at load-time for weaving all other classes (both application and aspect classes). It uses `ajc` out of the box for compiling aspects (with `ajc` weaving disabled), and also supports AspectJ annotation syntax with any Java compiler. Once aspects and application classes are compiled, MAJOR2 supports the specification of level deployment, for instance:

```
major2.sh Racer[1] Prof[1] Main
```

deploys both the `Racer` and the `Prof` aspects at level 1 (the default). This corresponds to Figure 4a, while the following deploys `Racer` at level 1 and `Prof` at level 2 (Figure 4b):

```
major2.sh Racer[1] Prof[2] Main
```

The semantics of this configuration is that `Prof` profiles memory allocation of the implementation of the `Racer` aspect.

Aspect instantiation. With dynamic aspect deployment, aspect instantiation is explicit. It is therefore possible to deploy the *same* aspect instance at different levels, so that it observes the computation of all these levels. MAJOR2 also supports specification of aspect instantiation, statically. For instance, we may want to deploy a single instance of `Prof` at both levels 1 and 2 (Figure 4c), so that it profiles memory allocation of both the base program and the `Racer` aspect:

```
major2.sh Racer[1] Prof[1+2] Main
```

We may also want to deploy *different* instances of an aspect at each level. For instance, this can be used to deploy an instance of `Racer` on top of another instance of `Racer`, thereby doing race detection of the race detection aspect itself (Figure 4d):

```
major2.sh Racer[1,2] Main
```

To summarize, in MAJOR2:

- aspect deployment is specified on the command line per aspect, using square brackets containing a comma-separated list of *level specifications*;
- a level specification is either one number, or several numbers separated by the + sign;

- there is a separate instance of the aspect deployed per level specification; each aspect instance stands at the specified level(s).

Codebases. In practice, an analysis aspect is usually complex and does not fit in a single aspect definition file. MAJOR2 supports coarser grained specifications than those we have introduced. In particular, MAJOR2 supports the notion of *codebases*. A codebase gathers a set of of Java packages and archives under a single name. It is then possible to associate an aspect identifier (like `Racer` or `Prof`) to a set of codebases. The interest of codebases is to clearly identify which code only belongs to certain aspect(s), and use that information for optimization purposes, as will be explained in Section 4.7. Beyond the command line interface illustrated above, MAJOR2 supports XML configuration files that embed all codebase and aspect deployment declarations. We do not present configuration files in this paper.

To sum up, on the surface, our extension of AspectJ to support execution levels is limited to the small syntactic extension at startup time to specify instantiation and deployment of aspects with respect to levels. Compared to the original formulation of execution levels, our proposal does not support dynamic deployment of aspects, explicit level-shifting and level-capturing objects, but it includes a specification of the semantics of levels in the presence of threads: threads run at the level at which they are created.

4. Implementing Execution Levels

In this section we discuss our implementation of execution levels in MAJOR2. We explain how execution levels—a dynamic control flow property—are mapped to statically woven code. Our implementation of execution levels in MAJOR2:

1. enables aspect weaving with full bytecode coverage,
2. supports libraries that are shared between the base program and aspects (as well as between aspects deployed at different execution levels), and
3. uses the unmodified AspectJ weaver as a black box.

The first goal is crucial for analysis aspects, as excluding certain classes from weaving would yield incomplete and inaccurate results for most dynamic analyses. The second goal is closely related to the first goal, since both the base program and aspects invoke methods in the Java class library; that is, the Java class library is always shared; our approach also supports the sharing of other libraries. The third goal allows us to use different versions of AspectJ, and (typically) to switch to a new AspectJ release without any changes in our implementation.

We now expose our implementation of execution levels for AspectJ, starting by a high-level overview of the approach (Sections 4.1 and 4.2). We then describe the code transformation approach to integrating execution levels, as well as the handling of multiple aspect instances (Sections 4.3 to 4.6). We conclude by discussing some optimizations (Section 4.7) and limitations (Section 4.8) of the current implementation.

4.1 From Execution Levels to Code Versions

Semantically, the execution of a method produces join points. These join points may be seen by pointcuts that may match them; if so, the corresponding pieces of advice are triggered.

In aspect languages that perform weaving statically, join point production is partially evaluated [24]: based on the static properties of code, it is determined whether or not a given expression can produce a join point that will be matched at runtime [20]. If so, such a join point *shadow* is transformed so as to invoke advice appropriately. If it can be statically determined that the pointcut

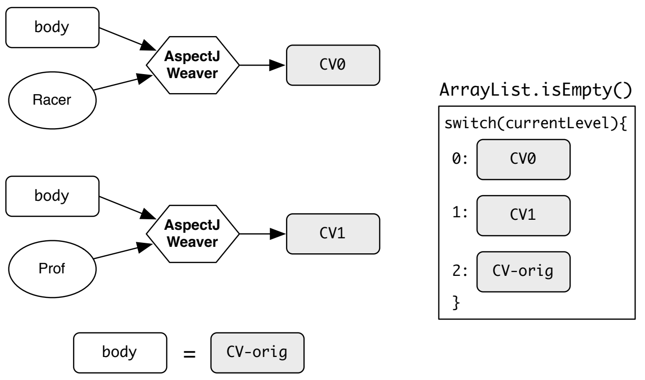

Figure 5. Obtaining and combining different code versions for the scenario of Figure 4b. (In practice, the AspectJ compiler is invoked with, and produces, whole classes.)

however never matches join points corresponding to the shadow, then no transformation happens. The matching of the pointcut may also depend on runtime information not available at compile time: in that case, the shadow is woven together with a *residue*, *i.e.* a conditional expression that guards the invocation of the advice.

With execution levels, the join points produced by the execution of a method vary. If base program code, running at level 0, invokes a method, it produces join points at level 1, that may be matched by aspects deployed at that level. If an aspect deployed at level n calls this same method, then it produces join points at level $n + 1$, visible only for aspects deployed at level $n + 1$.

Taking into account levels when weaving aspects statically can be done in different ways. The most straightforward is to simply weave aspects normally, and add an extra residue at each shadow that checks if the current execution level matches the level at which a given aspect has been deployed. However, this approach implies that for *each and every* join point shadow in a method, the check of levels is performed. A more efficient approach in practice, which we use in MAJOR2, is to factor out the check of levels upon method entry and dispatch appropriately to a particular *code version*. More precisely, there is one code version per execution level, and each code version corresponds to the code with the instrumented shadows of the aspects deployed at the level directly above it.

4.2 Overall Weaving Process

To illustrate the overall weaving process with code versions, let us consider the scenario of Figure 4b: `Racer` is deployed at level 1 and `Prof` is deployed at level 2. Figure 5 depicts how aspect weaving is done for a method that can be called from any execution level, such as `ArrayList.isEmpty()`.

For each execution level, a code version of that method body is generated. It is obtained by invoking the AspectJ weaver with the definition of the aspects deployed at the level immediately above. In this scenario, there are three levels: code executed at level 0 produces join points at level 1, therefore shadows corresponding to `Racer` are inserted in code version `CV0`; code executed at level 1 produces join points at level 2, so shadows are instrumented for the `Prof` aspect in code version `CV1`. Finally, code executed at level 2 is not visible to any aspect, so in this case the code version is the original, uninstrumented method body `CV-orig`.

All three code versions `CV0`, `CV1` and `CV-orig` are combined into a single definition of `isEmpty`: the method starts by checking the current execution level (a property of the running thread), and dispatches accordingly to the appropriate code version. This intuitive description of the weaving process is made more precise and complete in the remainder of this section.

```
1  if (!GlobalState.isBootstrap()) goto WovenBodies;
2  goto OriginalBody;        // bytecode: goto_w
3
4  WovenBodies:
5     int currentLevel = ExecutionLevel.currentLevel();
6     switch (currentLevel) {  // bytecode: tableswitch
7        case 0:   goto WovenWithAspectsAtLevel_1;
8        ...
9        case N-1: goto WovenWithAspectsAtLevel_N;
10       case N:   goto OriginalBody;
11       default:  goto Error; // should never happen
12    }
13 Error: throw new IllegalExecutionLevelError();
14 WovenWithAspectsAtLevel_1: ... // code version for execution level 0
15 ...
16 WovenWithAspectsAtLevel_N: ... // code version for execution level N − 1
17 OriginalBody: ...              // code version for execution level N
```

Figure 6. Code pattern generated for each method, assuming that aspects are deployed at levels $1..N$ (pseudo-code)

4.3 Merging Code Versions

Figure 6 illustrates the general code pattern generated by MAJOR2, merging multiple code versions into a single method body.

The first two lines in Figure 6 enable bootstrapping the JVM with a woven Java class library; they are only generated for methods in the Java class library. The static method `GlobalState.isBootstrap()` returns true while the JVM is bootstrapping. It is crucial not to execute any woven code during JVM bootstrapping, since otherwise the class dependencies introduced by the AspectJ weaver would change the class initialization order and crash the JVM. Because the JVM mandates lazy class initialization [22] and the woven bytecode that would trigger class initialization is not executed during bootstrapping, the original class initialization order is preserved. This approach to bootstrapping a standard JVM with a modified Java class library was first introduced in [7] and also applied in the initial version of MAJOR [35].

Lines 4–12 jump to the code version corresponding to the current thread's execution level. The static method `ExecutionLevel.currentLevel()` returns the execution level of the current thread, which is kept in a thread-local variable.[6] If the current thread's execution level does not correspond to any code version, an `IllegalExecutionLevelError` is thrown (this may only happen if the user wrongly specifies that certain code versions need not be created as an optimization, see Section 4.7). The code versions are simply concatenated in lines 14–17.

Note that all code versions are reached using the `goto_w` and `tableswitch` bytecodes to transfer control to the actual bytecode sequence corresponding to the current execution level. A naive solution would be to use standard conditional branch bytecodes instead, but this could fail because the offset is then limited to signed 16 bit values. This proves to be insufficient in practice because the generated method can be fairly long (all code versions are concatenated). In contrast, `goto_w` and `tableswitch` bytecodes support signed 32 bit values as offsets [22].[7]

4.4 Merging Class Files

If aspects are deployed at n levels, weaving of a class file C_0 involves merging of $n + 1$ class files (if no optimizations are applied): the original class file C_0, and the n class files C_i resulting from weaving C_0 with the aspects deployed at level i using the standard AspectJ weaver. While the merging of code versions at the level of individual method bodies has been explained in Section 4.3, we now discuss issues of merging whole class files.

Weaving with AspectJ may introduce extra methods, such as in the case of around advice [20]. As methods introduced by dynamic crosscutting are relevant only for the class file C_i where they are defined, we rename such methods in the case of name clashes (*i.e.* if dynamic crosscutting introduces two methods with the same name in class files C_i and C_j, $1 \leq i < j \leq n$). After an eventual renaming step, for each method defined in at least one class file, the corresponding code versions in the different class files are merged as described in Section 4.3. If a code version is missing, a jump to the `Error` label is generated (see Figure 6). Constructors are treated in the same way as methods.

Weaving may also introduce extra static fields to hold instances of type `JoinPoint.StaticPart`, which are initialized in the woven class initializer (static method `<clinit>`). If necessary, we rename these inserted static fields so as to avoid any name clashes. All fields are preserved in the final class file after merging.

The class initializer is treated specially. For each class file C_i ($1 \leq i \leq n$), we extract the code that initializes the static fields holding `JoinPoint.StaticPart` instances from the class initializer. Independently of the execution level at which the class initializer is executed, all extracted initialization code must be executed. To this end, in the final class file after merging, we generate a private static method to hold all extracted initialization code and insert an invocation to that method in the beginning of the class initializer. The remainder of the class initializer corresponds to the code pattern presented in Figure 6, where the different code versions exclude the previously extracted initialization code. For the Java class library, which is statically woven, the private static method is not generated. Instead, the inserted static fields and the extracted initialization code are moved to a separate class, in order not to disrupt JVM bootstrapping by introducing class dependencies during the bootstrapping phase; for details, see [7, 35].

Our approach results in the following initialization behavior. The class initializer is executed when a class is used for the first time [19, 22]; it is executed by the thread T that uses the class for the first time. After JVM bootstrapping, each class initializer is executed at the current level of thread T. During JVM bootstrapping, the original class initializer is guaranteed to execute. For classes in the Java class library, the static fields holding `JoinPoint.StaticPart` instances are initialized lazily when they are accessed for the first time. For all other classes, they are immediately initialized.

4.5 Inserting Level-Shifting Operations

Regarding the generation and merging of code versions, the classes of compiled aspects are treated like any other classes. In addition, level-shifting operations are inserted in each advice method and in each method corresponding to compiled `if` pointcuts.

Figure 7 illustrates the code pattern implementing level shifting. The static methods `ExecutionLevel.up()` and `ExecutionLevel.down()` increment respectively decrement the execution level of the current thread (which is kept in thread-local storage) by one. The advice body surrounded by the level-shifting operations corresponds to the merged code versions (see Figure 6). Consequently, when an advice method is invoked by woven code, shifting up is executed before the `switch` statement that jumps to the according code version.

For around advice, level shifting is more complex. While around advice bodies are treated as discussed before, it is also necessary to properly handle the `proceed` calls. Recall from Section 3.1 that the

[6] Conceptually, this variable corresponds to the Dynamic Inserted-Code Bypass (DIB) used by the initial version of MAJOR to avoid infinite regression when advice invoke methods in the Java class library [35]. In fact, from the implementation point of view, the support of execution levels in MAJOR2 is a generalization of the code duplication technique used in MAJOR.

[7] While the total size of a merged method must not exceed 2^{16} bytes (see Section 4.8), `goto_w` and `tableswitch` bytecodes are needed to encode jump offsets $\geq 2^{15}$.

```
ExecutionLevel.up(); // increments the current thread's execution level
try {
   ... // advice body (corresponding to the code pattern in Figure 6)
}
finally {
   ExecutionLevel.down(); // decrements the current thread's execution level
}
```

Figure 7. Generated level-shifting operations surrounding compiled advice bodies and `if` pointcuts.

desired semantics is as follows: `proceed` should imply going down one level only for the *last* `proceed` in the chain of advice. This is because that last occurrence of `proceed` triggers the execution of the original base computation. Fortunately, the AspectJ compilation strategy for around advice [20] makes it fairly straightforward to obtain the proper semantics.

First of all, the original computation is extracted by the AspectJ compiler into its own synthetic method in the woven class. In a chain of advice, `proceed` calls are rewritten to invocations of a *proceed closure*. This closure embeds either a call to the next advice method in the chain, or a call to the synthetic method that encapsulates the original computation. Therefore, all we have to do is to insert level-shifting operations in this synthetic method: invoking `ExecutionLevel.down()` upon method entry—thereby ensuring that base computation happens at its original level—and then calling `ExecutionLevel.up()` upon method completion—ensuring that the advice code after the `proceed` statement runs at the upper level.

4.6 Accessing Aspect Instances

MAJOR2 supports singleton aspect instances as well as multiple aspect instances that are deployed at different execution levels. Other kinds of non-singleton aspect instances are not supported at the moment, such as per-object or per-control flow aspect associations (*i.e.* `per*` clauses is AspectJ). The AspectJ compiler generates a static field and the public static method `aspectOf` in each compiled aspect class in order to hold and access the singleton aspect instance. If only a single instance of an aspect is deployed (either at one or at multiple execution levels), the generated `aspectOf` method remains unchanged.

However, if multiple instances of an aspect are deployed (*i.e.* at different execution levels), we need a mechanism to access the aspect instance of the current level[8]. To this end, MAJOR2 generates extra static fields to hold the specified aspect instances and generates a `switch` statement in the `aspectOf` method (similar to the code pattern illustrated in Figure 6), in order to return the aspect instance that corresponds to the current thread's execution level. That is, if the current thread is executing at level k, the aspect instance deployed at level $k + 1$ is returned (or `IllegalExecutionLevelError` is thrown if the aspect has not been deployed at level $k + 1$).

Aspect instantiation is performed in a special static method m that is generated by the AspectJ compiler and invoked by the class initializer of the compiled aspect class. MAJOR2 treats the class initializer of compiled aspect classes specially so as to ensure that method m is invoked independently of the current execution level of the thread that executes the aspect's class initializer. If multiple instances of the aspect are deployed, the body of method m is extended in order to create all required aspect instances.

[8] Note that our approach to support multiple aspect instances relies on the fact that an aspect has been designed to possibly be instantiated several times, just like with `per*` clauses in AspectJ. In particular, the use of shared (static) state must be considered with care.

4.7 Optimizations: Avoiding Useless Code Versions

Up to now, we have discussed the case of code that can be called at *any* execution level, such as methods in shared libraries like the Java class library. In the full-fledged formulation of execution levels [31], level-shifting operators make it possible for any code to be called from virtually any execution level.

Interestingly, our current design of execution levels for AspectJ does not feature explicit level shifting (recall Section 3.2). In particular, it is not possible to move some computation *down* to the level below the current one. Therefore, execution of code that is statically declared to "belong" to level k (such as the codebase of an aspect deployed only at levels $\geq k$) cannot happen at a lower level than k. It can happen at a higher level, though, because of the possibility of aspects to invoke methods on objects associated with a join point (*e.g.,* calling `hashCode` on the target object of a call). Therefore, it is possible to optimize the previously described weaving process by only inserting the code versions of higher levels. In the scenario of Figure 4b, this means that we need only a single code version for the `Prof` codebase, and only two for `Racer`.

While the aforementioned optimization is sound and easy to perform automatically, it might miss some opportunities for further reducing the set of generated code versions. Therefore, we also allow the user to explicitly specify that some codebases are never invoked at certain execution levels, resulting in more aggressive optimizations. For example, many analysis aspects do not invoke any (possibly overridden) method of the base program; if they acquire references to objects of the base program, they only use the identity of these objects (and possibly the identity hash codes as returned by the static method `System.identityHashCode`). Consequently, in this case, only a single code version needs to be generated for the methods in the base program (woven with the aspects deployed at level 1).

In the code pattern illustrated in Figure 6, jumps to code versions that have not been generated go to the `Error` label. That is, erroneous user configurations are explicitly signaled by throwing an error, instead of executing a wrong code version.[9]

4.8 Limitations

Currently, MAJOR2 does not support AspectJ constructs for static crosscutting, *i.e.* for explicit structural class transformations, such as the insertion of fields or changes to the class hierarchy. In addition, as mentioned before, `per*` clauses are not supported. Also, while `cflow` pointcuts are supported, they are not yet level-sensitive. Similarly, we are currently exploring level-sensitive exception throwing and handling. Finally, code generated by AspectJ within woven methods, such as type checks in residues or the allocation of dynamic join point instances, do not produce any join points that could be matched by aspects deployed at the next higher level; this is consistent with the semantics of AspectJ, but it could be interesting to expose this computation to some analysis aspects.

Threads used in shared libraries, such as Swing event-handler threads, execute at the level at which they have been created. This can raise issues because eventual callbacks on request objects execute at that level as well. Supporting level-capturing objects [31] would address the issue by ensuring that callback objects run at the level at which they were created, rather than at the level of the handler thread. Similar issues can happen with finalizers, which should therefore be avoided. An alternative is to use weak references and create separate threads for handling weak references that are cleared by the garbage collector.

[9] This approach also helps debugging analysis aspects: if the programmer believes and specifies that an aspect never invokes any method of the base program, an error will be thrown if the aspect violates this assumption.

The JVM specification [22] imposes various restrictions on class files, such as a limit of 2^{16} bytes for method bodies. Consequently, even though MAJOR2 generates jumps with 32 bit offsets, the code versions merged into a single method body may exceed that limit. While this limitation affects any instrumentation tool that inserts bytecodes (*e.g.,* it affects the standard AspectJ weaver), the merging of code version into a single method body aggravates the issue. It can be solved by placing code versions in separate private methods when the method size limit is exceeded.

5. Evaluation

In this section, we first validate our implementation of execution levels for AspectJ in the use case scenarios we first discussed in Section 2 and illustrated in Figure 4. Afterwards, we measure the size of woven code and compare the performance of code woven with MAJOR2 respectively with the standard AspectJ weaver.

Benchmarks and environment. For the evaluation, a selection of benchmarks (antlr, hsqldb, jython, luindex, and lusearch) from the DaCapo suite (dacapo-2006-10-MR2)[10] with small workload size serve as base programs. Our measurement environment is a 16-core machine (Sun Fire X4450 Server, 4 quad-core Intel Xeon CPUs, 2.4 GHz, 16 GB RAM) running CentOS Enterprise Linux 5.3 and the Oracle JDK 1.6.0_18 Hotspot Server VM (64 bit version with default settings). We are using MAJOR2 with AspectJ 1.6.5. We are weaving the aspects `Racer` and `Prof` presented in Section 2.

Handling the scenarios. To validate the soundness of our approach, we analyzed the output of the dynamic analyses in the four scenarios illustrated in Figure 4. The output of `Racer` is a list of potential data races (typically empty for single-threaded programs, although finalizer threads may potentially introduce data races). The output of `Prof` is the number of profiled object allocations.

In *scenario 1* (Figure 4a), both `Racer` and `Prof` are deployed at execution level 1. In order to validate that the aspects do not interfere with each other, we first deployed only `Racer` and collected the output; next we deployed only `Prof` and gathered the profile. Afterwards, we deployed both aspects at the same level and collected the combined output. For all our benchmarks, the output of `Racer` is not affected by the presence of `Prof`.

For antlr, luindex, and lusearch, the number of object allocations reported by `Prof` is not affected by the presence of `Racer` either. For hsqldb, we can observe small differences of approximately 10 object allocations. However, this difference comes from the non-determinism of the program itself[11]: re-running hsqldb with the same aspect(s) never gives the same exact number of reported object allocations; the observed difference is within the range of the non-determinism.

For jython, we obtain an unexpectedly high difference of 11354 object allocations. An examination of the benchmark code reveals that initially, jython analyzes the archives in the class path, including the Java class library (`rt.jar`) and also the archives comprising the classes of the deployed aspects. Since MAJOR2 statically weaves the Java class library, jython analyzes a larger `rt.jar` when `Racer` is deployed. This example shows the limits of a code transformation approach: for applications that rely on some form of reflection, the extra code inserted by the weaver may cause perturbations. Because the measurement of object allocation in jython is significantly perturbated by inserted code, we omit that benchmark in the validation of the following scenarios.

[10] http://dacapobench.org

[11] In many JVM implementations, identity hashcodes are random numbers, and random number generators are sources of non-determinism. For multi-threaded programs, thread scheduling is another source of non-determinism. hsqldb is a heavily multi-threaded benchmark [6].

	Size [MB]	**Increase factor**
Scenario 1	95.68	**2.06**
Scenario 2	104.36	**2.24**
Scenario 3	105.56	**2.32**
Scenario 4	116.40	**2.56**

Table 1. Size of the woven Java class library (rt.jar) in the four scenarios (no compression). The original size is 46.45 MB

In *scenario 2* (Figure 4b), `Racer` is deployed at level 1 and `Prof` is deployed at level 2. The output of `Racer` effectively does not change when `Prof` is deployed on top of it.

In *scenario 3* (Figure 4c), `Racer` is deployed at level 1, and the same instance of `Prof` is deployed at both level 1 and level 2. For antlr, luindex, and lusearch, we confirmed that the reported number of object allocations corresponds to the sum of the object allocations reported in scenario 1 and in scenario 2. For hsqldb, we again notice a small difference due to non-determinism. This observation confirms the consistency of our results.

In *scenario 4* (Figure 4d), two instances of `Racer` are deployed, one at level 1 and the other at level 2. First, we refactor `Racer` to avoid any static fields, which would otherwise introduce unwanted interference between the two `Racer` instances. Then, in order to show the benefits of self-application, we introduce a simple data race in its implementation. More precisely, we drop synchronization for an access to a shared data structure that updates the finite-state machines for data race detection within the method `checkRaceUponFieldSet`. The broken `Racer` is still stable enough to produce valid results in our environment. Then, using the multi-threaded benchmark lusearch as base program, the instance of `Racer` at level 2 is indeed able to report the simple data race we introduced within `Racer` [12]. This illustrates how an analysis aspect can be used to analyze itself, without circularity issues.

Code bloat. Our implementation of execution levels in MAJOR2 can introduce significant code bloat, in particular for shared libraries where often all code versions are needed. In order to quantify code bloat, we measured the size of the woven Java class library (`rt.jar` in our JVM) for the four aforementioned scenarios. In the first scenario, two code versions are generated. In the other scenarios, three code versions are produced.

Table 1 presents the size of the woven Java class library (without any compression). In the first scenario, the woven Java class library is about twice as large as the unmodified version. The generated two code versions, the `switch` statement upon method entry, the invocations of advice methods, and the inserted static fields to hold instances of type `JoinPoint.StaticPart` contribute to the increase in size, which is however mitigated by the fact that some parts of the class files, such as the constant pools, are not significantly enlarged.

The additional increase in size for the other scenarios is relatively small, although they require three code versions. This is explained by the fact that object-oriented programs have many small methods. For such methods, the code bloat is often dominated by the inserted `switch` statement. Furthermore, comparing scenarios 1 and 2, the invocations of advice methods in `Prof` are not duplicated, but moved from code version 0 to code version 1.

Surprisingly, code bloat in scenario 4 is worse than in scenario 3, although only a single aspect is woven. In fact, weaving with the `Racer` aspect deployed at level 1 respectively

[12] As reported in [1], the original release of `Racer` had a race condition. However, the algorithm would not be able to detect that particular race condition, because it was not a simple data race (but a general race condition).

	Orig.	AspectJ		MAJOR2	
	[ms]	[ms]	ovh.	[ms]	ovh.
antlr	112	6187	**55.24**	6313	**56.37**
hsqldb	594	55594	**93.59**	59038	**99.39**
jython	157	122401	**779.62**	129810	**826.82**
luindex	129	40142	**311.18**	42819	**331.93**
lusearch	235	87616	**372.83**	89948	**382.76**

Table 2. Overhead comparison for scenario 1: AspectJ versus MAJOR2 (wall-clock time in milliseconds and overhead factor)

at level 2 yields equivalent bytecode. However, in the current implementation, the renaming of inserted static fields to hold `JoinPoint.StaticPart` instances prevents identification of equivalent code versions. We are now exploring whether semantically equivalent `JoinPoint.StaticPart` instances in different code versions should be replaced by a single instance.

Runtime overhead. Here we explore the overhead introduced by the management of execution levels (*i.e.* the `switch` statement generated within method bodies and the inserted level-shifting operations in advice methods). We measure execution time for scenario 1 and compare it with measurements using the standard AspectJ weaver. For a fair comparison, both settings must weave the aspects in the same set of classes. To this end, it is necessary to restrict the scope of the two aspects to exclude each other's classes and to exclude the Java class library, because otherwise the standard AspectJ weaver would break it (recall Section 2)[13].

The measurements for our benchmarks are presented in Table 2. In order to attenuate the impact of just-in-time compilation and garbage collection, for each benchmark we report the median of 15 runs within the same JVM process. The extra overhead introduced by MAJOR2 is insignificant compared to the excessive overhead due to the actual dynamic analyses. If we take the execution time with the AspectJ weaver as a baseline, MAJOR2 introduces an overhead of 2–7%. Note that in general, `Prof` introduces only minor overhead, while `Racer` is a very expensive dynamic analysis that requires some access to shared state upon each field access.

6. Related Work

There is a considerable body of literature on aspect composition and interaction issues [10]. Many proposals focus on the shared join point problem, whereby two aspects affect the same join points [5, 16, 25, 30]. Sanen *et al.* present a classification of aspect interactions [28], which includes: conflicts (semantical interference), dependencies (aspects that need other aspects), reinforcement (aspects influencing correct working of others) and mutex (interaction type of mutual exclusiveness). Our work is concerned with the first kind of interactions, *i.e.* conflicts: a conflict happens when an aspect works correctly in isolation, but fails to work when composed with other aspects. Some work on aspect conflicts go beyond syntactic conflicts, for instance by relating these issues to feature interaction [23], or detecting semantic interference through annotations [36]. Also, work on explicit effect analysis [11] can be used to detect semantic interactions.

This work however departs from all this body of literature by focusing on what we call *computational interference*: the mere computation of an aspect can interfere with others and thereby affect their output. As we have seen, execution levels [31] provide an elegant and effective solution to this issue. This article builds on previous work on execution levels by addressing their practical and efficient integration in a mature and compiled aspect language. Execution levels themselves are rooted in work on metalevel architectures [15], in particular the meta-helix architecture of Chiba *et al.* [12]. The work on stratified aspects of Bodden *et al.* [8] is a first step to address the particular issue of infinite regression of aspects by declaring aspects on different levels. This is very similar to execution levels but differs in one fundamental point: with stratified aspects, levels are *static* properties of code (classes at level 0, aspects at level 1, meta-aspects at level 2, etc.). Because this approach fails to recognize levels as a property of the *execution flow*, it is unable to support code that may run at different levels. This is fundamental to be able to handle callbacks from aspects to application code as well as shared libraries.

Regarding instrumentation of shared libraries, the Twin Class Hierarchy [18] replicates the full hierarchy of instrumented classes into a separate package that coexists with the original one. As pointed out in [32], the use of replicated classes limits the applicability of instrumentation in the presence of native code. Thus, this approach does not allow transparent instrumentation of the complete Java class library. In contrast, MAJOR2 does not duplicate any class, but relies on code duplication within method bodies, an estabished technique that has been successfully used by the instrumentation framework FERRARI [7], the initial version of MAJOR [35], and the dynamic AOP system HotWave [34].

7. Conclusion

Aspect-oriented programming is particularly interesting to define various kinds of dynamic analyses. A major attraction is the use of a common weaving infrastructure with a high-level interface. To fully realize the potential of dynamic analysis aspects, however, it is important to be able to reuse and compose different analyses together in a robust and flexible manner. Current approaches to aspect weaving fail to support this vision because of the computational interference effect: the mere presence of an aspect perturbates the observations made by other aspects, because of the extra join points that are produced and observable by all aspects. Execution levels provide a simple and effective framework to address this issue. This paper has shown how execution levels can be integrated into AspectJ, and has demonstrated with a case study and different composition scenarios that this integration is indeed practical. Future work includes addressing the current limitations of the MAJOR2 implementation, in particular the lack of support for level-capturing objects and level-sensitive `cflow`, and studying a larger body of analysis aspects.

Acknowledgments

The work presented in this paper has been supported by the Swiss National Science Foundation.

References

[1] D. Ansaloni, W. Binder, A. Villazón, and P. Moret. Parallel Dynamic Analysis on Multicores with Aspect-Oriented Programming. In AOSD 2010 [3], pages 1–12.

[2] D. Ansaloni, W. Binder, A. Villazón, and P. Moret. Rapid Development of Extensible Profilers for the Java Virtual Machine with Aspect-Oriented Programming. In *WOSP/SIPEW 2010: Proceedings of the First Joint International Conference on Performance Engineering*, pages 57–62. ACM Press, Jan. 2010.

[3] *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, Rennes and Saint Malo, France, Mar. 2010. ACM Press.

[13] In addition, the benchmark results given here use a refactored and optimized version of `Racer` that does *not* use the control flow check in the `scopeRacer` pointcut. With the control flow check, AspectJ is much slower than MAJOR2 (taking MAJOR2 as a baseline, the AspectJ overhead ranges from 18% for hsqldb to over 80% for antlr and luindex).

[4] A. Assaf and J. Noyé. Dynamic AspectJ. In *Proceedings of the 4th ACM Dynamic Languages Symposium (DLS 2008)*, Paphos, Cyprus, July 2008. ACM Press.

[5] L. Bergmans, M. Akşit, and B. Tekinerdogan. Aspect composition using composition filters. In *Software Architectures and Component Technology: The State of the Art in Research and Practice*, pages 357–382. Kluwer Academic Publishers, 2001.

[6] W. Binder, D. Ansaloni, A. Villazón, and P. Moret. Parallelizing Calling Context Profiling in Virtual Machines on Multicores. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 111–120, New York, NY, USA, 2009. ACM.

[7] W. Binder, J. Hulaas, and P. Moret. Advanced Java Bytecode Instrumentation. In *PPPJ'07: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, pages 135–144, New York, NY, USA, 2007. ACM Press.

[8] E. Bodden, F. Forster, and F. Steimann. Avoiding infinite recursion with stratified aspects. In *Proceedings of Net.ObjectDays 2006*, Lecture Notes in Informatics, pages 49–54. GI-Edition, 2006.

[9] 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*, pages 155–165, New York, NY, USA, 07 2008. ACM.

[10] L. Bussard, L. Carver, E. Ernst, M. Jung, M. Robillard, and A. Speck. Safe aspect composition. In J. Malenfant, S. Moisan, and A. Moreira, editors, *Object-Oriented Technology: ECOOP 2000 Workshop Reader*, volume 1964 of *Lecture Notes in Computer Science*, pages 205–210. Springer-Verlag, 2000.

[11] B. C. d. S. Oliveira, T. Schrijvers, and W. R. Cook. EffectiveAdvice: discplined advice with explicit effects. In AOSD 2010 [3], pages 109–120.

[12] S. Chiba, G. Kiczales, and J. Lamping. Avoiding confusion in metacircularity: The meta-helix. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag, 1996.

[13] R. Chitchyan, J. Fabry, S. Katz, and A. Rensink. Editorial for special section on dependencies and interactions with aspects. In *Transactions on Aspect-Oriented Software Development V*, volume 5490 of *Lecture Notes in Computer Science*, pages 133–134. Springer-Verlag, 2009.

[14] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, 30(3):Article No. 14, May 2008.

[15] O. Danvy and K. Malmkjaer. Intensions and extensions in a reflective tower. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 327–341, Snowbird, Utah, USA, July 1988. ACM Press.

[16] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188, Pittsburgh, PA, USA, Oct. 2002. Springer-Verlag.

[17] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, Dec. 2006.

[18] M. Factor, A. Schuster, and K. Shagin. Instrumentation of standard libraries in object-oriented languages: The twin class hierarchy approach. In *OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 288–300, New York, NY, USA, 2004. ACM.

[19] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, 2005.

[20] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In K. Lieberherr, editor, *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 26–35, Lancaster, UK, Mar. 2004. ACM Press.

[21] C. Kung and C. Ju-Bing. Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs. In *Computer Software and Applications Conference, 2007. COMPSAC 2007*, pages 23–28, Beijing, China, 2007. IEEE Computer Society.

[22] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[23] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A disciplined approach to aspect composition. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 68–77, New York, NY, USA, 2006. ACM.

[24] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC 2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[25] I. Nagy, L. Bergmans, and M. Aksit. Composing aspects at shared join points. In *NetObjectDays (NODe 2005)*, Lecture Notes in Informatics 69, pages 19–38, Erfurt, Germany, Sept. 2005.

[26] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, June 2007.

[27] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th ACM Symposium on Foundations of Software Engineering (FSE 12)*, pages 147–158. ACM Press, 2004.

[28] F. Sanen, E. Truyen, B. D. Win, W. Joosen, N. Loughran, G. Coulson, A. Rashid, A. Nedos, A. Jackson, and S. Clarke. Study on interaction issues. Technical Report AOSD-Europe Deliverable D44, AOSD-Europe-KUL-7, Katholieke Universiteit Leuven, 28 February 2006 2006.

[29] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[30] É. Tanter. Aspects of composition in the Reflex AOP kernel. In W. Löwe and M. Südholt, editors, *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *Lecture Notes in Computer Science*, pages 98–113, Vienna, Austria, Mar. 2006. Springer-Verlag.

[31] É. Tanter. Execution levels for aspect-oriented programming. In AOSD 2010 [3], pages 37–48. Best Paper Award.

[32] E. Tilevich and Y. Smaragdakis. Transparent program transformations in the presence of opaque code. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 89–94, New York, NY, USA, 2006. ACM.

[33] R. Toledo, P. Leger, and É. Tanter. AspectScript: Expressive aspects for the Web. In AOSD 2010 [3].

[34] A. Villazón, W. Binder, D. Ansaloni, and P. Moret. Advanced Runtime Adaptation for Java. In *GPCE '09: Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, pages 85–94. ACM, Oct. 2009.

[35] A. Villazón, W. Binder, P. Moret, and D. Ansaloni. Comprehensive Aspect Weaving for Java. *Science of Computer Programming*, 2010. `http://dx.doi.org/10.1016/j.scico.2010.04.007`.

[36] A. Zambrano, S. Gordillo, and J. Fabry. A fine grained aspect coordination mechanism. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, December 2010. To appear.